SMART SWAMP COOLER

Design Documentation

Team: Jayden Smith
Jeniffer Limondo
Maxwell Cox

Project: Smart Swamp Cooler

Date Prepared: April 14, 2021

Table of Contents

List of Figures

Revision History

Date	Author(s)	Revision Made
April 14, 2021	**Smith, Jayden Limondo, Jeniffer**	**Original Document**

List of Abbreviations and Definitions

API	Application Programming Interface
CSS	Cascading Style Sheets
LAN	Local Area Network
MC PCB	Master Controller Printed Circuit Board
RPi SBC	Raspberry Pi Single Board Computer
RXD	Receive Data
SSC	Smart Swamp Cooler
TXD	Transmit Data
UART	Universal Asynchronous Receiver/Transmitter
UI	User Interface
VNC	Virtual Network Computing
WPAN	Wireless Personal Area Network

1.0 Introduction

Evaporative coolers are an energy efficient way of cooling living spaces in hot and dry parts of the country, such as the desert regions of the western United States. In these regions, evaporative coolers use about 15-35% the energy of an A/C system. However, because of the relative humidity and temperature of ambient air, these coolers have a limit to their cooling capacity.

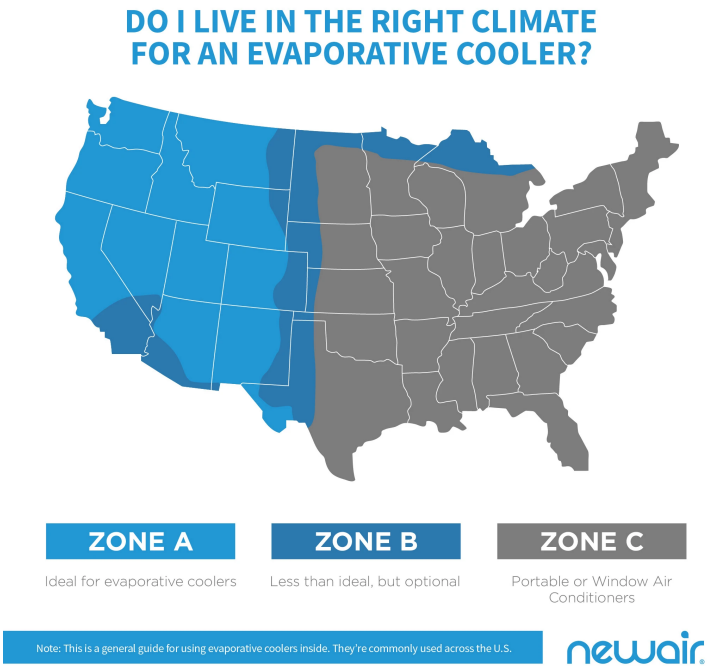

Figure 1: Where are Evaporative Coolers Used

Most installations of evaporative coolers are manually controlled, leaving it up to the building occupants to decide the optimum operation of the cooling system. Knowing when to turn on which setting can be a very complicated and cumbersome process for users, leading to highly inefficient utilization of their coolers. This project designs a smart, microprocessor-based, control system to maintain efficient operation of evaporative coolers.

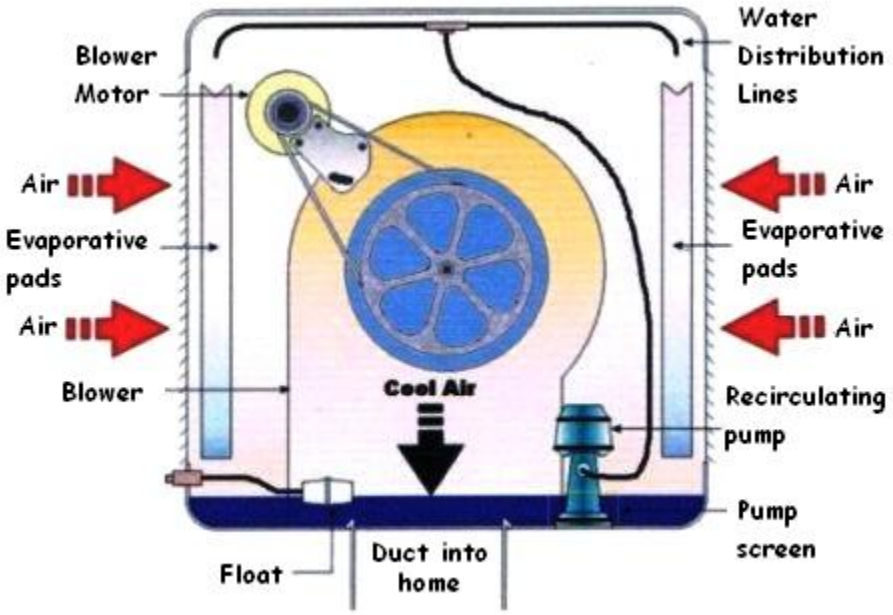

Figure 2: Current Swamp Coolers

At the conclusion of the project, a demonstration smart cooling unit is presented consisting of a central controller and a plurality of sensors, collectively known as the Smart Swamp Cooler (SSC). The central controller is connected to the sensors and will autonomously change the cooler's water pump and fans to appropriate settings. The UI on the central controller accepts user input to adjust the cooler's settings. A web UI is provided that can also change or set cooler settings and provides a log of sensor data and fan settings for the most recent 1-31 days, depending on the user's selection. If the sensors are not placed in the evaporative cooler, they will have their own enclosures.

2.0 Scope

This document describes the software and hardware design, implementation, and testing of a smart evaporative cooler. It includes the requirements, dependencies, and theory of operation. Design details and requirement validation testing are also described. Complete software code and hardware schematics are included in the appendices.

This project designed a smart, microprocessor-based, control system to maintain efficient operation of an evaporative cooler. The cooling unit has temperature and humidity sensors that connect to a central microprocessor controller via a wireless Zigbee microprocessor. Based on the input from these sensors, the controller adjusts the water pump flow and fan settings using mechanical relays.

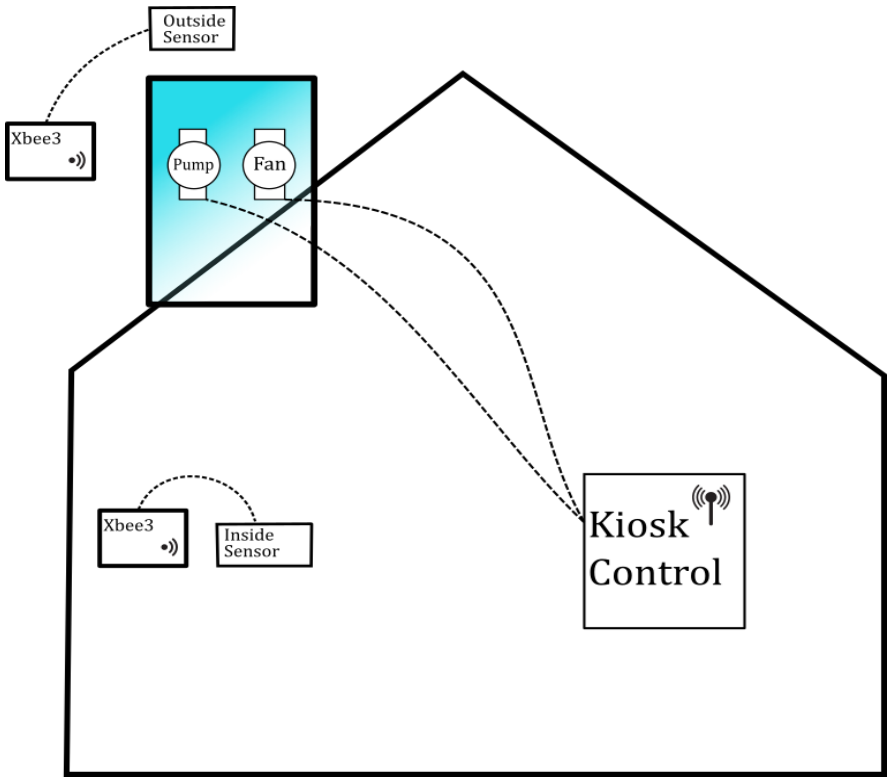

Figure 3: Proposed Solution

The focus of the SSC project was on the execution of an efficient algorithm for the cooler's functionality. A user interface on the central controller allows

adjustments to any of the cooler's settings, displays the upcoming local forecast, most recent temperature and humidity readings from both indoor and outdoor sensors, and maintains a running log of all changes in the cooler's settings and temperature and humidity sensor readings. A web UI is also provided to emulate the user interface on the central controller with full functionality, accessible on any device connected to the same network as the main controller.

The Stakeholders in the project are the following:

Project Advisor:
Ronald Jensen

Faculty Advisor:
Dr. Fon Brown

Engineers:
Jeniffer Limondo
Jayden Smith
Maxwell Cox

Financial Backers:
Office of Undergraduate Research (OUR)
Ronald Jensen
Jayden Smith

Customer/User:
Ronald Jensen
Jayden Smith

The roles of the engineers were as follows: Jeniffer focused primarily on electrical engineering and electrical power circuits, Max performed general software programming, and Jayden performed general software and

hardware design and project management. The project advisor oversaw minor issues that arose during the project.

Things that one might expect to be delivered that are not delivered at the completion of this product include packaging, mechanical drawings, and labeling of the enclosure and microprocessor.

3.0 Design Overview

The SSC has the following components:

1. Master Controller (MC) powered by Raspberry Pi 4
   a. Zigbee 3 Coordinator/Master
   b. Mechanical Relays
   c. LCD Touchscreen Display
   d. Kiosk & Web Interface
   e. Smart Algorithm

2. Wireless Modules
   a. Zigbee 3 Routers/Slaves
   b. HDC1080 Temperature & Humidity Sensors
   c. Solar Panel for Outside Module

3.1 Requirements

The given requirements for the SMART SWAMP COOLER are as follows:

1. The system shall control a roof mount permanently installed swamp cooler.
2. The system shall utilize relays to control pump, fan, and fan speed.
3. The system shall run on an external 120 VAC supply.
4. The system shall operate in manual or automatic mode. Automatic mode provides a smart algorithm to determine the best cooler setting.
5. The system shall use a 1024x600 Touch Screen IPS Display.
6. The system shall use the Raspberry Pi 4 Model B w/4 GB RAM
7. The system shall use a custom PCB containing relays, power supply, and connectors required to connect the system to an existing swamp cooler.
8. The system shall use three Zigbee 3 microcontrollers for distributed communication with remote temperature sensors. Two Zigbees need to wirelessly connect to the third Coordinator Zigbee

9. The system shall provide a user interface using Flask running under an Apache2 web server.
10. The system web service shall be accessible over the house LAN utilizing Wi-Fi.
11. The system shall operate in a kiosk mode for simple user interaction.
12. The system shall be capable of connecting to the National Weather Service to obtain forecast temperatures for a smart algorithm.
13. The system shall utilize sensor data and forecast temperature to provide the best operating mode for the external swamp cooler.
14. The system shall use MySQL to provide logging of system data.
15. The main system shall be enclosed in a 3D printed case capable of being attached to a building wall.
16. The remotes sensor systems shall be enclosed in 3D printed cases.
17. The external remote sensor shall utilize a solar panel for battery longevity.

3.2 Constraints

3.2.1 Cost

A final cost for producing this product at a commercial level was not specified, but the entire prototype system needed to cost under $200 to fabricate from beginning to end. A grant proposal was submitted and approved by the Office of Undergraduate Research, which was used to develop and test various prototypes and to provide backup parts for every part of the system.

3.2.2 Scheduling

The project needed to be designed, tested, and delivered in 7 months maximum. All testing needed to be conducted with a portable swamp cooler because winter weather was not conducive to testing a roof-mounted evaporative cooler. Much of the data had to be constructed so as to simulate weather conditions of warmer and more humid weather.

3.2.3 Software

As much as possible, the software for the Smart Swamp Cooler needed to be written in Python. Micropython was requested to be used on the Zigbee modules, a Python Flask backend was used to develop the web service, and Python scripting was used to manage database management, smart algorithm implementation and testing, and relay control.

3.3 Applicable Standards

3.3.1 IEEE 802.15.4

The wireless Zigbee 3 modules build on the physical layer and media access control defined in IEEE standard 802.15.4 for low-rate WPANs. Communication between wireless routing modules and the coordinating Zigbee module connected to the MC PCB followed this standard.

3.3.2 SPI

The HDC1080 Humidity & Temperature sensors communicate through SPI communication protocol to the wireless Zigbees.

3.3.3 IEEE 802.11ac

The Raspberry Pi 4 Model B provides access to the touchscreen kiosk on the local network using Wi-Fi IEEE 802.11ac standard.

3.3.4 RS-232

The SSC communicates directly with the Zigbee Coordinating module through a UART channel. The Raspberry Pi 4 outputs TTL levels (0v to 3.3V) which are inverted; a mark is low and a space is high. The sensor data from the wireless Zigbee nodes in the house and on the roof are relayed from the Coordinator Zigbee to the Raspberry Pi through this RS-232 standard.

3.4 Dependencies

The SMART SWAMP COOLER depends on the following:

1. An external 120 VAC supply
2. Various Open Source Software products:
    a. Raspbian Linux operating system
    b. Python 3.7, Micropython, bash, HTML, JavaScript, CSS programming languages
    c. Apache2 Web Server
    d. Flask web framework
    e. MySQL database software with PHPMyAdmin
3. National Weather Service forecast API
4. Local Wi-Fi infrastructure

3.5 Theory of Operation

Two XBee 3 wireless modules are used to gather temperature and humidity data from HDC1080 sensors. These XBees are configured to sample every 15 minutes. The readings are then sent wirelessly to another XBee Coordinator on the main controller circuit. This Coordinating Zigbee continually samples its wireless port until data is received.

Once received, the new sensor readings are forwarded to the RPi controller. The RPi adds these readings to the locally hosted MySQL database. It also services the LCD touchscreen to display the landing page for the SSC, where cooler settings can be manually or automatically configured and where sensor readings and forecasted data can be viewed.

Using the data from the wireless sensor nodes, upcoming forecasted weather, cooler efficiency specifications, and general house settings, the RPi runs a smart algorithm to determine the optimal cooler setting, then turns on/off mechanical relays to realize the desired setting.

When the SSC is first hooked up to a swamp cooler, the main scripts to run the kiosk website and control the XBee devices and relays will start automatically in the background.

The RPi will then load into a Raspbian desktop. A virtual keyboard will pop up. From here, the user is instructed (via the Installation Manual) to click on the Wi-Fi icon in the top-right corner of the screen. They select their desired

network, enter their password, then wait until it connects. Once connected, the user closes the virtual keyboard (via the ‘x’ on the top-right corner of the virtual keyboard), and the SSC main website loads automatically.

The user sees a QR code and IP address that they can either scan or type, respectively, to load the web page on any of their devices connected to the same network. The most recent temperature and humidity settings will display, followed by the options to change the cooler setting (either to AUTO or any of the manually-controlled settings) and desired temperature of the house. They hit submit on their choices, and the background scripts will run the smart algorithm and control the relays to the best possible setting.

Below this on the website, the user can view historical settings of the cooler and historical logs of the wireless sensors. The historical logs can be configured to show various ranges of previous data.

If anything on the website appears to no longer work, the SSC will automatically restart. The user will then either have to reconnect to Wi-Fi or the SSC will automatically reboot the website kiosk.

3.6 Design Alternatives

A few design alternatives were considered and rejected for the Smart Swamp Cooler:

3.6.1 DHT11 Sensors

DHT11 humidity and temperature sensors could have been used, but the communication protocol capabilities of the Zigbee 3 modules were not able to reproduce the needed SPI communication. An additional small Arduino processor would have needed to be added to the wireless nodes, which would increase overall cost, speed, size, and performance of the wireless nodes. HDC1080s were chosen instead because of their natural ability to connect via I2C directly to the Zigbee 3 modules.

3.6.2 RockPro64

RockPro64 SBC's were considered in place of the Raspberry Pi 4 Model B SBC that is employed in the current product. The Pi 4 was chosen because of its popularity, widespread support and community, and overall ease of use. The RockPro64 was tested for a similar project and basic problems with UART functionality, lack of built-in Wi-Fi capabilities, and incompatibility with much of the basic software required for the Smart Swamp Cooler rendered this SBC a poor alternative.

3.6.3 SMD HDC1080

Surface mount HDC1080 sensors were considered and tested initially for this project because of their small size and high efficiency, but the desire to easily replace a faulty or damaged sensor led to the current design of a breakout board for the HDC1080 that can be easily installed or replaced into a header on the wireless PCB.

3.6.4 C for Zigbee Programming

The programming of the Zigbee 3 wireless modules was initially designed to be in the C programming language. The desire for end-to-end consistency at all levels of the software combined with the built-in packages for Zigbee communications in the Micropython programming language proved sufficient reason for the Smart Swamp Cooler to be programmed almost entirely in Python.

4.0 Design Details

This section provides a detailed description of the design and development of the Smart Swamp Cooler. All the necessary information setting up the RPi to work with the Smart Swamp Cooler is provided in the first section. Each of the sections will describe the design, logic, and construction of the two PCBs.

A general overview of the SSC is depicted below:

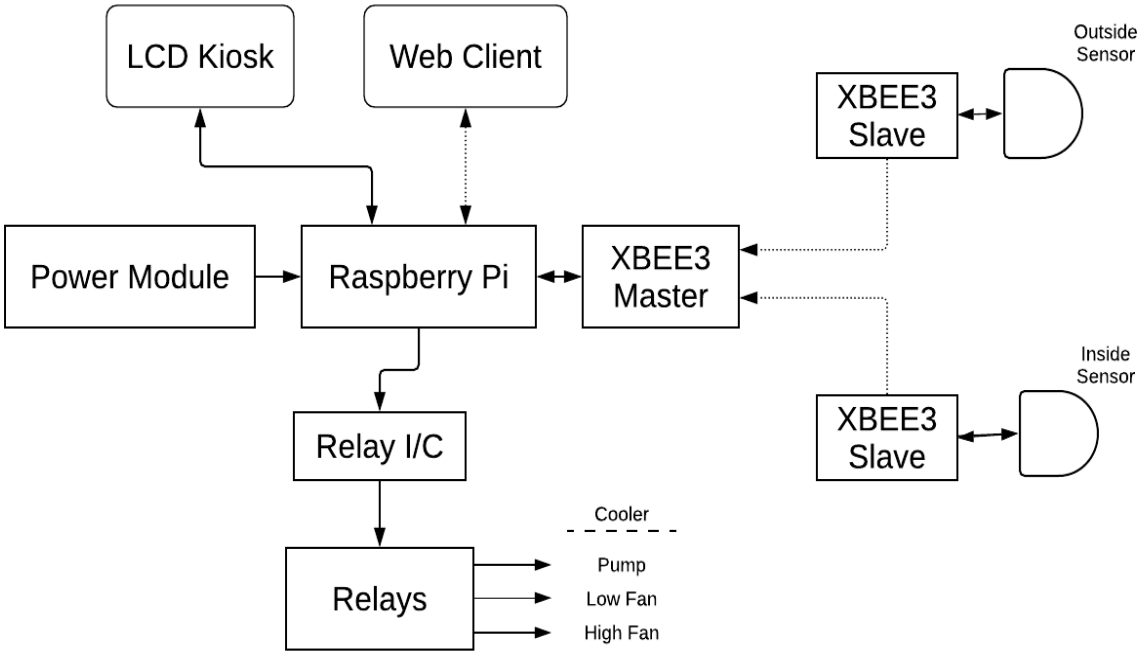

Figure 4: SSC Block Diagram

Two XBee 3 wireless modules (shown as XBEE3 Slaves above) are used to gather temperature and humidity data from HDC1080 sensors. The XBees gather the data from these sensors through their UART pins (see wireless node schematic in Appendices). These XBees are configured for 15 minute intervals in deep sleep, meaning they only sample the HDC1080s once every 15 minutes.

The readings are then sent wirelessly to the XBEE3 Master. This Coordinating Zigbee continually samples its wireless port until data is received. Once received, the new sensor readings are forwarded via UART pins to the RPi controller. The RPi adds these readings to the locally hosted MySQL database. It also services the LCD touchscreen to display the landing page for the SSC, where cooler settings can be manually or automatically configured and where sensor readings and forecasted data can be viewed.

Using the data from the wireless sensor nodes, upcoming forecasted weather, cooler efficiency specifications, and general house settings, the RPi runs a smart algorithm to determine the optimal cooler setting, then turns on/off mechanical relays to realize the desired setting.

4.1 Raspberry Pi

This project uses the RPi SBC (version 4, Model B) as the primary system logic, interface, and control.

4.1.1 The Raspberry Pi GPIO Pins

The Swamp Smart Cooler uses GPIO pins 14, 15, 16, 20, and 21. Pin 14 and 15 are used for serial communication with the Zigbee 3; while pins 16, 20, and 21 are used as logic outputs to drive the relays. The GPIO also receives +5 VDC on pins 2 and 4.

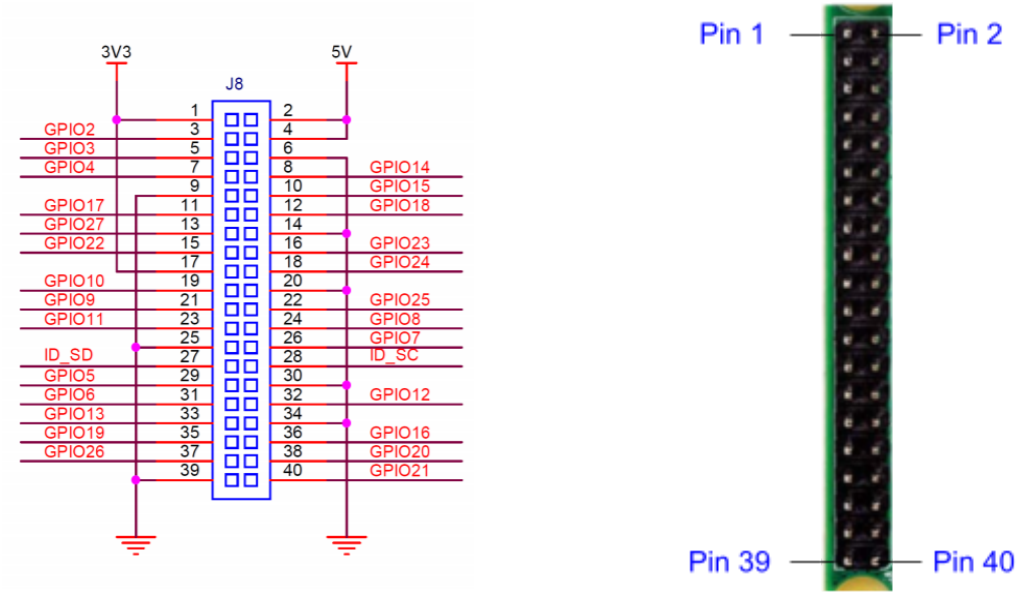

Figure 5: RPi GPIO Pin Out and P1 Header with Numbering

4.1.2 Setting up Raspberry Pi with Raspbian Lite

The RPi is configured using Raspbian Linux. The major packages are: Apache, Flask, mySQL, and Python 3. VNC Viewer was used during development to enable the development team to access the RPi without using the built-in monitor.

This design requires a GUI system on the RPi. Since the RPi does not come with a built-in GUI system, it was required to install it. For this project, Raspbian Lite was used as the GUI system on the RPi. Installing Raspbian involves a few steps. The following website was used as the reference for setting this up and can be followed step-by-step as in [1]. Alternatively, Micro SD cards can be purchased with Raspbian NOOBS software pre-installed. Following these steps or simply inserting the pre-installed software will allow one to install Raspbian Lite on the RPi. After running the last command from the installation steps, the RPi needs to be restarted. On reboot, it should open to the GUI system.

4.1.3 Raspberry Pi Packages

Several packages needed to be installed on the Pi, including the following that do not typically come pre-installed with Raspbian lite: python3, json, re, mysql, gpiozero, reverse-geocoder, Apache2, flask_mysql_connector, matplotlib, geocoder, qrcode, flask, logging, io.

4.2 Hardware

The Smart Swamp Cooler controller is implemented on Raspberry Pi SBC and a customized main board and a user interface consisting of an IPS Touch Screen Display. The IPS Touch Screen Display is an MDS-7B02 display module manufactured by EVICIV that has a resolution of 1024x600 pixels. The main board consists of an IRM-15-5 5 V power supply module, an ULN2803A relay driver chip, three RZ03-1C4-D005 relays, and an Zigbee 3.0 RF interface module.

The board also contains connectors to interface with the house wiring and other components. The two wireless nodes are custom PCB boards containing an LT1300 buck-converter with associated passive components, an HDC1080 temperature and humidity sensor, and a Zigbee 3.0 RF interface module. The wireless nodes are powered by two NiMH 1.2 V AA batteries. The external node also has a small solar panel to keep its batteries charged. The controller and wireless nodes are enclosed in custom 3D printed cases.

4.2.1 IRM-15-5 5V power supply module

The IRM-15-5-5 is a 5 VDC power supply module fed from the house mains via fuse F-1 from J1, a TN-T03G screw terminal connector. It supplies a stable voltage source for the RPi via the GPIO connector, display module via J9, and the main board itself.

4.2.2 RZ03-1C4-D005 relays and ULN2803A relay driver chip

The RZ03-1C4-D005 relays are used to control the 120 V supplied to the swamp cooler. The 120 V is supplied from the house mains via fuse F-2. R1 supplies voltage to the pump motor via its normally open contacts. R2 provides voltage to R3 via its normally open contacts, and is used to enable fan operation. R3 receives input voltage from R2, it outputs the voltage to low speed fan on its normally closed contacts and high speed fan on its normally open contacts. These relays are rated for 16 A at 250 V. The board connects the relays to house wiring via J2, a TN-T03G screw terminal connector.

The ULN2803A relay driver chip contains eight separate drivers, they are used in pairs to drive the three relays, with one pair being unused. The chip is driven by the RPi GPIO pins 16, 20, and 21 for pump, fan on, and fan speed respectively.

4.2.3 Zigbee 3.0 RF interface module

The Zigbee 3.0 RF interface module attaches to the main board via 0.7 mm spaced headers J5 and J6. The main board connects the Zigbee 3 to GPIO 14 (TXD) and GPIO 15 (RXD) for serial communication.

4.2.4 Wireless Nodes

The two wireless nodes are custom PCB boards containing an LT1300 buck-converter with associated passive components to provide 3.3 VDC, an HDC1080 temperature and humidity sensor, and a Zigbee 3.0 RF interface module. The wireless nodes are powered by two AA batteries. The external node also has a small solar panel with a charging circuit controller built-in to keep its batteries charged properly.

4.2.5 3D Fabricated Enclosures

Three different enclosures were designed to house the project using Solidworks software. The Kiosk enclosure was designed to mount on the wall of the house, it holds the RPi, display and main board. The inside sensor will go anywhere in the house, this enclosure houses one of the wireless nodes along with a battery pack. The outside sensor enclosure holds the other wireless node, along with another battery pack and a small solar panel. This sensor is intended to go on the roof near the swamp cooler, so the enclosure will be waterproofed using silicone gaskets to protect it from the elements.

Figure 6: 3D Fabricated Enclosures

4.2.6 Operating Parameters

Figure 7: Typical Swamp Cooler Motor

The swamp cooler motor typically has its operating characteristics printed on a label on the side of the motor. This controller is rated at 16 Amps.

For the smart algorithm to be most accurate, the following values are needed:

- Low and high speed RPMs to estimate the cooler's exhaust flow in cubic feet per minute (CFM). Default values are 70 cfm low and 106 cfm high.

- Cooler efficiency, which is generally estimated to be 0.744.

- Estimated house volume. This can be done by using the square footage times the ceiling height. A typical 1200 sq ft home would be about 10,000 cubic feet.

- Latitude and longitude for weather forecasting, which will be estimated by the RPi once powered on. Default values are 41.1616 -112.026.

Currently, these values need to be written manually to the database by the programmer via the *house_settings* table in PHPMyAdmin, but a second version of this project will allow the user to add in these parameters from the website. A class is defined below in *smartSwampCooler.py* to read these values to and from the database. Another function to map the database object to a Python is shown below the class declaration, and a final function is shown that uses these two functions to actually retrieve the house settings from the database and pass them into the smart algorithm.

```
# Define a class for holding house setting information
# from db to pass into smart algorithm
class HouseSettings:
    def __init__(self, id="-1", latitude=0.0, longitude=0.0, house_volume=0.0,
                 lo_fan_volume=0.0, hi_fan_volume=0.0, efficiency=0.0):
        self.id = id
        self.latitude = latitude
        self.longitude = longitude
        self.house_volume = house_volume
        self.lo_fan_volume = lo_fan_volume
        self.hi_fan_volume = hi_fan_volume
        self.efficiency = efficiency

    # these two functions allow for easy printing of house settings object
    # to check data of a given object...common Python practice
    def __repr__(self):
        return "%s(%r)" % (self.__class__, self.__dict__)

    def __print__(self):
        return "%r" % (self.__dict__)
```

```
# Maps database data to HouseSettings object
def map_to_house_object(data):
  house_settings = HouseSettings()

  try:
    # assign database fields to object
    house_settings.id                    = (data["id"])
    house_settings.latitude              = (data["latitude"])
    house_settings.longitude             = (data["longitude"])
    house_settings.lo_fan_volume         = (data["lo_fan_volume"])
    house_settings.hi_fan_volume         = (data["hi_fan_volume"])
    house_settings.house_volume          = (data["house_volume"])
    house_settings.efficiency            = (data["efficiency"])

  except Exception as ex:
    print("ERROR: Error mapping database data to house_settings object")
    template = "An exception of type {0} occurred. Arguments:\n{1!r}"
    message = template.format(type(ex).__name__, ex.args)
    print(message)
    return house_settings

  return house_settings
```

```
# Read most recent house settings from database
def read_house_settings_db():
  house_settings = HouseSettings()

  # create new connection cursor
  mycursor = smartswampcooler_db.cursor()

  # retrieve house settings from database
  mycursor.execute(SQL_SELECT_HOUSE_SETTINGS)

  # this will extract row headers
  row_headers = [x[0] for x in mycursor.description]
  myresult = mycursor.fetchall()

  # convert SQL entry to Python dict object
  json_data = []
  for result in myresult:
    json_data.append(dict(zip(row_headers,result)))

  data = json.dumps(json_data, indent=4, sort_keys=True, default=str)
  data = json.loads(data)

  #print("Displaying house settings: ")
  # print out all json data entries and map to HouseSettings object
  for i in range(len(data)):
      #print(data[i])
      house_settings = map_to_house_object(data[i])

  return house_settings
```

Figure 8: Reading House Settings from Database

4.2.7 Electrical Connections

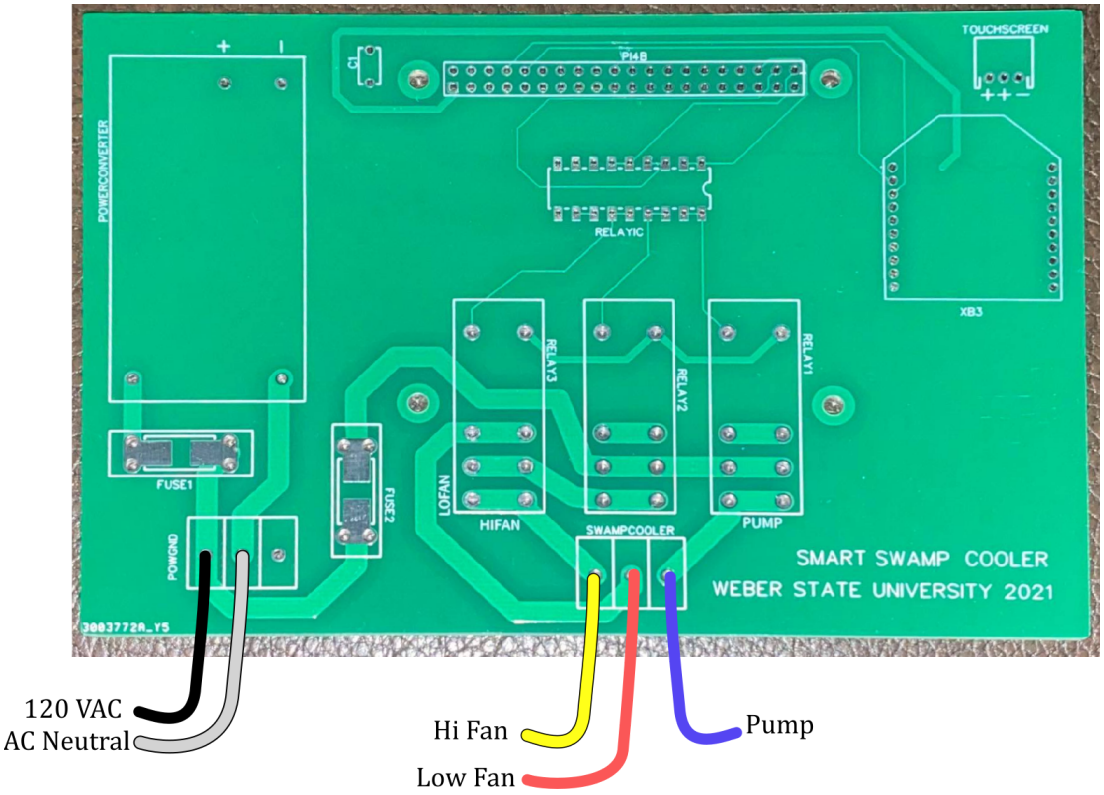

Figure 9: Master Controller Printed Circuit Board

CAUTION: Ensure the swamp cooler circuit breaker is off before working on AC wiring!

The main controller board (pictured in Figures 3 and 4) hooks the Raspberry Pi SBC Smart Swamp Cooler controller into the swamp cooler's current house wiring. The left side of the board is where 120 VAC and AC Neutral enter the board. AC Neutral should also be wire-tied to the fan motor wiring outside the board. Note there are two 15 Amp fuses, one for the 5 Volt supply to the control board, and one for the swamp cooler.

The controller mounting plate should be fastened to the wall before connecting the AC power.

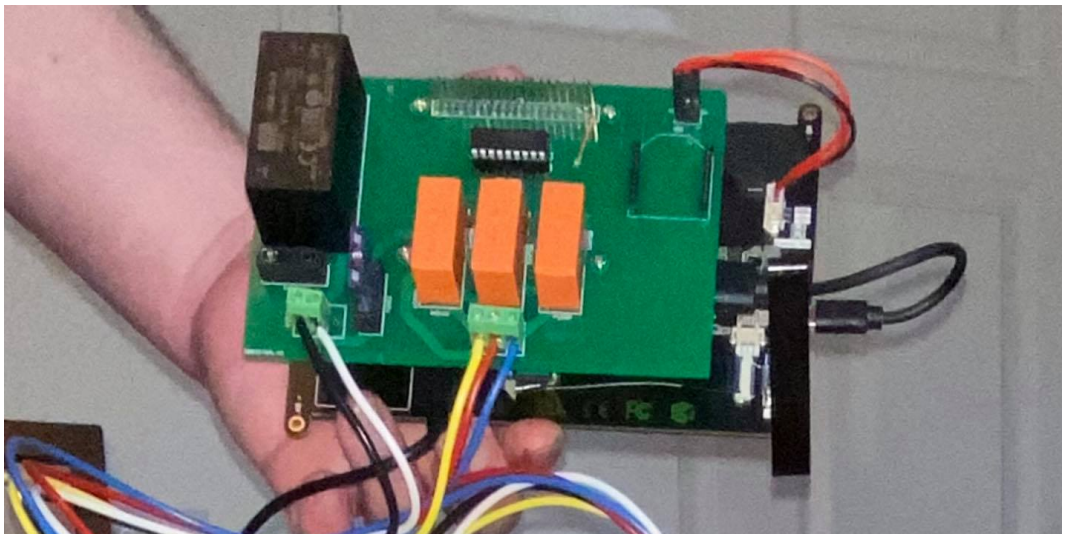

Figure 10: Master Controller Printed Circuit Board Assembled

Make sure that the wires are correctly hooked into the swamp cooler as shown in Figure 4. A typical house schematic wiring diagram for a roof-mounted swamp cooler is shown in Figure 5. This diagram should be compared to the specifications of the user's swamp cooler and the fan and pump wires should be adapted to match the wiring of Figure 3.

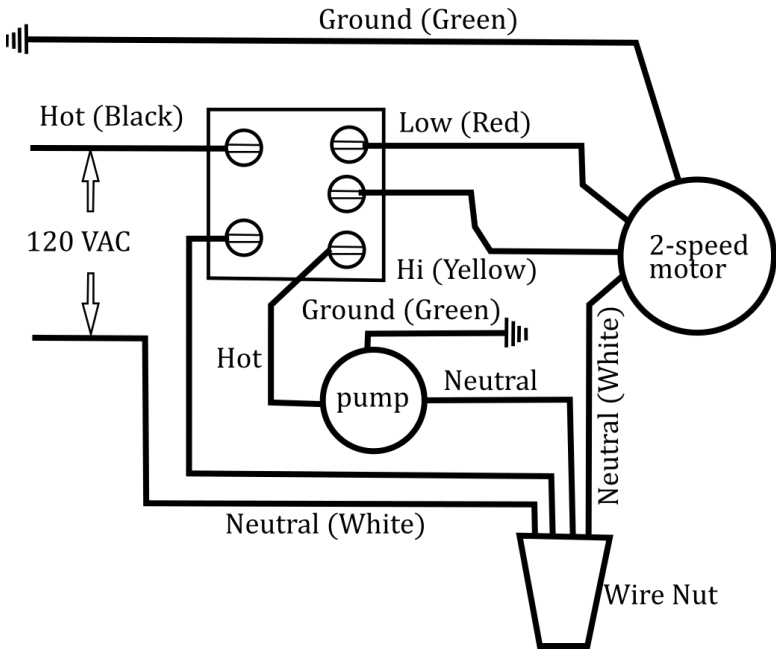

Figure 11: Typical House Schematic

4.3 Software

4.3.1 Software Flow Chart

The SSC software uses a variety of languages, including Python, Bash, HTML, CSS, and JavaScript on the RPi, and MicroPython for the Zigbee Slaves/Routers and Master/Coordinator.

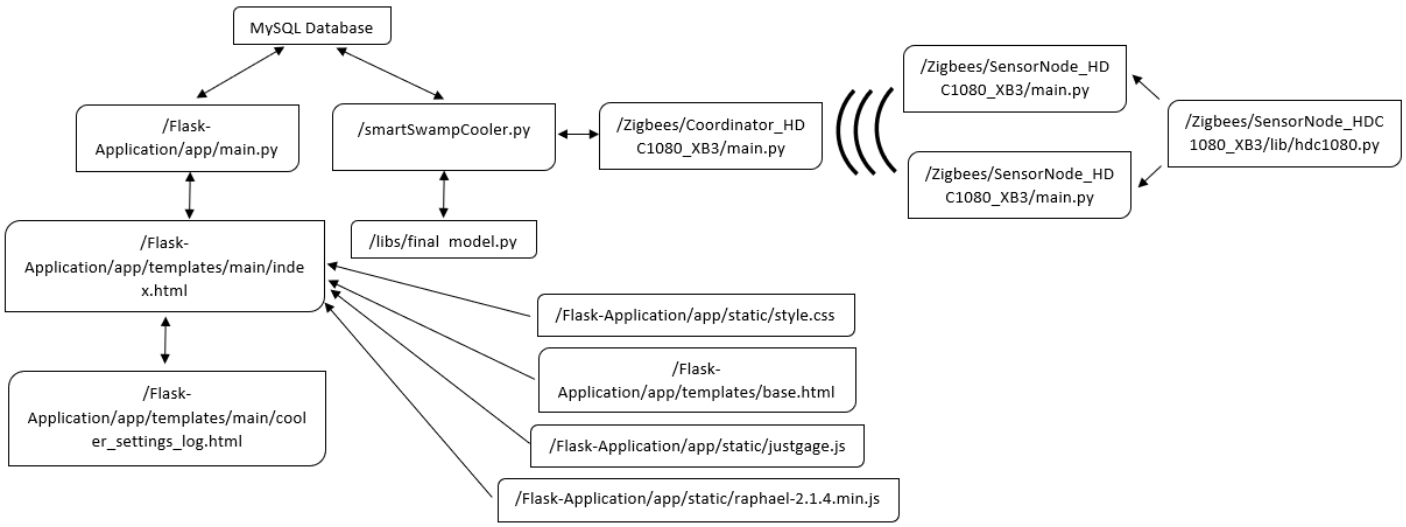

Figure 12: Software Diagram

The above software flowchart shows the major files and scripts that make the SSC function. The files and their locations can be found in the GitHub link provided in the Code Listing.

4.3.2 hdc1080.py and wireless node main.py

From right to left, the hdc1080.py file interfaces with the HDC1080 sensors to gather temperature and humidity data. The main.py runs on the XBee3 acting as a router node receiving temperature and humidity data it collects from the HDC1080, then transmits that data to the main.py running on the Coordinator Zigbee module. This router node samples the sensor every 15 minutes. Deep sleep configuration will be detailed more in a later section.

4.3.3 Coordinator main.py

The Coordinator main.py file runs on the Zigbee receiving data from the wireless nodes. It configures the LAN that the wireless Zigbee nodes can connect to and then forwards the data from the wireless nodes to the RPi through the Zigbee's serial port.

4.3.4 Main script for MC PCB - smartSwampCooler.py

This code runs on the Raspberry Pi 4 Model B acting as the master controller. This is the main script controlling the MC PCB.

4.3.4.1 Reading from the Serial port

It reads data from the Zigbee Coordinator receiving data it collects from the sensor router nodes. This serial communication is done through the serial UART port of the RPi. Functionality is provided for serial settings, opening/closing the port, resetting the buffer, and reading data any time a message is received. The following code snippets show how these serial functions are set up and performed:

```
# ---------------------- Pi 4 Serial Functionality ---------------------- #
#
# Setup for Serial port to match Zigbee protocol
ser = serial.Serial(
    port = '/dev/ttyS0',
    baudrate = 57600,
    parity = serial.PARITY_NONE,
    stopbits = serial.STOPBITS_ONE,
    bytesize = serial.EIGHTBITS,
    timeout = 1
)

# Read and return data in the serial receive buffer
def read_serial():
    return ser.readline()

# Write data out the serial pin
def write_serial(the_char):
    ser.write(the_char)

# Clear the serial read buffer
def flush_read_buffer():
    ser.reset_input_buffer()
```

```
# Wait for a response to be received on serial line or a timeout to occur.
# Return the response (which may often be an empty string if nothing received)
def wait_for_serial_response(timeout):
    response = read_serial()
    timer = 0
    while response == b"" and timer < timeout:
        # check user's settings and desired temperature from database
        # and set up algorithm to implement setting
        coolerSetting, desired_temperature = get_cooler_setting()
        set_cooler(coolerSetting, desired_temperature)

        # count to 10, sampling serial port every second
        print("  . ({})".format(timeout-timer))
        timer += 1
        time.sleep(1)
        response = read_serial()
    return response

# reset serial port by closing port and flushing buffer
def reset_serial_port():
    # close/open serial port
    ser.close()
    time.sleep(1)
    ser.open()

    # flush
    flush_read_buffer()
```

Figure 13: Serial Port Configuration

The data is decoded and converted into a SensorData class object, then converted to be added to the main database storing all sensor readings.

4.3.4.2 Connecting to the MySQL database

Connecting to the database is done as follows:

```
# Connect the mySQL database and provide login credentials
smartswampcooler_db = mysql.connector.connect(
  host="localhost",
  database="smartswampcooler",
  user="pi",
  passwd=           ,
)
```

Figure 14: Connecting to MySQL Database

Once the raw data received from the Zigbee coordinator is decoded and interpreted, this data is sorted into key-value pairs that are used to create SensorData objects. These objects are used as entries into a PHPMyAdmin database using MySQL queries.

4.3.4.3 Decoding and using Zigbee packets

Converting the sensor data from the Zigbees is done in the xbee_to_object(xb_raw_data) function, which is then converted into a SensorData object using the map_to_object(data) function, both of which can be viewed with detailed commentary in the code listing.

4.3.4.4 Database SQL queries used

Functionality is provided to read out the data from the database for the whole database, or for any sensor node and for any number of days in the past. Here are few examples of the SQL queries used and found in this script:

```
# Select all data from the sensor_data table
SQL_SELECT_DATA = "SELECT * FROM sensor_data"

# Insert new row of from SensorData object
SQL_INSERT_DATA = """
 INSERT INTO sensor_data (sensor_id, timestamp, temperature, humidity)
 VALUES (%s, NOW(), %s, %s)
"""

# Select all data from given sensor and interval (# of days previous)
SQL_SELECT_SENSOR_DATA = """
  SELECT *
  FROM sensor_data
  WHERE sensor_id=%s
  AND timestamp >= NOW() - INTERVAL %s DAY
  ORDER BY timestamp
"""
```

```
# Select most recent entry for cooler setting
SQL_SELECT_COOLER_SETTING = """
  SELECT *
  FROM cooler_settings
  ORDER BY timestamp
  DESC LIMIT 1
"""

# Insert new row of cooler settings
SQL_INSERT_SETTINGS = """
 INSERT INTO cooler_settings (timestamp, setting, desiredTemperature)
 VALUES (NOW(), %s, %s)
"""
```

Figure 15: Example SQL Queries Used in Main Script

4.3.4.5 Database access from smartSwampCooler.py

Functionality is provided to read out the data from or write any data to the database, including retrieving the forecasted weather that is being updated every 5 minutes and updating all changes from the AUTO setting from the smart algorithm.

One example of reading from the database will be provided here to show that there are slight differences from main.py script supporting the website, this one to retrieve the most recent sensor reading:

```
def get_recent_data(sensor_name=""):
    sensor_data = SensorData()
    # creating a connection cursor
    mycursor = smartswampcooler_db.cursor()

    # get chosen sensor data based on id from database
    if sensor_name == "roof":
        params = (ROOF_SENSOR_ID,)
    elif sensor_name == "home":
        params = (HOME_SENSOR_ID,)
    else:
        print("ERROR: Invalid sensor name: {}".format(sensor_name))
        print("Specify valid sensor name: 'roof' or 'home'\n")
        return -1

    # retrieve most recent data
    mycursor.execute(SQL_SELECT_RECENT_SENSOR_DATA, params)
```

```
# this will extract row headers
row_headers = [x[0] for x in mycursor.description]
myresult = mycursor.fetchall()

# convert SQL entry to Python dict object
json_data = []
for result in myresult:
  json_data.append(dict(zip(row_headers,result)))

data = json.dumps(json_data, indent=4, sort_keys=True, default=str)
data = json.loads(data)

# print out all json data entries and map to SensorData object
for i in range(len(data)):
    #print(data[i])
    sensor_data = map_to_object(data[i])

return sensor_data
```

Figure 16: Retrieving most recent sensor data for smart algorithm

4.3.4.6 Gathering parameters for the Smart Algorithm

This script also reads current cooler settings and desired temperature set by the user on the website (using the get_cooler_setting() function), combines those with the forecasted data from the database and most recent sensor readings (using the get_forecast() and get_recent_data() functions, respectively), then feeds all of these inputs into a smart, predictive algorithm to determine the best cooler setting to achieve desired temperature (calling the libs.get_auto_setting() function calls the smart algorithm). It also incorporates house settings, including efficiency of the cooler, square footage of the house, volume of air produced by each cooler setting, latitude and longitude of the device, etc.

4.3.4.7 Smart Algorithm - libs/final_model.py

The smart algorithm uses a building block approach.

All of the possible cooler setting modes are checked and compared to the desired temperature.

Inside and outside temperatures and humidities are obtained, along with outside air temperatures and relative humidity forecasts. The exhaust temperature is estimated using the method in [2]. A difference equation approach is then used to estimate the change in the house temperature using heat flow as outlined in [3]. The simulation is then run and scored in each of the five operating modes to determine the most suitable mode of operation. The mode with the lowest score is chosen as the preferred mode. The 'Pump' mode is not considered in the simulation as it is the same as 'Off.' The most suitable mode is returned to the controller.

The cooler operates in one of the following modes:

'Off'
'Fan Hi'
'Fan Lo'
'Fan Hi (w/Pump)'
'Fan Lo (w/Pump)'
'Pump'

```
    # loop over modes
    best_mode = 'Off'
    best_score = 99999
    best_next_T = 0
    for mode in modes:
        if mode == 'Pump': break
        logger.info( 'Mode: {}, House: Temp {:0.1f}, Rh {:0.1f}, Ambient: Temp {}, Rh {}'.format(mode, T_house, rh_house, T_ambient, rh_ambient ) )
        result=forecast_inside_conditions(4,
                                  modes[mode]['fan'], modes[mode]['pump'],
                                  T_ambient, rh_ambient,
                                  T_house, rh_house, v_dot_air, house_vol
                                  )
        score = sum((result.T_house - desired_temp )**2)
        logger.debug( 'Score at time steps\n{}'.format ( (result.T_house - desired_temp )**2 ) )
        logger.debug( 'Temperature at time steps\n{}'.format ( result ) )
        logger.info( 'Final Score {}'.format(score) )

        if score < best_score:
            best_score = score
            best_mode = mode
            best_next_T = result.T_house[0]

    # score and return result
    logger.info('Next inside temperature: {}'.format(c2f(k2c( best_next_T ))))
    return best_mode
```

Figure 17: Determining Best Cooler Setting

4.3.4.7.1 Environment testing - libs/environment.py

This file makes a class representing a mass of air with temperature, relative humidity, and volume. It has a method that mixes two environments together and a method that removes a given volume of air.

```
8   class Environment():
9       def __init__(self, temp, rh, volume):
10          self.tem = temp
11          self.rh  = rh
12          self.vol = volume
13
14      def __repr__(self):
15          return "%s(%r)" % (self.__class__, self.__dict__)
16
17      def __print__(self):
18          return "%r" % (self.__dict__)
19
20      def mix(self, env):
21          tvol = self.vol + env.vol
22          self.rh  = (self.rh * self.vol + env.rh * env.vol) / tvol
23          self.tem = (self.tem* self.vol + env.tem* env.vol) / tvol
24          self.vol = tvol
25
26      def remove(self, vol):
27          self.vol -= vol
```

Figure 18: environment.py Class for Testing

4.3.4.7.2 Conversion factors from [7] - libs/cooler_model.py

The following functions were used to convert between various units of measurement for temperature:

```
16
17  def c2k(c):
18      ''' Celcius to Kelvin '''
19      return c+273
20
21  def k2c(k):
22      ''' Kelvin to Celcius '''
23      return k-273
24
25  def f2c(f):
26      ''' Fahrenheit to Celcius '''
27      return (f-32)*5/9
28
29  def c2f(c):
30      ''' Celsius to Fahrenheit '''
31      return (c)*9/5 + 32
32
```

Figure 19: Conversion Factors

4.3.4.7.3 Cooler output - libs/cooler_model.py

The estimated cooler exhaust temperature and relative humidity are obtained from calculations from [3]. The function *calculate_outlet_temperature()* takes ambient air temperature and relative humidity and cooler efficiency as inputs and returns exhaust temperature and relative humidity.

```
99  def calculate_outlet_temp(t_amb, rh, efficiency = 0.75):
100     ''' t_amb - ambient temperature celcius, rh - relative humidity %
101     efficiency = (t_amb - t_exh) / (t_amb - t_wet)
102     return calculated cooler output temperature
103     '''
104     ### abdel-faheed eq 2, solved for t_wet
105     t_wet = calculate_t_wet(t_amb, rh)
106     t_exh = t_amb - efficiency * (t_amb - t_wet)
107     rh_exh = U(c2k(t_exh), c2k(t_wet)) * 100
108     return(t_exh, rh_exh)
```

Figure 20: Calculating Exhaust Values

4.3.4.7.3 Scoring

An error score is assigned to each of the modes by taking the sum of the squared errors for each simulation time step compared to the target temperature. The mode with the lowest score is returned as the result.

4.3.5 Main script for Flask website - /app/main.py

This script manages the main website, including accessing MySQL database, controlling which cooler setting and desired temperature is chosen, displaying inside and outside sensor data, showing historical cooler settings and sensor readings, and providing live forecast conditions.

4.3.5.1 Flask User Interface

The Flask user interface was implemented using Python, HTML, CSS and JavaScript. The portions of software developed within Python provided the data that would be supplied to the Flask pages and the HTML and JavaScript provided the function and format of the actual pages. The index page, which was used as the main page in the kiosk application, provides the base of this user interface. Above the method name “index,” this line of code is used:

```
@app.route("/")
def index():
```

The ‘@app.route(“/”)’ indicates that whenever the main or index page is opened or accessed, the method “index” will be executed. In this method, other methods are called within the main Python script to populate the main web page. This data was compiled into a dictionary, which was then sent to the HTML page.

```
templateData = {
    'time_roof': recent_roof_data.timestamp,
    'temp_roof': recent_roof_data.temperature,
    'hum_roof': recent_roof_data.humidity,
    'time_home': recent_home_data.timestamp,
    'temp_home': recent_home_data.temperature,
    'hum_home': recent_home_data.humidity,
    'num_days': numDays,
    'cooler_setting': coolerSetting,
    'desired_temp': desiredTemperature,
    'temperatures': temperatures,
    'times': times,

    'upcomingWeekWeatherData': upcomingWeekWeatherData,
    'upcomingWeekLength': upcomingWeekLength,
    'currentCommonTime': currentCommonTime,
    'currentTemperature': currentTemperature,
    'currentIcon': currentIcon,
    'currentForecast': currentForecast,
    'currentDate': currentDate,
    'cityAndState': cityAndState,
    'ip_addr': ip_addr
}

return render_template('main/index.html', **templateData)
```

Figure 21: Variables Provided to index.html

After running this method, the page located at ‘main/index.html’ is rendered with the data from the template. In the index.html file, specific syntax is used to bring in the template data and to organize the file. In the following image, the syntax outlined in blue is used to organize the header section of the HTML file. It is similar to how normal HTML header sections are organized.

The section outline in orange is the syntax used to organize the main content of the HTML file. Once the main content of the file is added, it is closed by using the syntax ‘{% endblock %}’. The syntax outlined in green demonstrates how to use the template data. The value ‘ip_addr’ is found

within the template dictionary that is rendered to the HTML page. The brackets are also needed in order to reference the value from the template data.

```
{% extends "base.html" %}

{% block header %}

<script src="static/Chart.min.js"></script>
<meta http-equiv="refresh" content="300">

{% endblock %}

{% block content %}
  <div class="qr_code">
    <h2> Connect personal devices to cooler by first connecting </h2>
    <h2> devices to same Wifi network as cooler. </h2>
    <h2> Then, scan QR code below to access this page on your phone. </h2>
    <h2> Or type this IP address in any browser search bar: {{ ip_addr }} </h2>
    <img src="/qr_code" alt="Image Placeholder" width="20%">
  </div>
```

Figure 22: Passing variables into HTML pages

Following this format, the HTML page can be constructed using HTML tags and structure according to the developer's preference.

4.3.5.2 Retrieving Forecast Data

The forecast data plays a large role in the algorithm and kiosk user interface for the SSC. The smart algorithm is able to make predictions on what cooler setting should be used based partly on the upcoming forecast. The forecast data is provided through weather.gov's API. This is one reason the system requires the use of Wi-Fi.

The API call is all done within the Flask server in the main Python script. To gain the correct forecast data from weather.gov, it needs to provide latitude

and longitude values of the SSC's location. In the main Python script for the website, the geocoder library is used for this. This library needs to be installed on the RPi and needs to be imported into the main script at the top of the file. The following code will produce a geocoder object using the IP of the device running the command. The string value 'me' indicates the IP address of the RPi.

```
g = geocoder.ip('me')
```

Once this value is retrieved, the command g.latlng will produce the latitude and longitude values of the raspberry pi. This value is supplied to the method used to retrieve the upcoming forecast data.

```
forecast_data = retrieve_raw_forecast_data(g.latlng)
```

Another library that is used in the main Python script to retrieve the forecast data is the "requests" library. It is installed in similar fashion to the geocoder library explained earlier. Following the steps used to import the geocoder library should also allow the requests library to be imported to the Python script. The requests library is used to perform the API query to retrieve the forecast data from weather.gov.

To achieve this, a session needs to be created and updated with a new header. The lines of code following this paragraph are used to perform that operation. When updating the header, the member 'User-Agent' needs to include the title of the project and an email address. This is used to identify the application and to make it less likely that there will be a security risk to the application. The string can be anything, but the more unique it is, the greater security there will be. weather.gov will keep the email confidential and will only use it to notify you if there is a security risk. Again, the following code gives an example of creating a session and updating the header:

```
session = requests.Session()
session.headers.update({'User-Agent': '(Smart Swamp Coooler Project, myEmail@gmail.com)'})
```

The next step to retrieve the data consists of providing a request string to weather.gov and then acquiring the response from the API call. The coordinates from the geocoder are used in the request to weather.gov. The variable entitled “coordinates” in the code used below holds both the latitude and longitude and are extracted by creating two new variables called “lat” and “lon.” They are then placed into the request string, which is later used to get the response from the API. Once the response is retrieved, a forecast request is extracted from the response and used to perform another request for the forecast data. This request can be found by referencing the ‘forecastHourly’ field from the ‘properties’ table. The results are shown in the following code snippet:

```
lat, lon = coordinates

request_string = 'https://api.weather.gov/points/{lat:0.3f},{lon:0.3f}'.format(lat=lat, lon=lon)
print('get '+request_string)

for i in range(0,5):
    response = session.get(request_string, timeout=(5, 60)) # wait 5 seconds
    if not response.ok:
        continue

    else:
        points_data = json.loads(response.content)
        forecast_request = points_data['properties']['forecastHourly']
        break

for i in range(0,5):
    response=session.get(forecast_request, timeout=(5, 60)) # wait 5 seconds
    if not response.ok:
        continue
    else:
        return json.loads(response.content)
```

Figure 23: Requesting Forecast from weather.gov

The return statement consisting of ‘json.loads(response.content)’ holds the data for the forecast data. This can then be formatted and extracted at the developer’s preference. For the SSC, the forecast displays as follows:

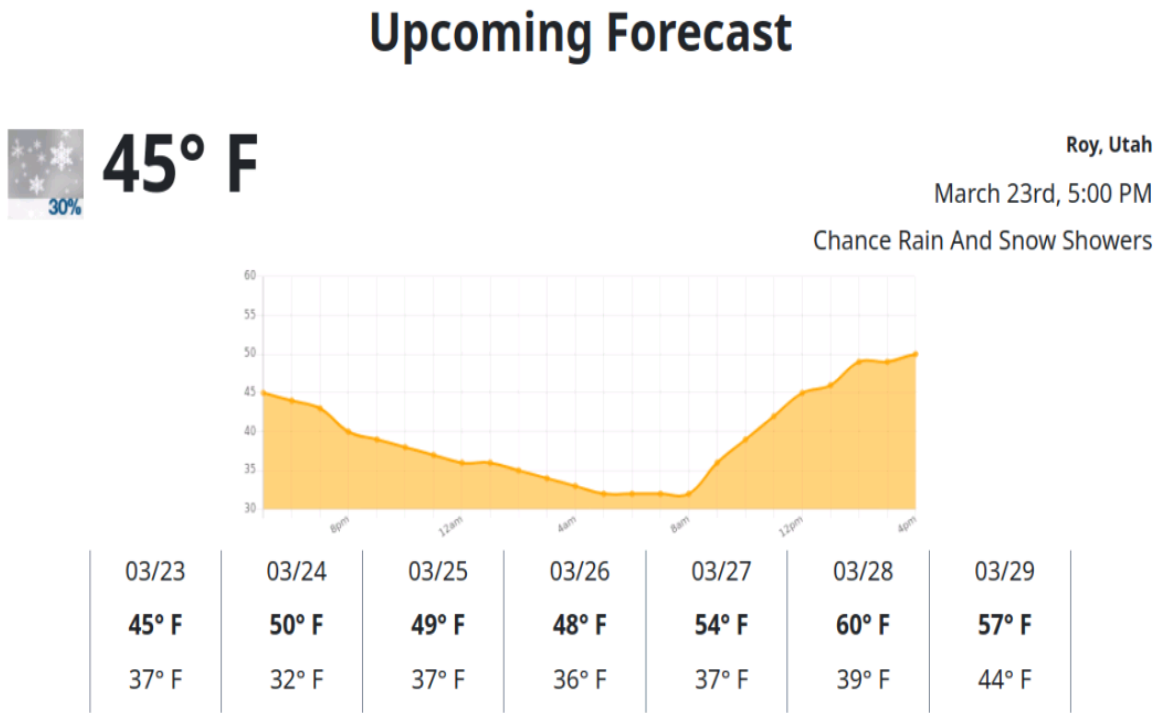

Figure 24: Forecast Displayed on Website

4.3.5.3 Connecting to the MySQL database

Connecting to the database is done as follows (similar to section 4.3.4.2):

```
# Needs to be used in every flask application
app = Flask(__name__)

# Connect to the mySQL database and provide login credentials
app.config['MYSQL_HOST'] = 'localhost'
app.config['MYSQL_USER'] = 'pi'
app.config['MYSQL_DATABASE'] = 'smartswampcooler'
app.config['MYSQL_PASSWORD'] = 'jeniffer'
mysql = MySQL(app)
```

Figure 25: Connecting to MySQL Database from Flask Script

4.3.5.4 Database access from Flask

Functionality is provided to read out the data from or write any data to the

database, including updating the database with the forecasted weather every 5 minutes and updating all changes from the user in the cooler operation and desired temperature.

Data is pulled from the database to display most recent indoor and outdoor sensor readings, historical temperature and humidity changes for both sensors, and changes in the cooler settings over time. SQL queries are similar to the section described for the smartSwampCooler.py script.

One example of reading from the database will be provided here to show that there are slight differences from smartSwampCooler.py script, this one also to retrieve the most recent sensor reading:

```
def getRecentData(sensor_name=""):
  # create a connection cursor to the database
  mycursor = mysql.new_cursor()

  # get chosen sensor data based on id from database
  if sensor_name == "roof":
    params = (ROOF_SENSOR_ID,)
  elif sensor_name == "home":
    params = (HOME_SENSOR_ID,)
  else:
    print("ERROR: Invalid sensor name: {}".format(sensor_name))
    print("Specify valid sensor name: 'roof' or 'home'\n")
    return -1

  # retrieve most recent data
  mycursor.execute(SQL_SELECT_RECENT_DATA, params)
```

```
# this will extract row headers
row_headers = [x[0] for x in mycursor.description]
myresult = mycursor.fetchall()

# convert SQL entry to Python dict object
json_data = []
for result in myresult:
  json_data.append(dict(zip(row_headers,result)))

data = json.dumps(json_data, indent=4, sort_keys=True, default=str)
data = json.loads(data)

# print out all json data entries and map to SensorData class object
for i in range(len(data)):
    #print(data[i])
    sensor_data = map_to_object(data[i])

return sensor_data
```

Figure 26: Retrieving most recent sensor data for website

4.3.5.5 Displaying most recent sensor data

The data retrieved in the previous section is displayed on the website:

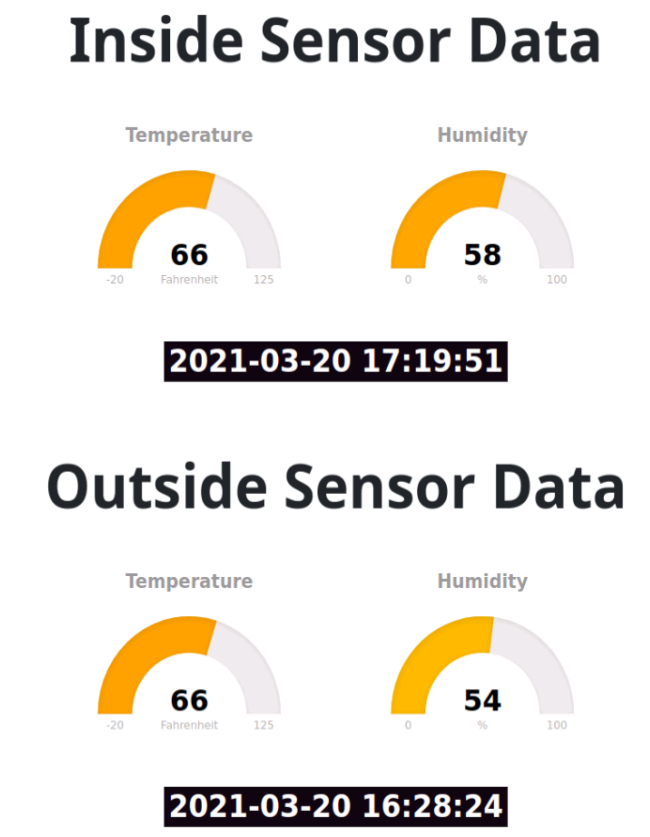

Figure 27: Recent Sensor Data on Website

The index.html file loads the values into these Indoor and Outside Sensor Data sections:

```
<div class="sensor_data">
  <h1> Inside Sensor Data </h1>
  <div id="g3"></div>
  <div id="g4"></div>
  <br>
  <span><a href="/"class="button">{{ time_roof }}</a></span>
</div>
<div class="sensor_data">
  <h1> Outside Sensor Data </h1>
  <div id="g1"></div>
  <div id="g2"></div>
  <br>
  <span><a href="/"class="button">{{ time_home }}</a></span>
</div>
  <hr>
```

Figure 28: Loading Sensor Data into index.html

The temperature and humidity sensor gages are created using the justgage.js JavaScript file [4]. This generates the images for the gages, updates the values, and changes the color of the gages depending on where the value falls on the scale for temperature or humidity. The actual code to display the gages on the main website is shown below.

```
<script src="../static/raphael-2.1.4.min.js"></script>
<script src="../static/justgage.js"></script>
<script>
var g1, g2, g3, g4;
document.addEventListener("DOMContentLoaded", function(event) {
    g1 = new JustGage({
    id: "g1",
    value: {{temp_roof}},
    valueFontColor: "black",
    min: -20,
    max: 125,
    title: "Temperature",
    label: "Fahrenheit"
});
,,,
    g2 = new JustGage({
    id: "g2",
    value: {{hum_roof}},
    valueFontColor: "black",
    min: 0,
    max: 100,
    title: "Humidity",
    label: "%"
});
    g3 = new JustGage({
    id: "g3",
    value: {{temp_home}},
    valueFontColor: "black",
    min: -20,
    max: 125,
    title: "Temperature",
    label: "Fahrenheit"
});
```

```
        g4 = new JustGage({
        id: "g4",
        value: {{hum_home}},
        valueFontColor: "black",
        min: 0,
        max: 100,
        title: "Humidity",
        label: "%"
    });
      });
  </script>
```

Figure 29: Displaying Sensor Gages on Website

4.3.5.6 Displaying cooler settings and desired temperature

The current cooler setting is pulled from and stored in the MySQL database, then passed into the index.html file and loaded as described in the above sections. To change a cooler setting, a dropdown menu is provided for all available settings:

1. Off
2. Auto
3. Fan Hi
4. Fan Lo
5. Fan Hi (w/Pump)
6. Fan Lo (w/Pump)
7. Pump

A slider is added below the menu to select the desired temperature for the AUTO setting. When the submit button is pressed, the pair of values {cooler_setting, desired_temperature} are written to the database and passed to smartSwampCooler.py script. The dropdown menu is constructed as shown below, followed by an image of the actual section of the website as seen by the user.

```
<div class="cooler_settings">
  <br>
  <h3>Current Cooler Setting: <span class="currentSelection">{{ cooler_setting }}</span></h3>
  <h3>Change Cooler Setting:</h3>
      <form method="POST">
      <label for="coolerSetting"></label>
      <select id="coolerSetting" name="coolerSetting" style="font-size:50px;">
          <option value="Off" style="font-size:50px;">Off</option>
          <option value="Auto" style="font-size:50px;">Auto</option>
          <option value="Fan Hi" style="font-size:50px;">Fan Hi</option>
          <option value="Fan Lo" style="font-size:50px;">Fan Lo</option>
          <option value="Fan Hi (w/Pump)" style="font-size:50px;">Fan Hi (w/Pump)</option>
          <option value="Fan Lo (w/Pump)" style="font-size:50px;">Fan Lo (w/Pump)</option>
          <option value="Pump" style="font-size:50px;">Pump</option>
      </select>
      <input type="submit" style="font-size:50px;"/>
      <div class="desiredTemperature">
          <h3>Set Auto Temperature (°F):</h3>
          <input id="desiredTemp" name="desiredTemperature" type="range" min="55" max="95" step="1"
                 oninput="GetDesiredTemperatureValue(this.value);">
          <input type="text" id="desiredTemperatureValue" value="{{ desired_temp }}" readonly>
      </div>
      </form>
```

Figure 30: Configuring dropdown menu for cooler settings

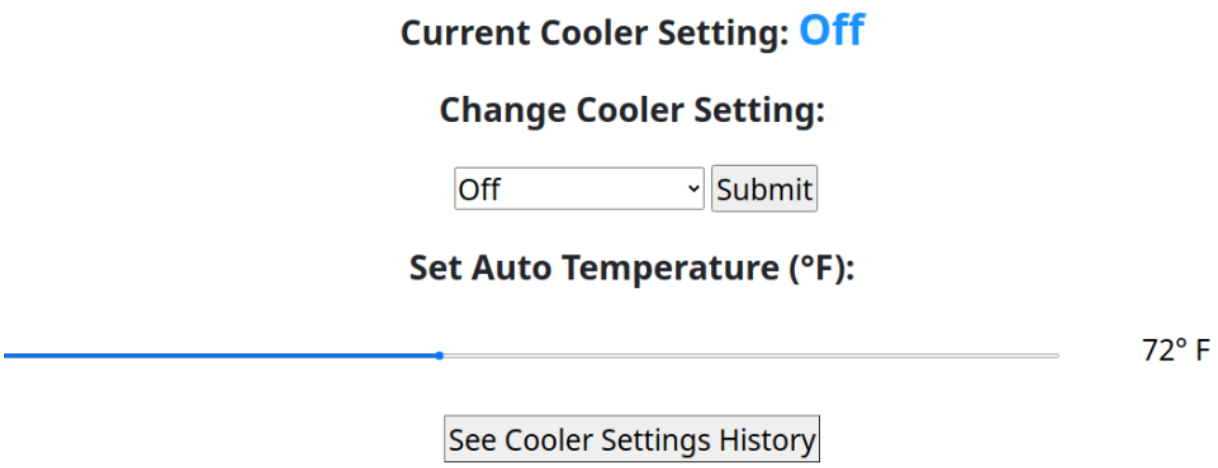

Figure 31: Displaying cooler settings and temperature slider on website

4.3.5.7 Displaying historical temperature and humidity data

Historical data from the wireless sensors is stored in the database and retrieved in the main.py script as shown below. A function is called to plot one of the following: 1. Home Temperature, 2. Home Humidity, 3. Roof

Temperature, 4. Roof Humidity. The plot is generated using matplotlib (also shown below). The Python plot is not compatible with HTML, so the lines beginning with the comment "# convert to image that can be displayed in html" is needed to convert the Python plot.

```
# route for index page to plot home temperature plot
@app.route('/plot/temp_home')
def plot_temp_home():
    # need global number of days to retrieve number of days in this function
    global numDays

    # retrieve home temperature data for given number of days from database
    times, temps, hums = read_sensor_db(sensor_name="home", days=numDays)
    return plot_sensor(temps, loc='inside', sensor='temperature', time=times)
```

Figure 32: Plotting Historical Data

```
# plot temperature and humidity plots on website, return png
# image to display
def plot_sensor(data, time, loc='inside', sensor='temperature'):
    # assign y axis for temperature or humidity
    ys = data
    #print("These are my {} {} values:\n{}".format(loc, sensor, ys))

    # create new subplot
    fig = Figure()
    axis = fig.add_subplot(1, 1, 1)

    # assign y axis label depending on temp or hum
    if sensor == 'temperature':
        axis.set_title("Temperature [°F]")
    else:
        axis.set_title("Humidity [%]")
```

```
    # add label, grid, x-axis spacing, other formatting
    axis.set_xlabel("Time")
    axis.grid(True)
    xs = time
    axis.plot(xs, ys)
    axis.tick_params('x', labelrotation=75)
    fig.set_tight_layout(True)

    # convert to image that can be displayed in HTML
    canvas = FigureCanvas(fig)
    output = io.BytesIO()
    canvas.print_png(output)

    # return png image to be displayed
    response = make_response(output.getvalue())
    response.mimetype = 'image/png'

    return response
```

Figure 33: Plotting and Converting to HTML-Viewable Image

The image is passed into the index.html file and one of four available settings for the number of past days of readings can be displayed: 1, 7, 14, or 31.

```
<div class="log_data">
 <br>
 <h3> Showing last <span class="currentSelection">{{ num_days }}</span>
      day(s) of temperature/humidity readings </h3>
 <h3> Change # of Days: </h3>
     <form method="POST">
     <label for="numDays"></label>
     <select id="numDays" name="numDays" style="font-size:50px;">
         <option value="1" style="font-size:50px;">1</option>
         <option value="7" style="font-size:50px;">7</option>
         <option value="14" style="font-size:50px;">14</option>
         <option value="31" style="font-size:50px;">31</option>
     </select>
     <input type="submit" style="font-size:50px;">
     </form>
```

```
<br>
<br>
<hr>
<h3> HISTORICAL OUTSIDE DATA </h3>
<hr>
<img src="/plot/temp_roof" alt="Image Placeholder" width="49%">
<img src="/plot/hum_roof" alt="Image Placeholder" width="49%">
<h4> Time given as YYYY-MM-DD or MM-DD Military Time for 1-day plot </h4>
<hr>
<h3> HISTORICAL INSIDE DATA </h3>
<hr>
<img src="/plot/temp_home" alt="Image Placeholder" width="49%">
<img src="/plot/hum_home" alt="Image Placeholder" width="49%">
<h4> Time given as YYYY-MM-DD or MM-DD 24-hour clock for 1-day plot </h4>
</div>
```

Figure 34: Displaying Historical Temperature Data in index.html

The resulting plot that displays on the website looks like the following:

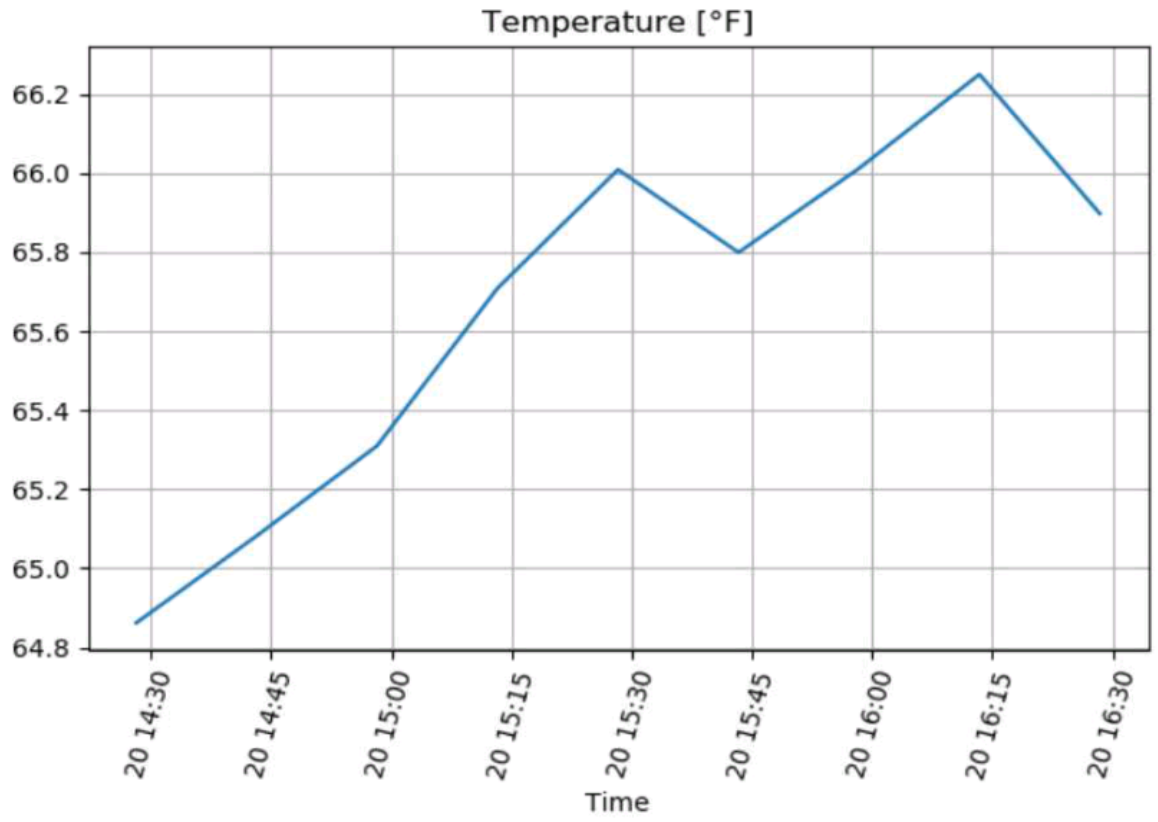

Figure 35: Displaying Historical Temperature Data on Website

4.3.5.8 cooler_settings_log.html

This HTML page displays a table of timestamps, cooler settings, and desired temperature, one row for each time a new cooler setting has been added or changed by the user or smart algorithm. Functionality to and from this page were added as simple navigation buttons on the index.html file.

```
# route to display historical cooler settings table
# displays rows of timestamps with each new cooler setting and desired temperature
@app.route('/cooler_settings_log')
def cooler_settings_log():
    # creating a connection cursor
    mycursor = mysql.new_cursor()

    # get all historical cooler settings to display
    mycursor.execute(SQL_SELECT_ALL_COOLER_SETTINGS)
    data = mycursor.fetchall()

    # render cooler settings table page
    return render_template('main/cooler_settings_log.html', value=data)
```

Figure 36: Calling the cooler_settings_log.html template

The table is created using HTML's <table> heading. Date and Time, Cooler Setting, and Desired Temperature are added as <th> label headings. The value variable shown below takes the timestamp, cooler setting, and desired temperature values from the database and feeds them into the body of the table using the <tr> tags. This HTML file is shown below.

```
<html>
<body>
<section id="cooler_settings_log">
  <form>
    <button formaction="/" style="font-size:45px;">BACK TO HOME</button>
  </form>
  <br>
  <font size="20" face="Courier New" >
  <table class="cooler_settings_table" cellspacing="50px">
    <thead>
    <tr>
    <th>Date and Time</th>
    <th>Cooler Setting</th>
    <th>Desired Temperature</th>
    </tr>
    </thead>
```

```
    <tbody>
    {% for row in value %}
        <tr>
            <td>{{row[1]}}</td>
            <td>{{row[2]}}</td>
            <td>{{row[3]}}</td>
        </tr>
    {% endfor %}
    </tbody>
  </table>
  </font>
  <br>
  <form>
    <button formaction="/" style="font-size:45px;">BACK TO HOME</button>
  </form>
</section>
</body>
</html>
```

Figure 37: Creating Table of Historical Settings

4.3.5.8 Finding RPi's IP and adding QR code to website

To retrieve the IP address of the RPi, a simple egrep function is run on the output of the ifconfig wlan0 terminal command as shown below. "http://" is appended to the found IP address to make the link clickable across all devices when converted to a QR code. Python has a built-in qrcode library that converts text to QR codes. Similar to the historical data, the QR code is converted to an HTML-compatible image and inserted into the website. Below the following code snippet, the QR code on the website is shown.

```
# return Pi's current IP address or IP address as QR code
def ip_or_qr(get=None):
    # find current IP address where website is available on network
    stream = os.popen("ifconfig wlan0 | egrep -o 'inet ([0-9]{1,3}\.){3}[0-9]{1,3}'
                      | egrep -o '([0-9]{1,3}\.){3}[0-9]{1,3}'")

    # append "http://" to make QR code link to website immediately across all devices
    ip_addr = "http://" + stream.read()

    # return IP address if specified
    # print("Current IP address: ", ip_addr)
    if get == 'ip':
        return ip_addr

    # make and return QR code image if specified
    if get == 'qr':
        # convert IP address to QR code to display on website,
        # where any phone can scan and connect to it
        return qrcode.make(ip_addr)
```

Figure 38: Retrieving IP Addressing and Generating QR Code

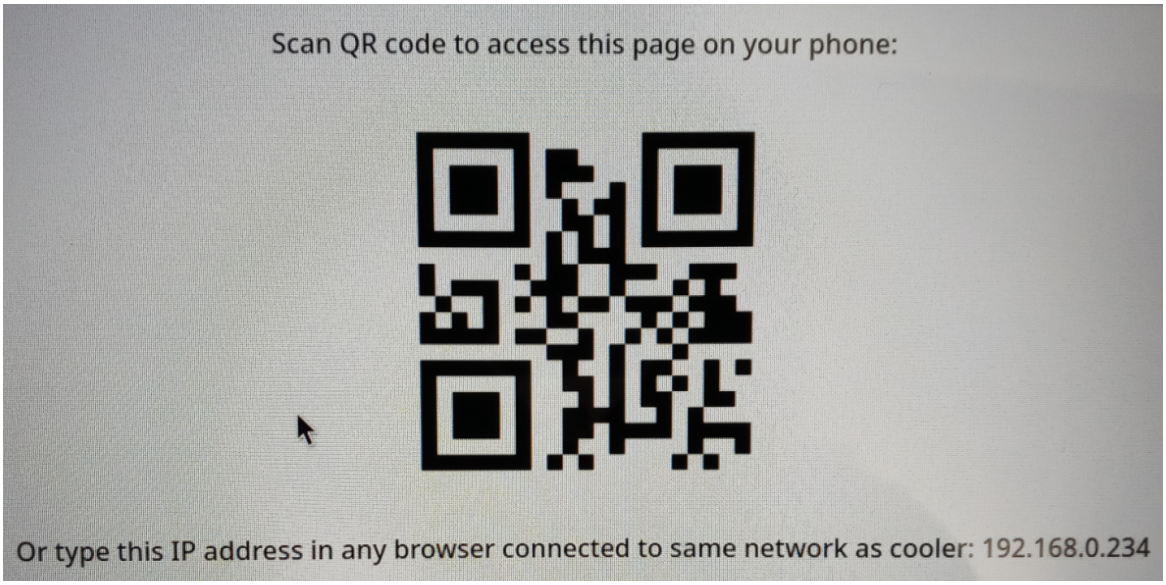

Figure 39: QR Code and IP Address Displayed on Website

4.3.5.9 Disabling website caching

When the RPi connects to a new network or reloads its main web page, it has a tendency to load images and data from cache. This resulted in issues with the QR code showing the incorrect IP address or any of the recent sensor readings, forecast image, or historical logs showing out-of-date data. The following code was added to the end of the script to prevent any of this caching:

```python
# disable caching so automatic refresh gathers most up-to-date values for
# QR code, IP address, sensor readings, and forecasted data
@app.after_request
def set_response_headers(response):
    response.headers['Cache-Control'] = 'no-cache, no-store, must-revalidate'
    response.headers['Pragma'] = 'no-cache'
    # make caching disabled immediately
    response.headers['Expires'] = '0'
    response.cache_control.max_age = 0
    return response
```

Figure 40: Disabling Website Caching

4.3.6 Chromium full-screen Kiosk Mode

The RPi is set to boot into full-screen Kiosk Mode immediately after being hooked up. The following line of code makes this possible:

chromium-browser --kiosk http://127.0.0.1

The website is set up to run on localhost port 80, so loading Kiosk mode is as simple as starting chromium-browser with the --kiosk flag on localhost.

The next section will detail how to get this and other scripts to run when the RPi powers up.

4.3.7 systemd.service running main scripts on boot

systemd services can be configured to run programs without any user input. To create a new service (in this case, the service to run the smartSwampCooler.py service on boot), the following command line is used:

sudo systemctl --force --full edit SSC.service

To simply modify an existing service, the --force flag is omitted. An empty editor will pop up, and the following statements are inserted:

```
[Unit]
Description=smartSwampCooler data collection
After=multi-user.target

[Service]
User=pi
WorkingDirectory=/home/pi/webapp
ExecStart=/usr/bin/python3 /home/pi/webapp/smartSwampCooler.py
Restart=on-failure
Environment=PYTHONUNBUFFERED=1

[Install]
WantedBy=multi-user.target
```

Figure 41: systemd.service for smartSwampCooler.py script on boot

Once the statements have been saved and the editor closed, the service is enabled and the system rebooted:

sudo *systemctl enable SSC.service*

sudo systemctl reboot

The status of the service can be checked with:

systemctl status SSC.service

To view text output from the script running, the output can be found in the systemd journal:

journalctl --boot --unit=SSC.service

The script to configure the Wi-Fi network and launch the website in Chromium's Kiosk Mode is as follows:

```
#!/bin/bash
sleep 5

x=`ifconfig wlan0 | grep "inet "`
while [ "$x" = "" ]
do
    echo Connecting
    matchbox-keyboard
    x=`ifconfig wlan0 | grep "inet "`
done

chromium-browser --kiosk http://127.0.0.1
```

Figure 42: Bash script for Wi-Fi and Chromium Kiosk Mode on boot

After a 5 second delay to allow the Raspbian Lite desktop manager to properly load, grep is used to determine if the SSC is connected to Wi-Fi. If it is not connected to a network, the SSC will boot into the Raspbian Lite desktop. A virtual keyboard will pop up and the only icon available on the top taskbar will be the Wi-Fi configuration tool. The user is directed in the Installation Manual/Quick Start Guide to select their desired Wi-Fi network, enter their password, and click connect on the Wi-Fi configuration GUI.

Once the Wi-Fi icon shows the unit is connected to Wi-Fi, the user closes the virtual keyboard, and the website launches automatically in full-screen Chromium Kiosk Mode.

With slight variations from the SSC.service detailed above, the same process is followed to use systemd.service to run this kiosk_wifi_setup.sh script:

```
[Unit]
Description=Chromium Kiosk
Wants=graphical.target
After=graphical.target

[Service]
Environment=DISPLAY=:0.0
Environment=XAUTHORITY=/home/pi/.Xauthority
Type=idle
ExecStart=/bin/bash /home/pi/webapp/kiosk_wifi_setup.sh
Restart=on-abort
User=pi
Group=pi

[Install]
WantedBy=graphical.target
```

Figure 43: systemd.service for kiosk and Wi-Fi setup script on boot

5.0 Testing

The smart algorithm underwent an intensive series of unit tests, ranging from extreme conditions to the most common, and the algorithm predicted the most efficient setting with no failures.

The mechanical relays and cooler settings control were extensively tested to turn on correct settings from the RPi's GPIO signals, verified by connecting to a portable swamp cooler and confirming correct fan and pump settings turn on/off.

End-to-end communication was verified modularly, from checking wireless Zigbee modules processing and sending data to the Coordinator Zigbee, to forwarding data serially to the RPi, to displaying temperature and humidity readings and their variation over time on the kiosk and website. Wireless modules were tested for weeks at a time to verify expected battery life and power consumption. Sensor data entries to the database were checked against house thermometers and local weather data, producing almost identical readings.

Each setting and button on the kiosk and website was checked and clicked hundreds of times, each achieving expected results.

This section will detail how each requirement from section 3.1 is met and/or tested.

5.1 Testing the 5 VDC Power Supply - Reqs 3, 6

The external 120 VAC power supply was confirmed to be dropped to the needed 5 VDC for the RPi when checked with a handheld multimeter. A load was then prepared on a breadboard and again checked with a handheld multimeter. Finally, the power supply was attached to the RPi SBC and correct operation was verified.

5.2 Testing the Touchscreen - Reqs 5, 6

The 1024 x 600 Touch Screen IPS Display was attached to the Raspberry Pi SBC using manufacturer supplied cables. The Raspberry Pi SBC was then used to validate the correct operation of the touchscreen. All touchscreen functionality worked as intended and no further testing was required.

5.3 Testing Smart Algorithm - Reqs 1, 4, 12, 13

To test and develop the smart algorithm, __main__ functions were added to the bottom of the applicable Python script which called the algorithm functions with test data and printed the results to the console for verification.

The function *calculate_cooler_efficiency()* found in the *libs/cooler_model.py* file provides the mathematical model of a swamp cooler in various ambient conditions. Its output was compared to an industry standard swamp cooler performance chart. The function *get_auto_setting()* in the main *smartSwampCooler.py* script was tested by providing various operational parameters and ensuring the correct mode was returned. The main predictive function called *forecast_inside_conditions()* in *libs/final_model.py* was tested by printing the result of several test trials to ensure the trend and end results meet expectations.

The *Environment* class found in the *libs/Environment.py* file was tested by mixing 1000 ft^3 air at 70°F into a 10,000 ft^3 container with air at 80°F over 20 time steps. The results are presented in the figure below.

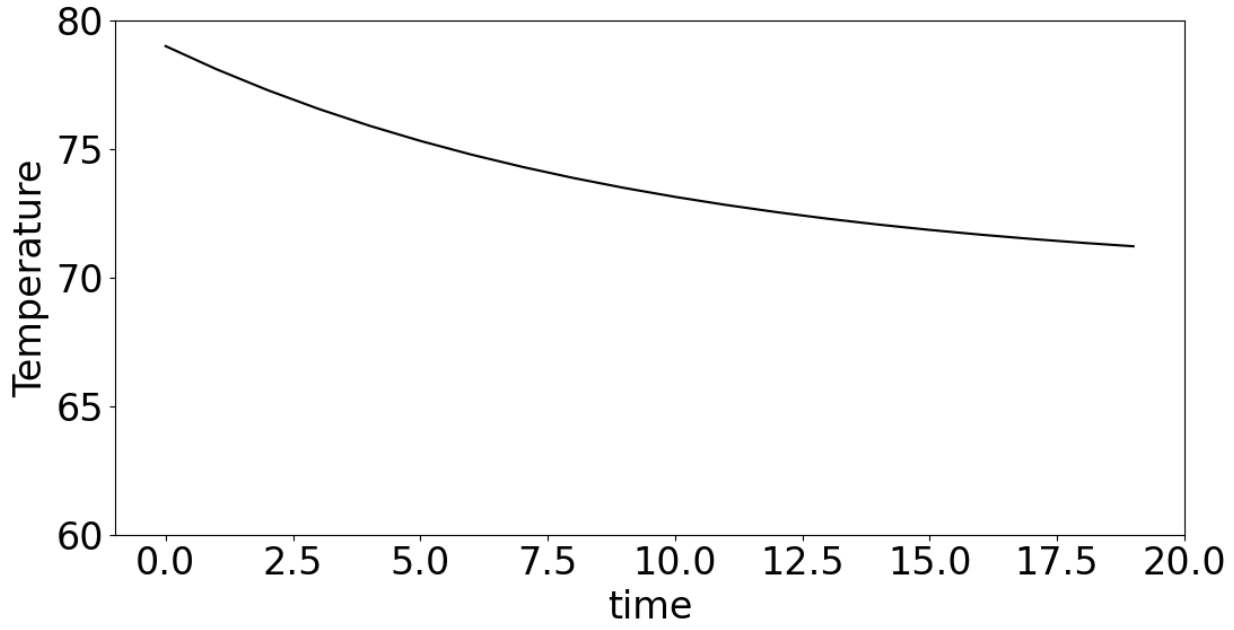

Figure 44: Environmental Class Test results

The swamp cooler model output model was run at several different temperature and relative humidity conditions and the results compared to a commonly accepted temperature chart. The SSC model is comparable to the following reference chart when a 0.744 efficiency was used.

	2	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80
75	54	55	57	58	59	60	61	63	64	65	66	67	68	69	70	71	72
80	58	58	60	61	63	64	65	67	68	69	70	71	72	73	74	75	76
85	61	62	63	65	66	68	69	71	72	73	74	76	77	78	79	80	81
90	64	65	67	68	70	72	73	75	76	77	79	80	81	83	84	85	86
95	67	68	70	72	74	75	77	79	80	82	83	85	86	87	88	90	91
100	70	71	73	75	77	79	81	83	84	86	88	89	91	92	93	94	95
105	73	74	76	79	81	83	85	87	89	90	92	94	95	97	98	99	100
110	76	77	80	82	84	87	89	91	93	95	96	98	100	101	103	104	105
115	78	80	83	85	88	90	93	95	97	99	101	103	104	106	108	109	110
120	81	83	86	89	92	94	97	99	101	103	105	107	109	111	112	114	115
125	84	86	89	92	95	98	101	103	106	108	110	112	114	116	117	119	120

Figure 45: Attaining optimum conditions for SSC
Source: Adapted from [2]

The algorithm was tested to obtain a swamp cooler setting by choosing 4 environmental conditions and comparing the received to expected results. The results of one of these unit tests is shown below:

Desired Temp	Outside Temp	Inside Temp	Expected	Result
75°	90°	90°	Fan w/Pump	Fan Low/Pump
75°	90°	60°	Fan	Fan Hi
75°	60°	90°	Fan w/Pump	Fan Lo
75°	60°	60°	Off	Off

Figure 46: Example unit test for smart algorithm

5.4 Testing MC PCB Functionality - Reqs 1, 2, 6, 7

After initial assembly of the main board's mechanical relays and 5 V power supply, relay operation was validated by jumpering 5 VDC to the correct GPIO relay driver pins and verifying the relays open and closed properly. After validating all main board functionality, the main board was connected to a Raspberry Pi SBC and a Python script was executed to ensure the Raspberry Pi SBC could control the relays via GPIO pins. Many scripts were used for testing this functionality and can be viewed in the Previous_Test/ folder in the Code Listing. One simple script run to check MC PCB functionality is shown below.

```
#!/usr/bin/python3.7

from gpiozero import LED
from time import sleep

speed = LED(21)
fan   = LED(20)
pump  = LED(16)
delay = 20

done = False

while not done:
    print('Low fan test starting...')
    sleep(delay/10)
    print('Turn on fan')
    fan.on()
    sleep(delay)
    print('Hi fan test ...')
    speed.on()
    sleep(delay)
    print('Low fan test ...')
    speed.off()
    sleep(delay)
    print('fans off')
    fan.off()
    done=True
```

Figure 47: Script to test MC PCB functionality

5.5 Testing Zigbee 3 modules - Reqs 1, 8, 13

The range for each Zigbee was tested over 3 days, incrementing from 1 ft up to 50 ft indoors (~3 ft increments). The MC should never be further away from any of the wireless sensors than 50 ft, so range testing was not tested

outside of this range.

In these tests, all sensor data packets were properly received, including through 4 walls for the final test. All results were verified by checking the MySQL database records during the testing periods. Outdoor tests were also conducted over 2 days and verified up to 200 ft.

Zigbee deep sleep functionality was also tested incrementally, starting at 1 minute and eventually configuring the final model to wake up and sample every 15 minutes. Packets were properly received at every deep sleep interval, verified by connecting the Zigbee transmitting module and Zigbee receiving module to Python REPL consoles to a common terminal and comparing send/receive messages side-by-side. The next two sections will provide example images of this testing strategy.

Furthermore, XBee 3 specifications were also verified with a multimeter, and the results are shown below.

	Zigbee
IEEE Spec	IEEE 802.15.4
Type of Module	XBee Series 3
Sleeping Mode	2 µA
Awake Mode	40 mA
Transmitting Mode	40 mA
Receiving Mode	17 mA
Power Supply	2.1 - 3.6 V

Figure 48: Verifying Zigbee Specifications

5.6 Testing Serial Communication - Reqs 7, 8

The RPi samples its serial UART pins once every second for ten seconds, then sleeps for five seconds. The RPi serial settings had to be matched to those of the Zigbee serial protocol. The baud rate was adjusted to 57600 to get consistent, clear results from the Coordinator Zigbee.

```
# Setup for Serial port to match Zigbee protocol
ser = serial.Serial(
    port = '/dev/ttyS0',
    baudrate = 57600,
    parity = serial.PARITY_NONE,
    stopbits = serial.STOPBITS_ONE,
    bytesize = serial.EIGHTBITS,
    timeout = 1
)
```

Figure 49: RPi serial settings to match Zigbee protocol

The connections shown below were also tested with an oscilloscope to verify the RX/TX pins on the RPi matched the TX/RX pins on the Zigbee Coordinator.

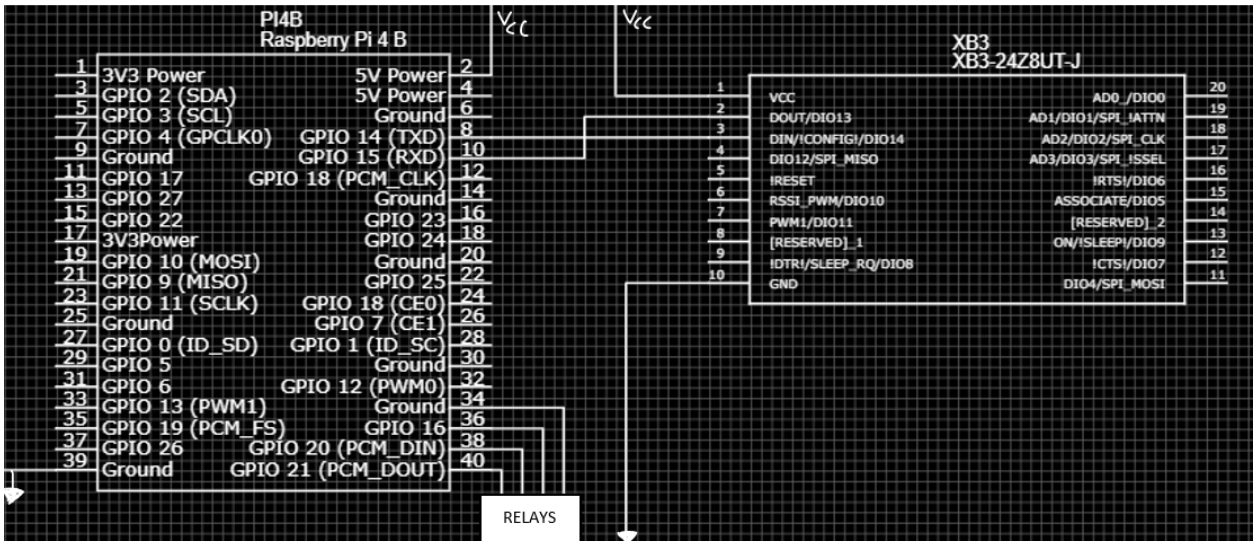

Figure 50: UART connections tested from Zigbee to RPi

Results from the Zigbee Coordinator were printed out from the smartSwampCooler.py script as shown below. The data was compared side-by-side with a wireless Zigbee Router connected to a Micropython REPL console. Results were shown to be 100% consistent in all tests conducted.

Figure 51: smartSwampCooler.py Zigbee sensor data on console

5.7 Testing Sensor Communication - Req 7, 8

Sensor data was verified as described above, with the Zigbee coordinator on the left receiving matching data to the Zigbee Router sensor data shown on the right in the below figure. Both Zigbee modules were connected via FTDI connectors to a central laptop to display two side-by-side PyCharm Micropython REPL consoles.

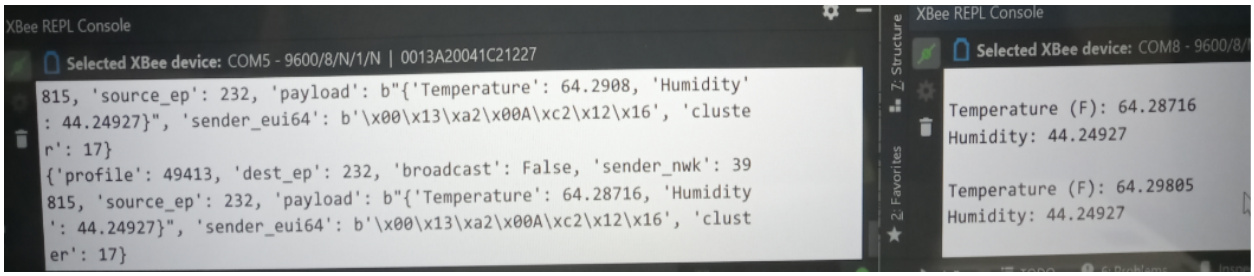

Figure 52: Zigbee Coordinator receiving data from Router

5.8 Testing the 3.3 VDC Power Supply - Req 8

The buck converter power supply was confirmed to be 3.3 VDC when checked with a handheld multimeter. This test was repeated with a single 1.5 V AA battery and two AA batteries. A load was then prepared on a breadboard and again checked with a handheld multimeter.

5.9 Testing UI using Flask under Apache2 - Req 9

The main script running the Flask website was added as a module under the Apache2 web server. This allowed the website to boot automatically on localhost port 80. localhost was searched in the Chromium browser on the RPi and the website was confirmed to boot properly. The full-screen Chromium kiosk mode was also confirmed to use the localhost address to load the website when the RPi boots.

5.10 Testing web service over LAN - Req 10

The website was tested on over two dozen devices and many different manufacturers. 4 tablets were tested, 8 laptops, and all brands of cell phones were able to either type in the IP address from the website or scan the QR code to access the website, control the swamp cooler, and manage and view all functionality available on the website. CSS was modified extensively to format the website for devices of all sizes, including laptops and mobile.

5.11 Testing kiosk mode on boot - Req 11

As described in section 5.9, the full-screen Chromium kiosk mode loads properly when the RPi powers on. The kiosk_and_wifi_setup.sh script was tested dozens of times to verify that the kiosk mode loads as soon as Wi-Fi connectivity is established. The accompanying systemd.service running the

kiosk and Wi-Fi setup script was also tested and confirmed to work dozens of times.

5.12 Testing the Forecast Data Retrieval - Reqs 12, 13

Simple tests were done to verify the function of retrieving the forecast data from weather.gov. Pulling the proper and accurate data was verified with print statements in Python. When a successful response was achieved, print statements were used to show the data from the API call.

Further tests were done while implementing the final design by plotting the forecast on the main website and comparing it to local forecast data from two other sources. When the data was not available or a bad response was received after performing the request, the data on the kiosk application would either not display or an error would display on the web page.

This prompted incremental changes to identify and correct the errors that occurred. Near the final design testing, it was found that the requests were displaying a bad request error as mentioned above. The main Python script was adjusted to perform multiple requests for the forecast data instead of just one. For the final design, the request limit was bumped up to five times if a valid response was not given.

5.13 Testing MySQL database connection - Req 14

Database SQL queries to pull from and add to the MySQL database were each tested extensively. Each query was verified by first running the query on the PHPMyAdmin website tool, then running it within the desired Python script. Results from each query were readily available on the database and were verified individually to produce desired results.

5.14 Testing MC PCB 3D enclosure - Req 15

The 3D fabricated enclosure was connected to the MC PCB and mounted as desired. All SSC functionality was again tested and verified to be accurate. All components within the enclosure were secured with screws.

5.15 Testing wireless node 3D enclosure - Req 16

The 3D fabricated enclosures for the wireless nodes were connected as specified. All SSC functionality was again tested and verified to be accurate. All components within the nodes were secured with screws.

5.16 Testing external sensor solar panel - Req 17

The solar panel connected to the external sensor was tested using a multimeter to verify the batteries still produced the desired voltage output. Depleted batteries were connected to the circuit, the solar panel was placed in direct sunlight, and the batteries were verified to be fully charged by the end of the day.

5.17 Final testing on swamp cooler - Req 1

The fully developed SSC was run for 14 days continuously. All indoor and outdoor sensor readings continuously transmitted accurate sensor data, the data was properly received by the MC PCB and added to the database, and all cooler control settings and website features worked as desired.

The SSC with all components is shown below.

Figure 53: Fully functional SSC

6.0 Conclusion

The SSC meets all design requirements and is ready for home use. It provides:

- A fully functional automated, predictive, and smart algorithm, which determines the most efficient operation of the cooler
- Manual control of cooler settings or desired temperature from the auto algorithm via the UI kiosk
- Storing and displaying temperature, humidity, and previous cooler settings information
- Current upcoming forecast data, accessible on any device on the local network

Future development may include reducing the size of the MC and kiosk touchscreen to reduce unit price and space needed for installation.

7.0 References and Bibliography

[1] peppe8o, “Install Raspberry PI OS Lite in your Raspberry Pi,” *peppe8o*, 20-Jan-2021. [Online]. Available: https://peppe8o.com/install-raspberry-pi-os-lite-in-your-raspberry-pi/. [Accessed: 14-Apr-2021].

[2] Abdel-Fadeel, Waleed A., and Soubhi A. Hassanein. “CALCULATIONS OF THE OUTLET AIR CONDITIONS IN THE DIRECT EVAPORATIVE COOLER.” *JES. Journal of Engineering Sciences*, vol. 40, no. 5, 2012, pp. 1351–1358., doi:10.21608/jesaun.2012.114509.

[3] “OpenEnergyMonitor.” *Learn*, learn.openenergymonitor.org/sustainable-energy/building-energy-model/dynamicmodel.

[4] Rovai, Marcelo. “From Data to Graph: a Web Jorney With Flask and SQLite.” *Medium*, Medium, 29 Nov. 2018, medium.com/@rovai/from-data-to-graph-a-web-jorney-with-flask-and-sqlite-6c2ec9c0ad0. [Accessed: 07-Apr-2021]

[5] US Department of Commerce, NOAA. “API Web Service.” *National Weather Service*, NOAA's National Weather Service, 7 Dec. 2020, www.weather.gov/documentation/services-web-api.

[6] “CFM 101: What Is CFM And How To Calculate CFM.” *Top Cooling Fan*, 8 May 2020, www.topcoolingfan.com/what-is-cfm-and-how-to-calculate-cfm/.

[7] Parish, O. O., and T. W. Putnam. NASA Dryden Flight Research Center, Edwards, CA, 1977, "*EQUATIONS FOR THE DETERMINATION OF HUMIDITY FROM DEWPOINT AND PSYCHROMETRIC DATA*." NASA-TN-D-8401

[8] “How Humidity Effects Evaporative Cooling.” *Portacool Evaporative Coolers*, 25 Feb. 2021, portacool.com/2020/03/24/how-humidity-effects-evaporative-cooling/.

[9] “Do Evaporative (Aka Swamp) Coolers Work In My Area? Find Out With Our Evaporative Cooler Humidity Chart.” *Fireplaces, Stoves, and Water Heater Guides* | *Heattalk.com*, 18 July 2020, heattalk.com/evaporative-cooler-humidity-chart/.

[10] *Is a Swamp Cooler or Desert Cooler Your Best Outdoor Cooling Solution?*, www.alloutcool.com/swamp-cooler.html.

[11] “Sine Wave.” *MATLAB & Simulink*, www.mathworks.com/help/simulink/slref/thermal-model-of-a-house.htm.

[12] “Thermal Modeling of the Air Flow Inside and Around Your House.” *COMSOL Multiphysics*, www.comsol.com/blogs/thermal-modeling-of-the-air-flow-inside-and-around-your-house/.

8.0 Appendices

8.1 Appendix A - Schematics

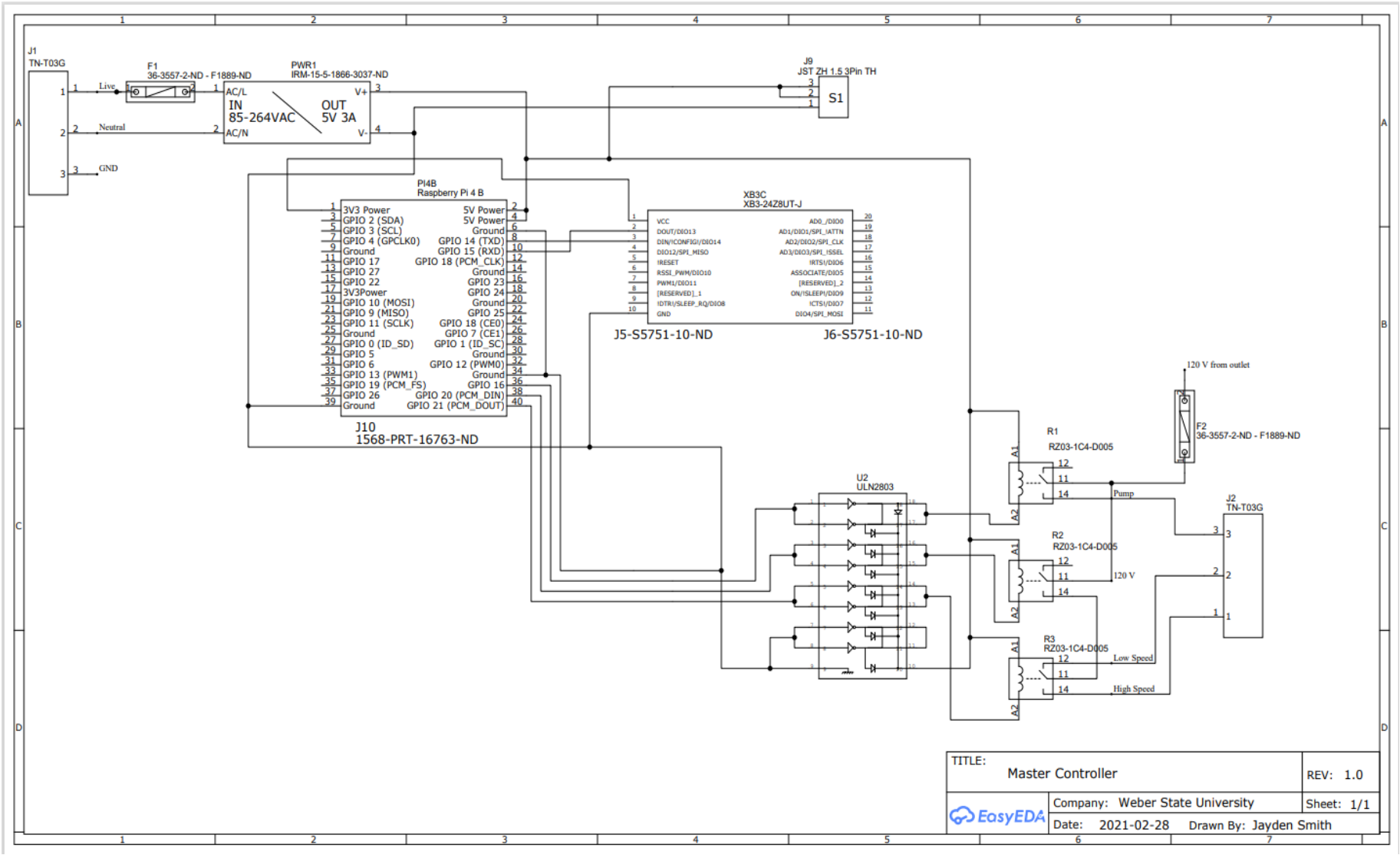

Figure 54: Master Controller Main PCB Schematic

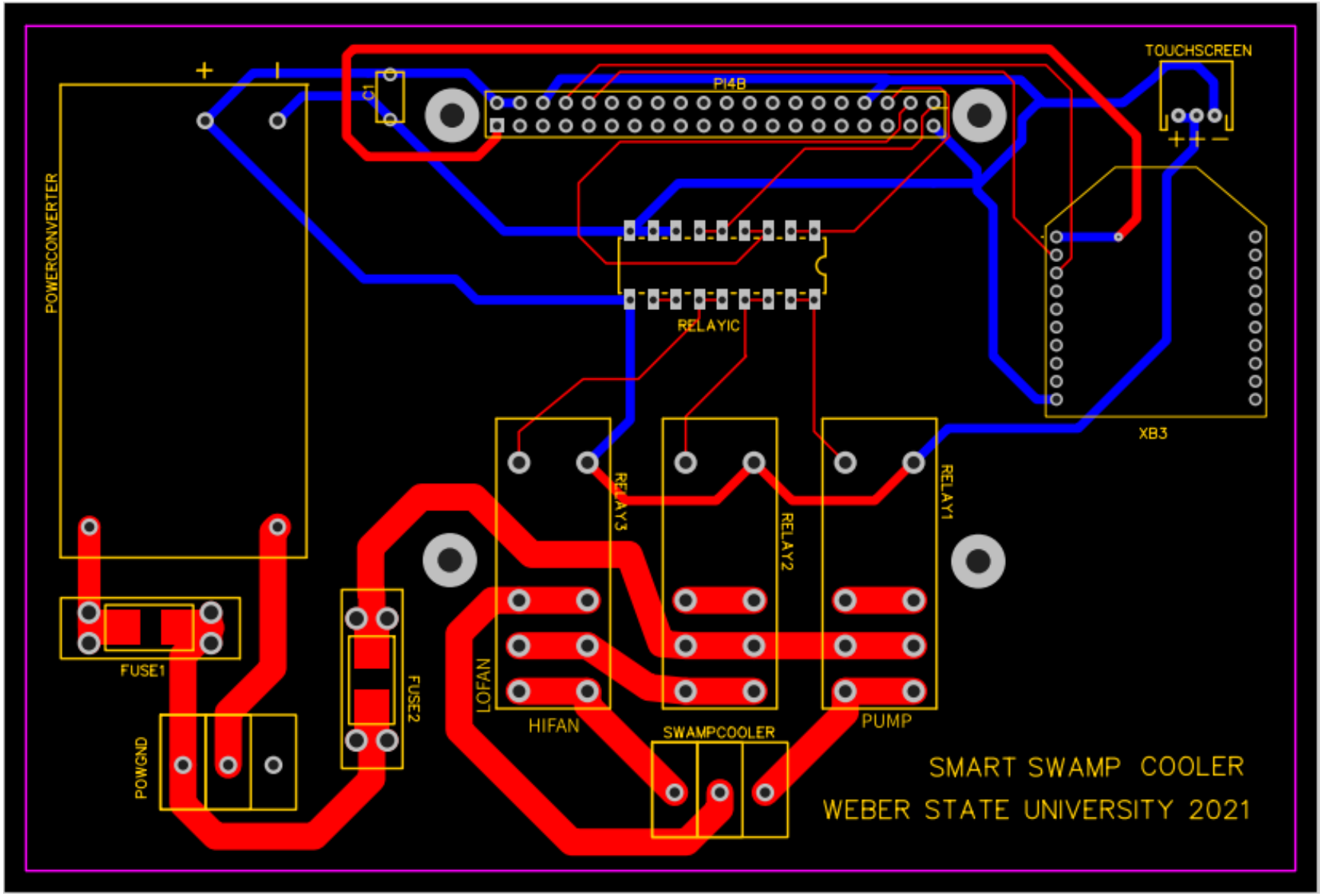

Figure 55: MC PCB Layout

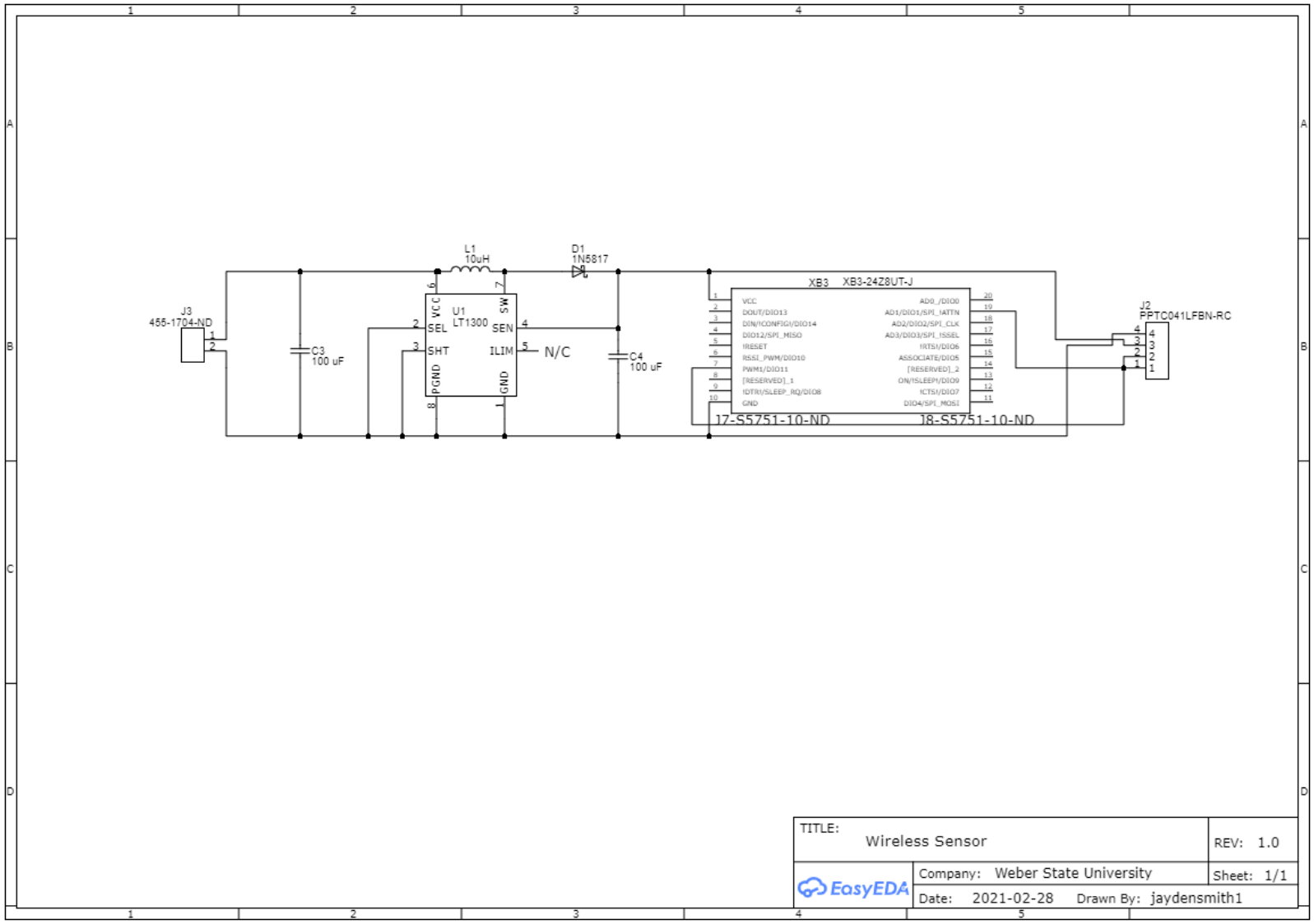

Figure 56: Wireless Node Schematic

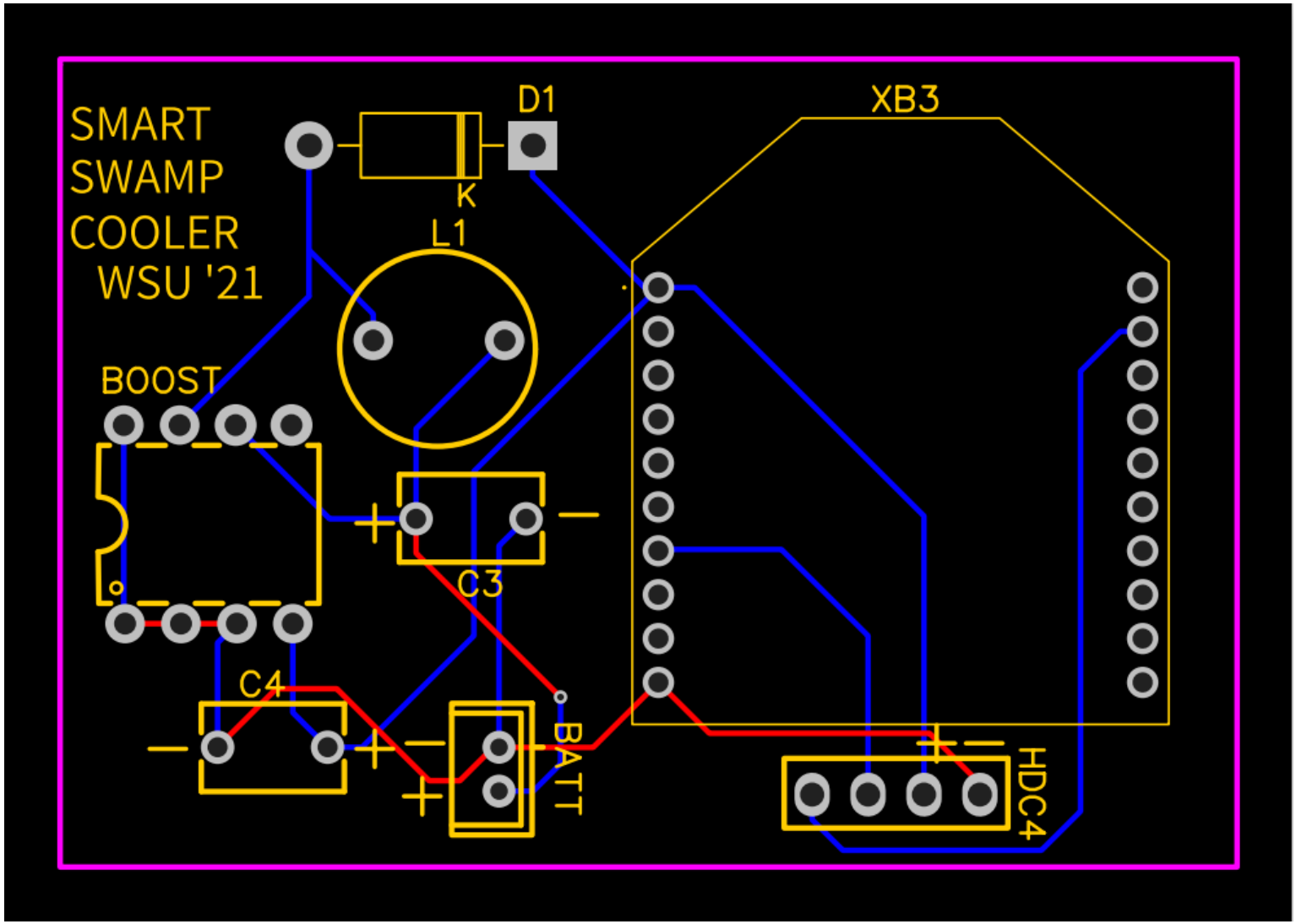

Figure 57: Wireless Node Printed Circuit Board Layout

8.2 Appendix B - Code Listing

See https://github.com/maxwellcox/Smart-Swamp-Cooler/tree/allJaydens/

8.3 Appendix C - Parts List

This section contains the list of parts for the Smart Swamp Cooler. The cost of the system was covered in part by the Ralph Nye Charitable Foundation through an Office Undergraduate Research (OUR) research grant totaling $740.

Parts List						
			DATE: February 28, 2021			
Ref #	**Part Description**	**Manufact.**	**Part #**	**Vendor**	**QTY**	**Unit Price**
J1,J2	3 Pin 5mm/0.2inch Pitch PCB Mount Screw Terminal Block Connector	Tnisesm	TN-T03G	amazon.com	2	$11.99
M1	Micro Center 32GB Class 10 Micro SDHC Flash Memory Card with Adapter (2 Pack)	Inland Store	B07K81Z6DF	amazon.com	1	$9.19
SCR1	Raspberry Pi Touchscreen Monitor, Upgraded 7'' IPS 1024X600 Dual-Speaker, USB HDMI Portable Monitor Capacitive Pi Display, Compatible with Raspberry Pi 3b+/Raspberry Pi 4b, Windows 7/8/10, Drive-Free	Lebula ltd.	LE070-01	amazon.com	1	$84.99

P1	Solar Battery Charger, 1W 4V Portable Black Solar Panel Charger Box for AA/AAA Battery Camping Hiking Travelling Charger Case	Dioche	B07FL548WC	amazon.com	1	$16.99
SNS1	GY-213V-HDC1080 HDC1080 Module Low Power High Accuracy Digital Humidity Sensor with Temperature Sensor	Huaban	MW-Temp-Humi-1027-1PCS	amazon.com	2	$10.99
MST1	Master Controller PCB	JLCPCB	Y5-3003772A	jlcpcb.com	1	$9.30
WIRE1	Wireless Sensor Node PCB	JLCPCB	Y3-3003772A	jlcpcb.com	2	$2.00
W1	JUMPER 03KR-6S-P - 03KR-6S-P 6"	JST Sales America Inc.	455-3379-ND	digikey.com	5	$1.03
R1,R2,R3	RELAY GEN PURPOSE SPDT 16A 5V	TE Connectivity Potter & Brumfield Relays	PB1275-ND	digikey.com	3	$1.98
W2,W3	JUMPER 02KR-6S-P - 02KR-6S-P 12"	JST Sales America Inc.	455-3154-ND	digikey.com	2	$1.07
J10	EXTENDED GPIO FEMALE HEADER - 2X	SparkFun Electronics	1568-PRT-16763-ND	digikey.com	1	$1.95
J3	CONN HEADER VERT 2POS 2MM	JST Sales	455-1704-ND	digikey.com	2	$0.17

		America Inc.				
U2	TRANS 8NPN DARL 50V 0.5A 18SOIC	Texas Instruments	ULN2803ADW-ND	digikey.com	1	$1.64
J4	CONN HDR 4POS 0.1 TIN PCB	Sullins Connector Solutions	S7002-ND	digikey.com	3	$0.45
L1	FIXED IND 10UH 3.6A 30 MOHM TH	Kemet	399-18905-ND	digikey.com	2	$0.96
PWR1	AC/DC CONVERTER 5V 15W	MEAN WELL USA Inc.	1866-3037-ND	digikey.com	1	$10.22
XB3,XB3C	RX TXRX MODULE 802.15.4 U.FL TH	Digi	602-2187-ND	digikey.com	3	$20.06
XB3-A	RF ANT 2.4GHZ WHIP STR U.FL FEM	Digi	A24-HABUF-P5I-ND	digikey.com	3	$5.50
U1	IC REG BUCK BST PROG 220MA 8DIP	Analog Devices Inc.	LT1300CN8#PBF-ND	digikey.com	3	$6.80
KEY1	RASPBERRY PI KEYBOARD (US) RED	Raspberry Pi	1690-RPI-KYB(US)RED-ND	digikey.com	1	$17.00
J5,J6,J7,J8	CONN HDR 10POS 0.079 GOLD PCB	Sullins Connector Solutions	S5751-10-ND	digikey.com	8	$0.87
M2	32GB NOOBS MICROSD CARD	Raspberry Pi	2648-SC0251B-ND	digikey.com	2	$9.28

MOUSE 1	RASPBERRY PI MOUSE RED	Raspberry Pi	1690-RPI-MOUSERED-ND	digikey.com	1	$8.00
F1,F2	FUSE BLOCK BLADE 500V 30A PCB	Keystone Electronics	36-3557-2-ND	digikey.com	2	$1.16
TEST1	WHITE MICRO-HDMI TO STANDARD-M C	Raspberry Pi	1690-RPIHDMICABLEWHITE-ND	digikey.com	1	$5.00
PI4B	RASPBERRY PI 4 MODEL B 8GB SDRAM	Raspberry Pi	1690-RASPBERRYPI4MODELB8G-ND	digikey.com	1	$75.00
TEST2	AC/DC WALL MNT ADAPTER 5.1V 15W	Raspberry Pi	1690-RPIUSB-C POWERSUPPLY BLACKUS-ND	digikey.com	1	$8.00
D1	DIODE SCHOTTKY 20V 1A DO41	STMicroelectronics	497-4547-1-ND	digikey.com	3	$0.41
C3,C4	CAP ALUM 100UF 20% 10V RADIAL	Panasonic Electronic Components	P19577CT-ND	digikey.com	6	$0.34
J9	CONN HEADER VERT 3POS 2MM	JST Sales America Inc.	455-1818-ND	digikey.com	1	$0.28
TEST3,TEST4	FTDI BREAKOUT (FT231X) - USB-SER	Watterott Electronic	2100-201832-ND	digikey.com	2	$10.26

		GmbH				
S1	Flat Head Thread-Forming Screws for Brittle Plastic, 410 Stainless Steel, Number 4 Size, 1/2" Long Pack of 100	McMaster-Carr	96068A105	mcmaster.com	1	$5.51
S2	Flat Head Thread-Forming Screws for Brittle Plastic, 410 Stainless Steel, Number 6 Size, 1/2" Long Pack of 100	McMaster-Carr	96068A153	mcmaster.com	1	$5.67
TOTAL						**$474.25**