CNIT 546 Lab 6: Thermostat System

Dhanish Patil and Colin Saumure

Table of Contents

System Design

Many IoT-enabled thermostat solutions exist for both home and enterprise applications. In order to better understand the components of such a system, the design team opted to create their own. The project combines local control with cloud-based remote control and monitoring, allowing users to quickly monitor indoor conditions and make adjustments from anywhere.

Circuit Components

An ESP32-Pico development board was programmed with MicroPython to implement the desired functionality for the thermostat system. The ESP32 was connected to a Digital Humidity and Temperature (DHT11) sensor via GPIO pin 0. The DHT11 provides an 8-bit measured value over UART communication. Additionally three push buttons were connected to GPIO pins 27, 12, and 14. These buttons acted as thermostat mode toggle, temperature up, and temperature down, respectively. The button outputs were connected to a SN74LS14 inverter and capacitor system in order to filter out signal bouncing.

A Newhaven 20x4 character Liquid Crystal Display (LCD) was used to provide a digital text output in response to input events. The LCD was configured to display the indoor temperature measured using the DHT11, the user's desired (setpoint) temperature, the current mode of the thermostat system (i.e. off, heat, or cool), and the current outdoor temperature. The LCD and visual components are shown in Figure 1.

Figure 1: 20x4 LCD showing thermostat status information

The system was originally connected to a unipolar stepper motor that was intended to emulate a fan, adding a mechanical component to the circuit. This idea was eventually abandoned for the sake of time, and a two-channel relay unit was used in its place. In practice, a thermostat system would likely be attached to a relay to allow control over the high-voltage electronics involved in air conditioning units and heaters.

Circuit Schematic

A detailed schematic of the interconnections between sensors and actuators was created to assist in understanding and future replication, depicted in Figure 1. Additionally, photographs of the breadboard configuration of the primary board and the button debouncer board are shown in Figure 2. A logical diagram of the system's components is included in Figure 3.

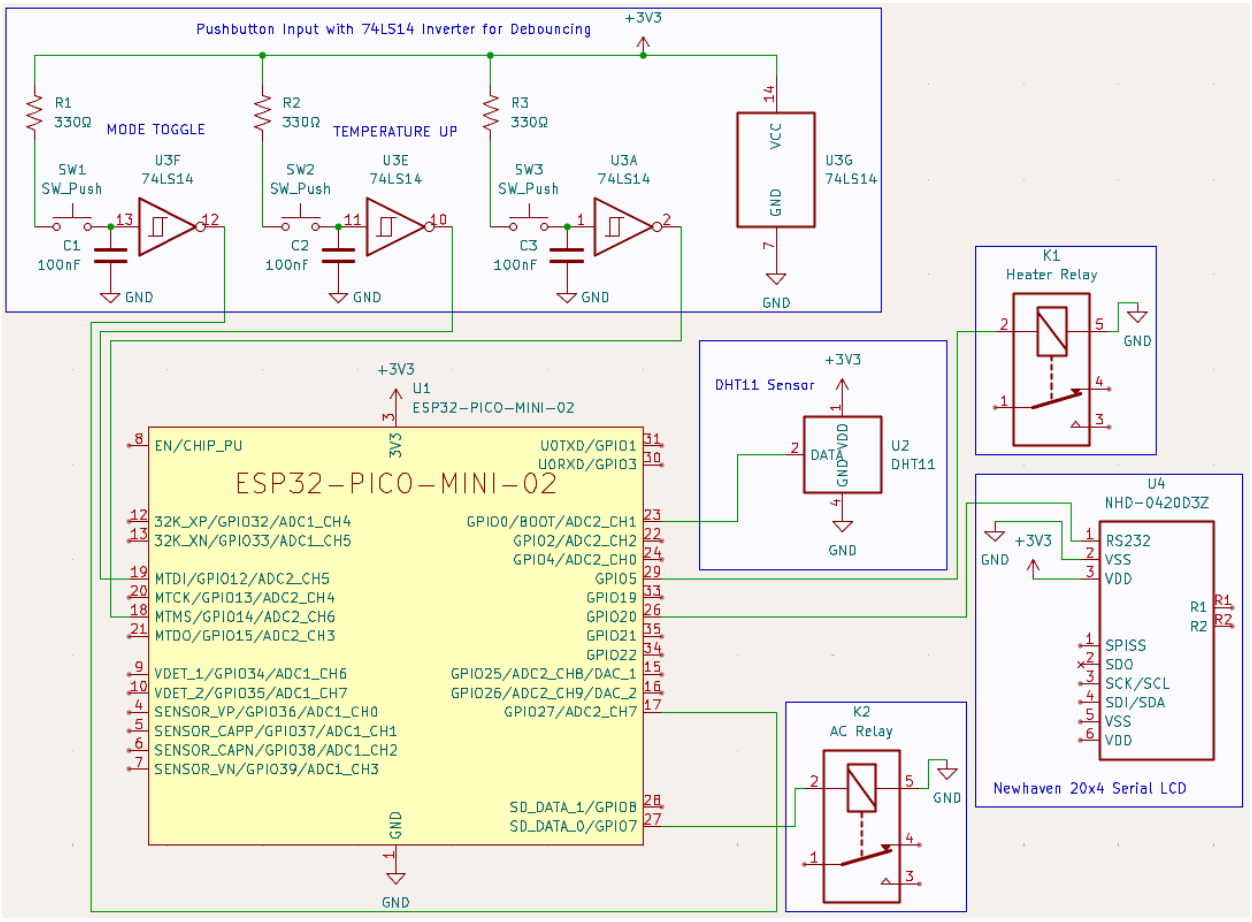

Figure 1: Circuit Schematic

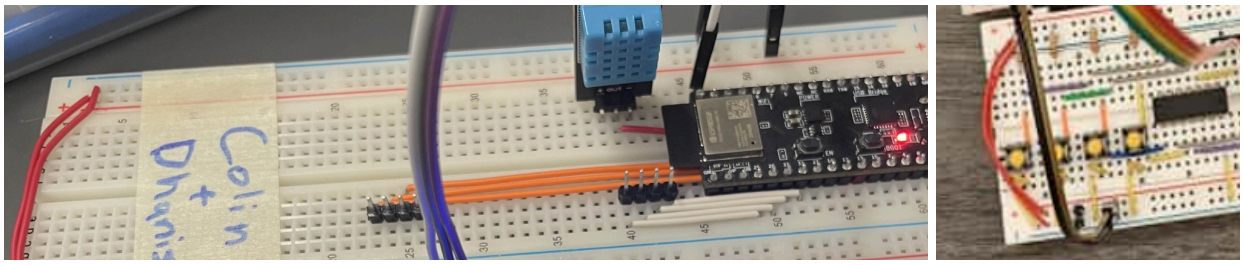

Figure 2: Physical Wiring Setup

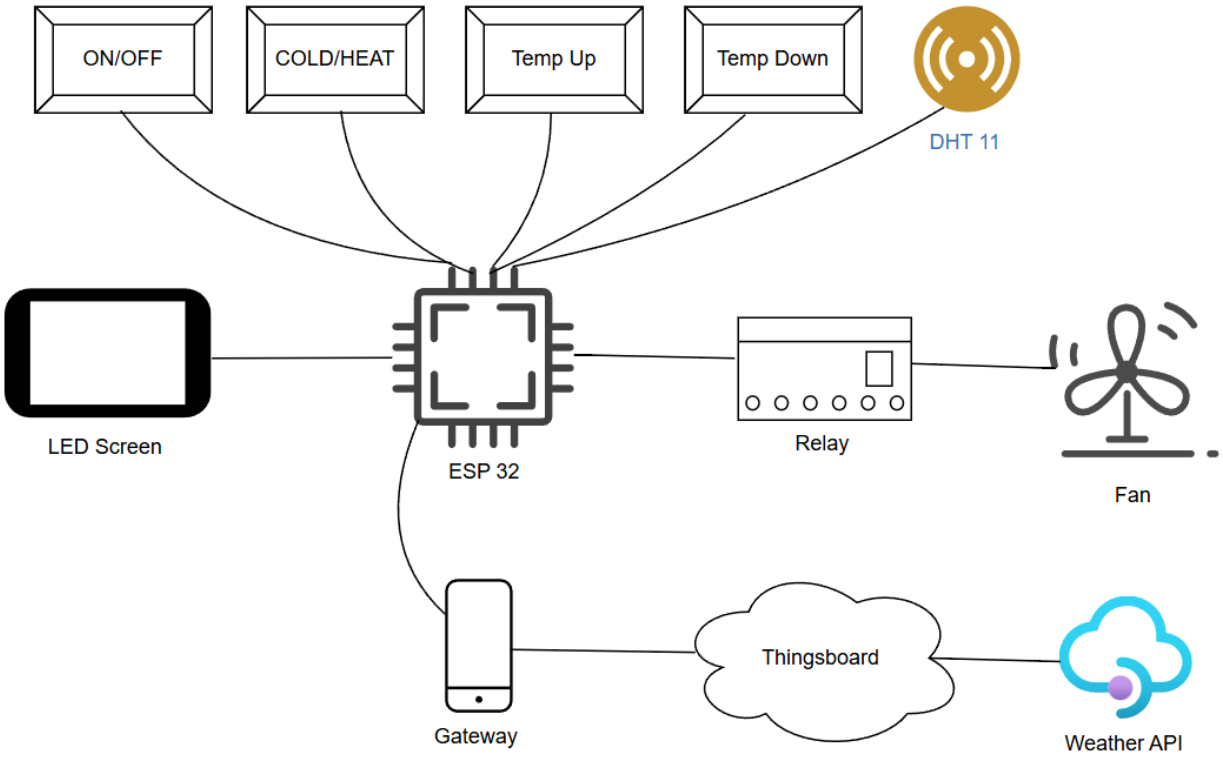

Figure 3: Logical Diagram of Thermostat Components

Cloud Telemetry

With the circuitry connected, the system was entirely sufficient for on-premises control. However, the thermostat was also intended to be capable of remote control. To meet these ends, the ESP32 was configured to transmit telemetry and receive instructions from a cloud platform. ThingsBoard was selected for its free tier and the design team's existing familiarity with the platform. The ESP32 was programmed to connect to a Wi-Fi network using WPA2 Pre-Shared Keys (PSK). For testing, a cellular hotspot was used. Once the connection was established, the device would begin transmitting live telemetry to ThingsBoard's MQTT telemetry topic every ten seconds. In addition, it subscribed to ThingsBoard's Remote Procedure Call (RPC) topic so that it could listen for and handle remote instructions. Telemetry included the measured temperature, the measured humidity (unused), the setpoint temperature, and the thermostat mode.

A dashboard was created to provide a smooth user experience when remotely configuring the thermostat mode and set point temperature. The dashboard provides a status indicator for the measured indoor temperature, a dial to configure the desired setpoint temperature, and two switches that control the mode. Notably, the team was unable to make a three-way slider to control the mode. A power switch and a heat/cool switch were created to handle this limitation. If the power switch was set to off, the mode would be set to off. If the power switch was set to on, the mode would be set to heat. If the heat/cool slider were changed, the mode would be set to match, and the power switch would show as powered on. The dashboard is shown in Figure 3.

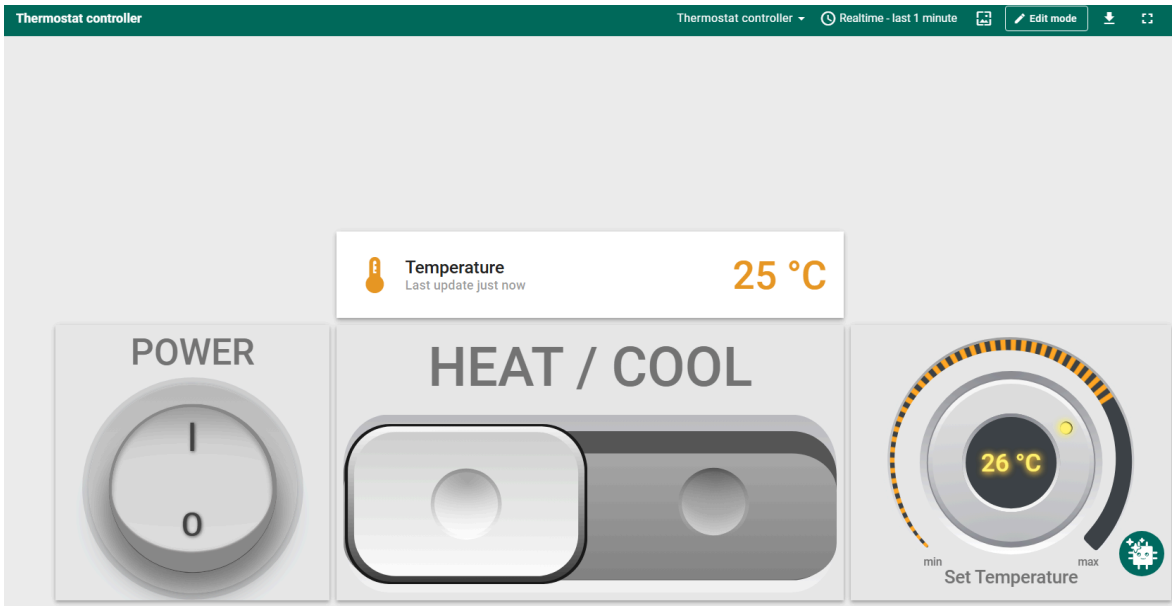

Figure 3: Thermostat Controller Dashboard on ThingsBoard

In addition to hosting a dashboard, ThingsBoard was also used to control actuation logic and outside temperature retrieval. A rule chain was created to trigger on incoming telemetry messages from the thermostat unit. Any time new data was received (every 10s), the rule chain would decide whether the heater or AC should be turned on. A script was used to check the thermostat's mode and whether the measured temperature was dissatisfactory with respect to the setpoint temperature. If the mode was set to heat and the observed temperature was lower than the setpoint, the heater would be turned on. Conversely, if set to cool and the observed was greater than the setpoint, the AC would be turned on.

The rule chain was also configured to concurrently send a request to weatherapi.com each time new telemetry was received. For simplicity, the location to query was hardcoded as West Lafayette, IN; however, it would have been feasible to introduce IP geolocation if the thermostat unit were likely to be used in multiple regions. Once a response from the weather API was received, the current outdoor temperature was sent back to the ESP32 using an RPC message.

The created rule chain was manually linked into the default root rule chain, as it was determined that the root rule chain performs core functionality in handling RPC messages. The created rule chain is depicted in Figure 4. Basic transform scripts (shown in blue) were used to prepare a new message that would be sent via RPC. Additionally, a sample weather API response is included in Figure 5. Sample received telemetry is shown in Figure 6, and RPC messages used to update the thermostat are included in Figures 7 and 8.

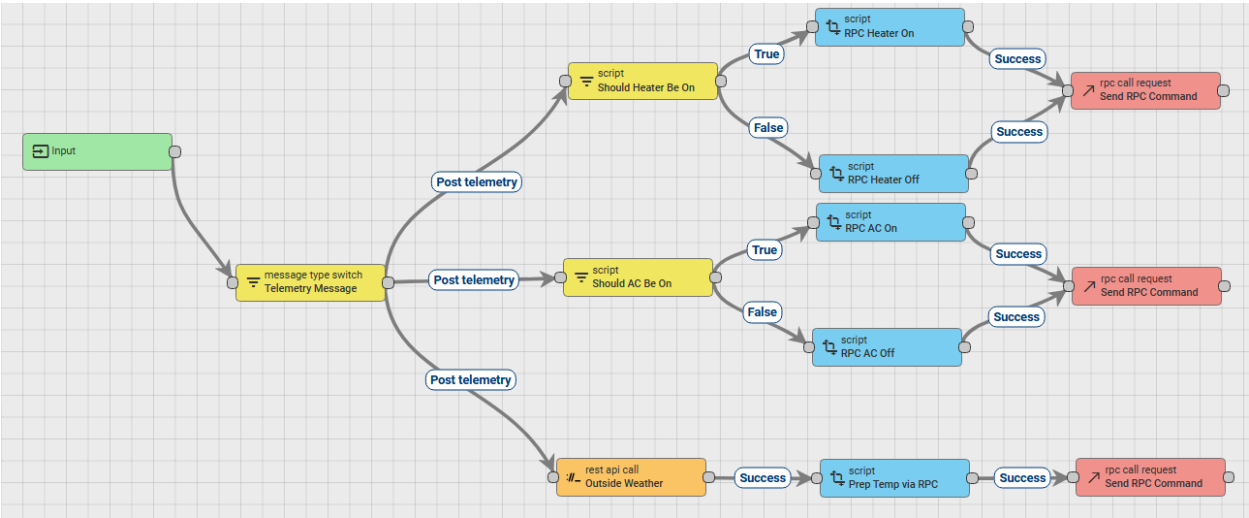

Figure 4: Custom rule chain for relay actuation logic and weather API calls

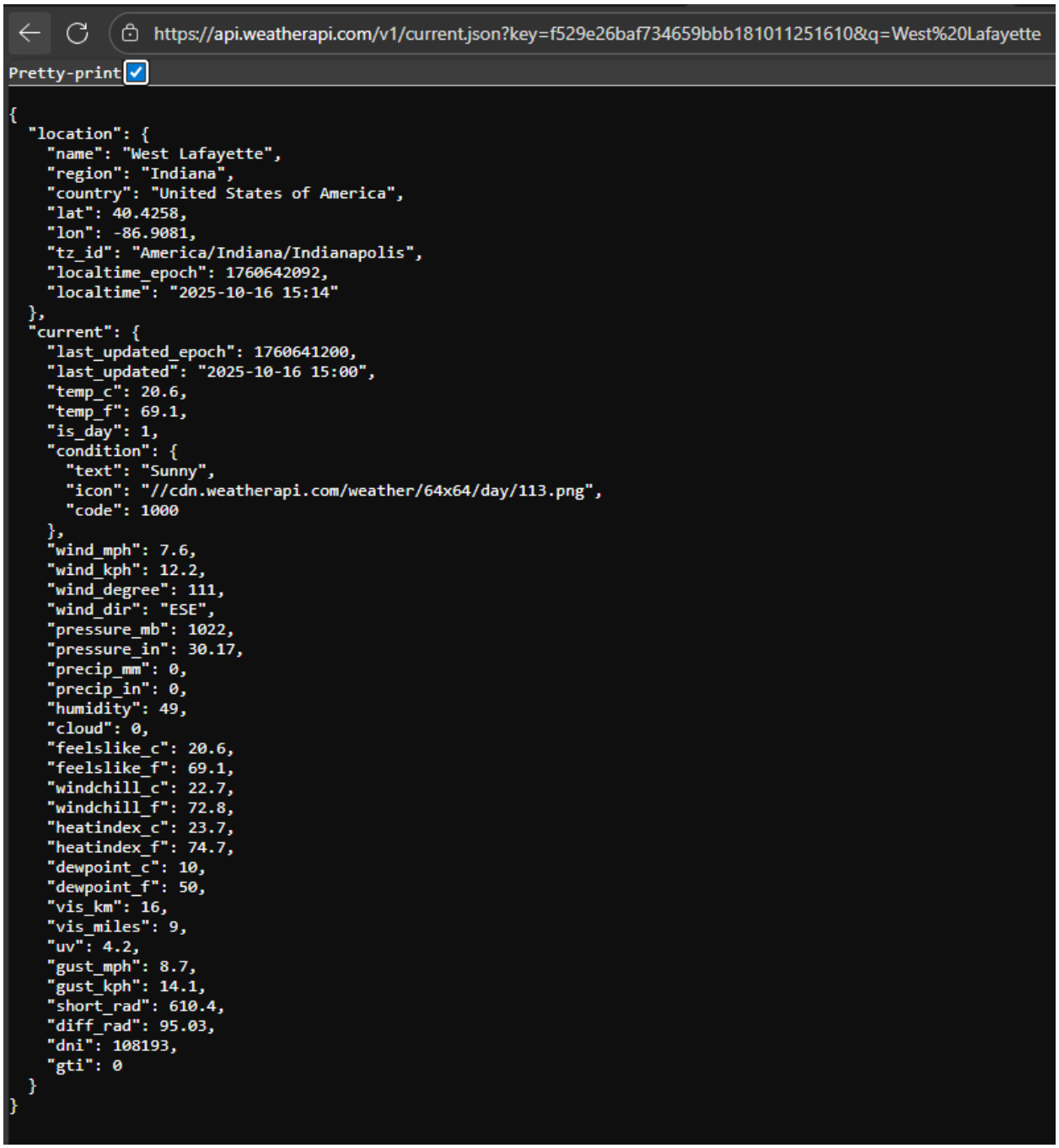

Figure 5: Sample weather API response

```
{
    "humidity": 43,
    "temperature": 24,
    "mode": 0,
    "setpoint": 21
}
```


Figure 6: Telemetry Data sent from ESP32 to Thingsboard


```
{
    "method": "setFan",
    "params": false
}
```


Figure 7: RPC message sent from Thingsboard to ESP32 related to the status of the fan


```
{
    "method": "setOutside",
    "params": 21.1
}
```


Figure 8: RPC message sent from Thingsboard to ESP32 related Outside Weather API

Appendix A: Source Code

The following section contains the source code that was deployed to the microcontroller unit to handle sensor input and actuator output. The code was designed in a modular fashion, with individual components separated into their own files. The Main.py file was used as the entrypoint script and imported the remaining modules as needed. For improved version control, the project was maintained on GitHub. Viewing the source code on GitHub may offer improved readability, but the contents of each file have also been included here. Because this code was made available online, the Thingsboard authentication key and Wi-Fi network password were redacted from Main.py.

Main.py

```
# Import Built-In Libraries
import machine
import ujson
import time

# Import Custom Libraries
import Stepper
import Buttons
import LCD
import DHT
import Network
from Thermostat import Thermostat

# Heat/Cool Toggle
toggle_button = machine.Pin(27, machine.Pin.IN,
machine.Pin.PULL_UP)

# Temperature Up/Down Buttons
up_button = machine.Pin(12, machine.Pin.IN, machine.Pin.PULL_UP)
dn_button = machine.Pin(14, machine.Pin.IN, machine.Pin.PULL_UP)

if __name__ == "__main__":
    print("Running!")

    Buttons.init_mode_toggle_button(toggle_button)
    Buttons.init_temp_up_button(up_button)
    Buttons.init_temp_dn_button(dn_button)

    screen = LCD.LCD()

    # Define callback for DHT periodic readings
    def sensorOnRead(temperature, _):
```

```
        if temperature != -1:
            screen.set_cursor(0, 0)
            screen.write_string(f"Temp: {temperature} C")
        else:
            screen.set_cursor(0, 0)
            screen.write_string("Temp: -- C")

    sensor = DHT.Sensor(0)
    sensor.start_periodic_read(2000, sensorOnRead)

    myThermostat = Thermostat(22, 0, screen)
    Buttons.set_thermostat(myThermostat)

    # Initial Display
    screen.set_cursor(0, 0)
    screen.write_string(f"Temp: -- C")
    screen.set_cursor(0, 16) # Write to end of line

screen.write_string(f"{myThermostat.modes[myThermostat.mode]}")
    screen.set_line(1, f"Set:  {myThermostat.temperature} C")
    #screen.set_line(3, f"Mode:
{myThermostat.modes[myThermostat.mode]} ")
    screen.set_line(3, f"Out:  -- C")

    # Define GPIO pins for stepper motor
    motor = Stepper.StepperMotor(33, 32, 26, 25)

    heater = machine.Pin(5, machine.Pin.OUT)
    heater.value(1)

    ac = machine.Pin(7, machine.Pin.OUT)
    ac.value(1)

    def rpc_callback(topic, msg):
        print("Received message:", msg)
        try:
            data = ujson.loads(msg)
            if data["method"] == "setOutside":
                outside = data["params"]
                if isinstance(outside, (int, float)):
                    myThermostat.outside = outside
                    screen.set_line(3, f"Out:  {int(outside)}
C")
            elif data["method"] == "setPower":
                power = data["params"]
                if (power == False):
                    myThermostat.set_mode(0) # Set to OFF
```

```
                else:
                    myThermostat.set_mode(1) # Set to HEAT
            elif data["method"] == "setMode":
                mode = data["params"]
                if mode in [0, 1, 2]:
                    myThermostat.set_mode(mode)
            elif data["method"] == "setTemp":
                temp = data["params"]
                if isinstance(temp, int) and (16 <= temp <= 30):
                    myThermostat.set_temperature(temp)
            elif data["method"] == "setHeater":
                heaterOn = data["params"]
                if heaterOn:
                    heater.value(0)
                    #motor.start_rotation(5) # Rotate
indefinitely
                else:
                    heater.value(1)
                    #motor.stop_rotation()
            elif data["method"] == "setAC":
                acOn = data["params"]
                if acOn:
                    ac.value(0)
                    #motor.start_rotation(5) # Rotate
indefinitely
                else:
                    ac.value(1)
                    #motor.stop_rotation()
        except Exception as e:
            print("Error processing message:", e)

    # Connect to Wi-Fi
    wlan = Network.connect_to_wifi("Phone Hotspot", "password")
    try:
        mqtt_client =
Network.init_MQTT("mqtt.thingsboard.cloud", "token", "",
rpc_callback, 1883)
    except Exception as e:
        print("MQTT connect failed:", e)

    while True:
        telemetry = {
            "temperature": sensor.temperature,
            "humidity": sensor.humidity,
            "setpoint": myThermostat.temperature,
            "mode": myThermostat.mode
        }
```

```
        # Emulate screen to console once per iteration
        print("\n\n\n")

        print(f"Temp: {sensor.temperature} C
{myThermostat.modes[myThermostat.mode]}")
        print(f"Set:  {myThermostat.temperature} C")
        print("")
        print(f"Out:  {int(myThermostat.outside) if
hasattr(myThermostat, 'outside') else '--'} C")
        print("\n")

        Network.publish_message(mqtt_client,
"v1/devices/me/telemetry", ujson.dumps(telemetry))
        time.sleep(10)
```

Buttons.py

```
import machine

# Module-level reference to the Thermostat instance
_thermostat = None

# Setter to allow Main.py to provide the Thermostat instance
def set_thermostat(instance):
    global _thermostat
    _thermostat = instance


# Initialize Buttons with Interrupts
def init_mode_toggle_button(pin):
    pin.irq(trigger=machine.Pin.IRQ_FALLING,
handler=mode_toggle_handler)

def init_temp_up_button(pin):
    pin.irq(trigger=machine.Pin.IRQ_FALLING,
handler=temp_up_handler)

def init_temp_dn_button(pin):
    pin.irq(trigger=machine.Pin.IRQ_FALLING,
handler=temp_dn_handler)


# Create Handler Functions for Interrupts
def mode_toggle_handler(_):
    if _thermostat is not None:
        _thermostat.set_mode(_thermostat.mode + 1)
    else:
```

```
        print("Thermostat instance not set!")


def temp_up_handler(_):
    if _thermostat is not None:
        _thermostat.temp_up()
    else:
        print("Thermostat instance not set!")

def temp_dn_handler(_):
    if _thermostat is not None:
        _thermostat.temp_down()
    else:
        print("Thermostat instance not set!")
```

DHT.py

```
import machine
import dht

class Sensor:
    def __init__(self, pin):
        self.sensor = dht.DHT11(machine.Pin(pin,
machine.Pin.IN))
        self.temperature = -1
        self.humidity = -1
        self._timer = None
        self._callback = None

    def start_periodic_read(self, interval_ms=2000,
callback=None):
        self._callback = callback
        if self._timer is None:
            self._timer = machine.Timer(0)
            self._timer.init(period=interval_ms,
mode=machine.Timer.PERIODIC, callback=self._read)

    def stop_periodic_read(self):
        if self._timer:
            self._timer.deinit()
            self._timer = None

    def _read(self, _):
        try:
            self.sensor.measure()
            self.temperature = self.sensor.temperature()
            self.humidity = self.sensor.humidity()
            if self._callback:
```

```
                self._callback(self.temperature, self.humidity)
        except Exception as e:
            print("Error reading sensor:", e)
            self.temperature = -1
            self.humidity = -1
```

LCD.py

```
import machine

# For use with Newhaven 20x4 LCD via UART
class LCD:
    def __init__(self, uart_port=1, baudrate=9600, tx_pin=20,
rx_pin=21):
        self.lines = ["", "", "", ""]

        # Initialize UART for LCD communication
        self.uart = machine.UART(uart_port, baudrate=baudrate,
tx=tx_pin, rx=rx_pin)
        self.uart.init(bits=8, parity=None, stop=1)

        # Turn on LCD Backlight
        self.uart.write(bytearray([0x41]))
        self.clear()
        self.home()

    def send_command(self, cmd):
        self.uart.write(bytearray([0xFE, cmd]))

    def set_cursor_addr(self, addr):
        self.uart.write(bytearray([0xFE, 0x45, addr]))

    def set_cursor(self, row, col):
        # Handle out-of-bounds values
        row = 0 if row < 0 else row
        row = 3 if row > 3 else row
        col = 0 if col < 0 else col
        col = 19 if col > 19 else col

        # Store calculated position value
        pos = 0

        if row == 0:
            pos = 0
        elif row == 1:
            pos = 0x40
        elif row == 2:
            pos = 0x14
```

```
        elif row == 3:
            pos = 0x54

        pos += col

        # Send command to set cursor position
        self.set_cursor_addr(pos)

    def clear(self):
        self.send_command(0x51) # Clear display command

    def home(self):
        self.send_command(0x46) # Return home command

    def write_string(self, s):
        self.uart.write(s.encode('utf-8')) # Send string to LCD

    def set_line(self, line_num, text):
        if line_num < 0 or line_num > 3:
            return

        # Pad text with spaces and truncate to 20 chars
        self.lines[line_num] = f"{text:<20}"[:20]
        self.set_cursor(line_num, 0)
        self.write_string(self.lines[line_num])
```

Network.py

```
import network
import machine
from umqtt.simple import MQTTClient

def connect_to_wifi(ssid, password):
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    if not wlan.isconnected():
        print('Connecting to Wi-Fi...')
        wlan.connect(ssid, password)
        while not wlan.isconnected():
            pass
    print('Connected to Wi-Fi:', wlan.ifconfig())
    return wlan

def init_MQTT(server, user, password, callback, port=1883):
    client = MQTTClient(
        client_id="esp32",
        server=server,
        user=user,
```

```
        password=password,
        port=port
    )
    client.set_callback(callback)
    print("Constructed MQTT client")
    client.connect()
    print('Connected to MQTT broker')
    client.subscribe(b"v1/devices/me/rpc/request/+")
    print("Subscribed to RPC topic")

    def _read(t):
        client.check_msg()


    MQTT_Timer = machine.Timer(1)
    MQTT_Timer.init(period=50, mode=machine.Timer.PERIODIC,
callback=_read)

    return (client, MQTT_Timer)


def publish_message(client, topic, message):
    client.publish(topic, message)
    print("Published updated cloud telemetry")
```

Stepper.py

```
import time
from machine import Pin, Timer

class StepperMotor:
    # Define Stepper Motor Step Sequences
    wave_steps = [0x01, 0x02, 0x04, 0x08]
    full_steps = [0x03, 0x06, 0x0C, 0x09]
    half_steps = [0x01, 0x03, 0x02, 0x06, 0x04, 0x0C, 0x08,
0x09]

    # Construct the Stepper Motor Object
    # Mode 0 = Wave Step, Mode 1 = Full Step, Mode 2 = Half Step
    def __init__(self, pin0, pin1, pin2, pin3, mode=1, delay=3):
        # Define GPIO pins for stepper motor
        self.pins = [
            Pin(pin0, Pin.OUT),
            Pin(pin1, Pin.OUT),
            Pin(pin2, Pin.OUT),
            Pin(pin3, Pin.OUT)
        ]
```

```
        # Set delay between steps (in milliseconds)
        self.delay = delay

        # Validate mode input
        if mode < 0 or mode > 2:
            mode = 1

        # Set operating mode
        self.mode = mode

        # Timer and step tracking for non-blocking rotation
        self._timer = None
        self._steps_remaining = 0
        self._current_step = 0

    # Setter for Delay
    def set_delay(self, delay):
        self.delay = delay

    # Setter for Mode
    def set_mode(self, mode):
        if mode < 0 or mode > 2:
            mode = 1
        self.mode = mode

    # Rotate the Stepper Motor
    def set_step(self, index):
        step_sequence = []
        if self.mode == 0:
            step_sequence = self.wave_steps
        elif self.mode == 1:
            step_sequence = self.full_steps
        else:
            step_sequence = self.half_steps

        # Handle differing lengths of step sequences
        index = index % len(step_sequence)

        # Set GPIO pins based on step sequence
        self.pins[0].value(1 if (step_sequence[index] & 0x01)
else 0)
        self.pins[1].value(1 if (step_sequence[index] & 0x02)
else 0)
        self.pins[2].value(1 if (step_sequence[index] & 0x04)
else 0)
        self.pins[3].value(1 if (step_sequence[index] & 0x08)
else 0)
```

```
    # Callback for internal timer
    # After delay timer, increment the current step
    def _timer_callback(self, t):
        if self._steps_remaining > 0:
            self.set_step(self._current_step)
            self._current_step += 1
            self._steps_remaining -= 1
        elif self._steps_remaining == -1:
            # Rotate until stopped
            self.set_step(self._current_step)
            self._current_step += 1

            # Roll over to reduce memory usage
            if self._current_step >= 256:
                self._current_step = 0
        else:
            self.stop_rotation()

    # Start rotation for a given number of revolutions
(non-blocking)
    def start_rotation(self, revolutions):
        if revolutions == -1:
            # Rotate indefinitely
            self._steps_remaining = -1
            self._current_step = 0
        else:
            steps_per_revolution = 512  # Assuming 512 steps per
revolution
            total_steps = int(revolutions *
steps_per_revolution)
            self._steps_remaining = total_steps
            self._current_step = 0

        # Cleanup any existing timer
        if self._timer:
            self._timer.deinit()

        # Initialize and start the timer for non-blocking
rotation
        self._timer = Timer(2)
        self._timer.init(period=self.delay, mode=Timer.PERIODIC,
callback=self._timer_callback)

    # Stop the motor rotation
    def stop_rotation(self):
        if self._timer:
```

```
            self._timer.deinit()
            self._timer = None
        self._steps_remaining = 0
```

Thermostat.py

```
class Thermostat:
    # Define Mode Strings
    modes = ["OFF", "HEAT", "COOL"]

    def __init__(self, temperature=22, mode=0, screen=None):
        self.temperature = temperature
        self.mode = mode
        self.screen = screen

    def set_temperature(self, temperature):
        self.temperature = temperature
        if self.screen:
            self.screen.set_line(1, f"Set:  {self.temperature}
C")
        print(f"Temperature set: {self.temperature}")

    def temp_up(self):
        self.temperature += 1
        if self.screen:
            self.screen.set_line(1, f"Set:  {self.temperature}
C")
        print(f"Temperature up: {self.temperature}")

    def temp_down(self):
        self.temperature -= 1
        if self.screen:
            self.screen.set_line(1, f"Set:  {self.temperature}
C")
        print(f"Temperature down: {self.temperature}")

    def set_mode(self, mode):
        self.mode = mode % 3
        if self.screen:
            self.screen.set_cursor(0, 16) # Write to end of line
            self.screen.write_string((self.modes[self.mode] + "
")[:4]) # Ensure always 4 chars
            #self.screen.set_line(3, f"Mode:
{self.modes[self.mode]} ")
        print(f"Mode set: {self.modes[self.mode]}")
```