Chapter 8: Class Project

Time: 2 ¾ Hours

At the end of this chapter you will have created an IoT thermostat simulator. It will:

- Read local “temperature”
- Display thermostat data on the TFT display
- Publish thermostat data to the cloud
- Get “set temperature” updates from the cloud

8.1 Introduction

Your project is to build an IoT thermostat simulator. It will connect to the class AWS account and will publish updates to a *thing* which you will create. You will use the potentiometer on your kit to simulate measuring temperature, use the CapSense buttons to determine the “set temperature” for your thermostat, determine what mode the thermostat should be in (heating, cooling, or idle) and display this data on the TFT display.

The required functionality is:

1. Use an ADC to read the voltage on the potentiometer. Use this as a simulated actual temperature.
2. Use the CapSense buttons to determine the set temperature for your thermometer.
3. Use the simulated actual and set temperatures to determine what mode your thermostat needs to be in. Light the red user LED for heating and blue user LED for cooling.
4. Display the actual temperature, set temperature, and mode on the TFT display.
5. Any time the actual or set temperatures change publish this new data to the cloud.
6. Subscribe to the current/state/desired topic of your *thing*’s shadow to determine when the set temperature has been updated remotely and update the local set value.
7. Advanced: Display graphics for heating and cooling mode on the TFT display.
8. Advanced: Implement the IoT functionality using HTTPS instead of MQTT.

8.2 Details and Hints

If you are using MQTT, it is probably best to start with the AnyCloud MQTT Client Example (see Chapter 7C).

You will edit the message so that it sends JSON messages to update the shadow instead of just alternately sending TURN ON and TURN OFF.

If you are using HTTP, the HTTP Bin example is a good starting point (see Chapter 7B). You will use POST requests to send your data to the server.

You will connect to the class AWS IoT endpoint:

amk6m51qrxr2u.iot.us-east-1.amazonaws.com

Your *thing* name will be "thermostat_<init>" where <init> will be your initials.

If you used the class AWS account for previous exercises, you may reuse a previously created *thing*.

Use an ADC to simulate actual temperature

You will need to use an ADC to read the value of the potentiometer and then convert that to a simulated temperature. See the ADC description and exercise in Chapter 2 or look at a code example. You should read the values on a regular basis (e.g. every 100ms). It is recommended to have an RTOS task to handle the ADC. Whenever the simulated actual temperature changes be sure to update the display and the cloud.

Use CapSense buttons to control set temperature

You will use the CapSense buttons to change the set temperature. Look for a CapSense code example. It is recommended to have an RTOS task to handle CapSense. Whenever the set temperature changes be sure to update the display and publish the new value to the cloud.

Use TFT to display information

The TFT display for your *thing* should look something like this:

```
Actual Temperature: XX
Set Temperature: XX
Mode: XX
```

The Mode variable could display a bitmap to indicate what state it is in. For example, a flame and a snowflake to indicate heating and cooling.

Hint You should only update the display if one of the values has changed. Use a task notification or a semaphore.

The starting (empty) shadow for your *thing* should look like the following. You will publish (MQTT) or post (HTTP) JSON messages to the *thing* shadow to provide updates.

Shadow State:

```
{
  "reported":{
      "actualTemperature": 0,
      "setTemperature": 0,
      "mode": "Idle",
      "IPAddress": "0.0.0.0"
  }
}
```

You can use the snprintf function to create the JSON messages.

Remember that spaces and carriage returns are not required in the JSON message. Also remember that quotation marks in the message must be escaped with a \ character in the code. For example, to create a JSON message to send the actual temperature from the integer variable actualTemp, you could do something like this:

```
char json[100];
snprintf(json, sizeof(json),
"{\"state\":{\"reported\":{\"actualTemperature\":%d}}}", actualTemp);
```

Hint Note that snprintf was used instead of sprintf so that sending more characters than the buffer size will not lead to a pointer overflow error, which can be very difficult to debug. However, you should still make sure the buffer is large enough for the message or else your formatted JSON message will not be correct.

Hint When doing initial testing, use the MQTT Test Client on the AWS site to examine the messages that you are sending. Note that this works even if you are using HTTP to post data since the update to the shadow still causes a notification to any MQTT subscribers. For example, to see all shadow messages for the *thing* named thermostat_key, you would subscribe to:

$aws/things/thermostat_key/shadow/#

Messages that show up on the topic *$aws/things/ thermostat_key /shadow/update* are the messages that you are sending. You will also see messages that tell you whether the broker accepted or rejected your update on *…/update/accepted* or *…/update/rejected*

Once you see that the broker is accepting your updates, go to your *thing* and click on the Shadow. You will then see the data that is published by your *thing* in real time.

Getting a new set temperature from the Cloud

In addition to sending the actual and set temperatures to the cloud, you should also be able to change the set temperature from the Cloud and have it update on the thermostat.

If you are using MQTT, use the subscriber project as a reference. Many functions are common between the publisher and subscriber so you will not need to duplicate those.

If you are using HTTP, you will use a GET request to get the data required. In this case, you will need to poll occasionally for data since there is no concept of subscription in HTTP.

8.3 Example Firmware Architecture

For those having trouble getting started, we have prepared an example architecture of one way to approach this problem. To maintain modularity and reduce complexity it is HIGHLY RECOMMENED that you add additional functionality in new RTOS tasks. For example, you may want separate the functionality as follows:

1. main: get everything up and going:
    a. Initialize the board and peripherals
    b. Create a main IoT task, semaphores, queues, mutexes, timers
    c. Start the IoT task
2. mqtt_task:
    a. Connect to Wi-Fi
    b. Connect to the broker (MQTT)
    c. Create and start all other tasks
    d. Handle IoT error events
3. subscriber_task: handle MQTT subscriptions and messages
4. publisher_task: handle MQTT publishing
5. pot_task: Read data from the potentiometer and convert to simulated actual temperature
6. capsense_task: Monitor CapSense buttons.
7. display_task: Update the TFT display.

Interaction between the tasks can (and should!) be controlled using notifications, semaphores, queues, and mutexes.

A pictorial representation of the architecture described above is shown here: