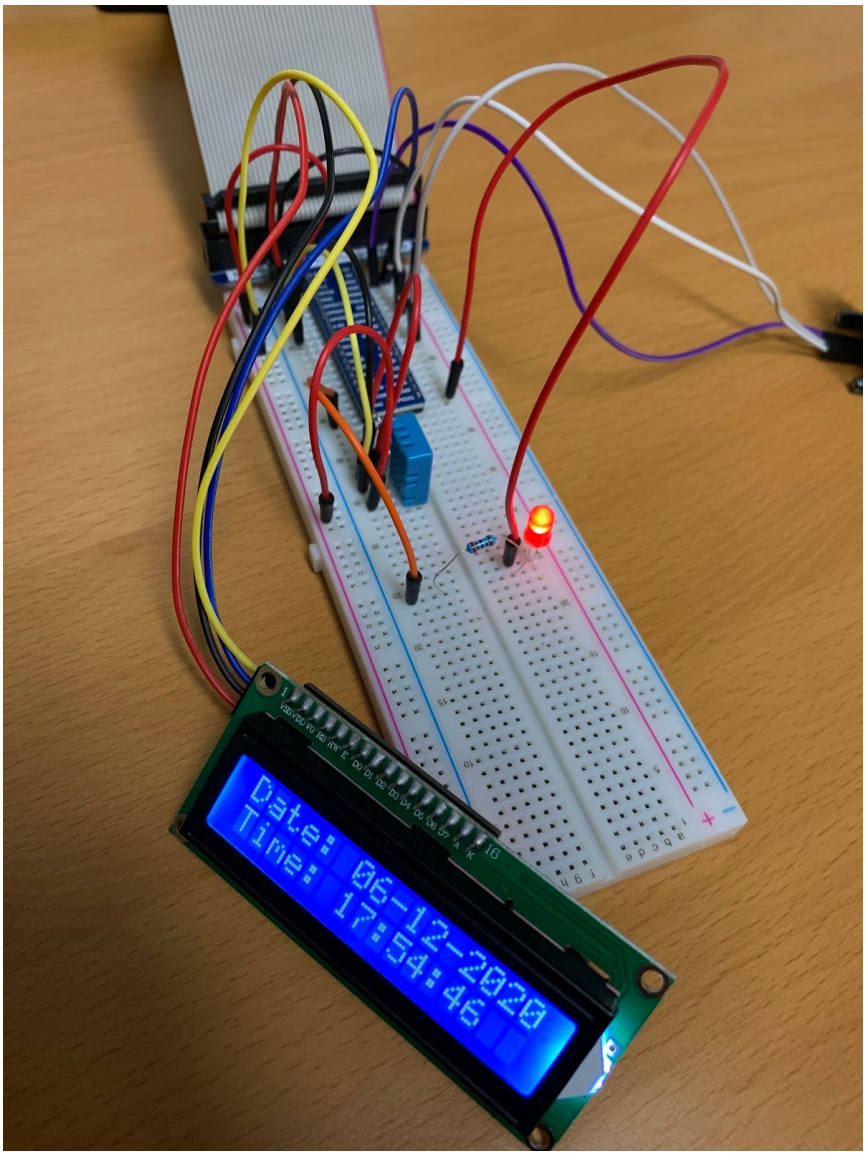
EECS 113: Final Project

Atmosphere Monitoring System



Johnathan Tang

63061294

EECS 113 : Spring 2020

INTRODUCTION

For this project we are developing a Raspberry Pi based IoT device to mimic a weather monitoring system. We will be using a DHT-11 sensor to check local temperature and humidity once per minute for 24 hours. After every hour, we will also need to pull data from CIMIS to retrieve weather information for the city of Irvine. The data used in this system will decide how much water is necessary to irrigate and toggle a virtual sprinkler system. Our system will simulate irrigation once per hour and turn back off after the amount of time needed to deliver the necessary amount of water. For our purposes we will assume that the irrigation system delivers 1020 gallons of water per hour across 200 square feet of lawn.

MATERIALS

1. Raspberry Pi 3
2. DHT-11 Temperature and Humidity Sensor
3. Breadboard
4. PIR Sensor
5. I2C LCD 1602
6. Jump Wires
7. 10kΩ and 220kΩ Resistor
8. GPIO Extension Board and Cable
9. LED

SETUP

For our setup we will use 3 main components on our breadboard. The DHT-11 module will help us capture local temperature and humidity, we will assign this to GPIO 17. We will also use a PIR sensor to detect motion when irrigation is in process such that if movement is detected, irrigation stops. Finally, we will use a I2C LED display to help us monitor time and data we retrieve from both the DHT and CIMIS. The entire setup is fairly inquisitive with the exception of the DHT. We wired the first leg to a 5 volt voltage source, second leg in parallel with the 10kΩ, third vacant, and fourth to our GPIO connector (GPIO 17). I also used an LED to help indicate motion detection from the PIR sensor such that if motion is detected during irrigation, the LED will shut off - LED is set default active. I wired the LED to GPIO 21.

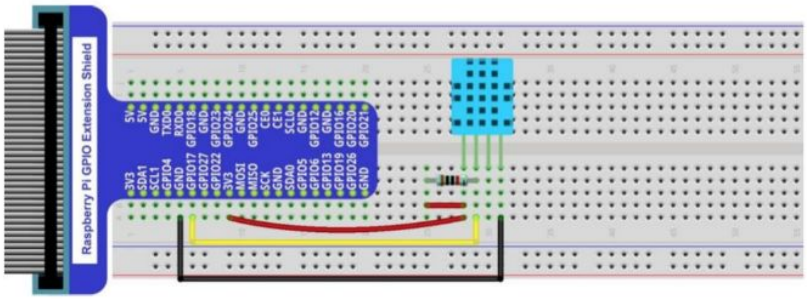


Figure 1: DHT-11 with 10kΩ resistor. [GPIO 17]

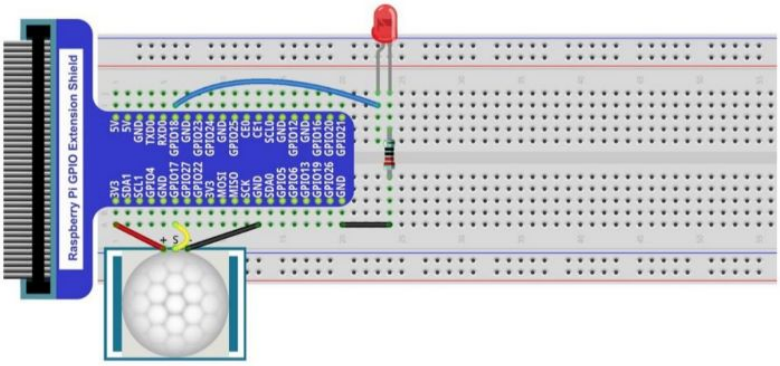


Figure 2: PIR Sensor [GPIO 25] *5V instead of 3.3V
LED set to GPIO 21

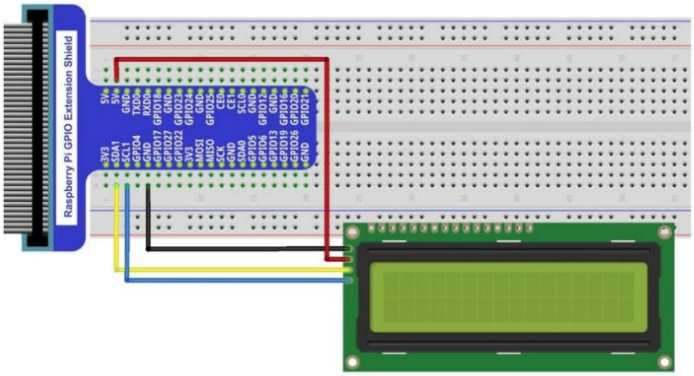


Figure 3: I2C LCD

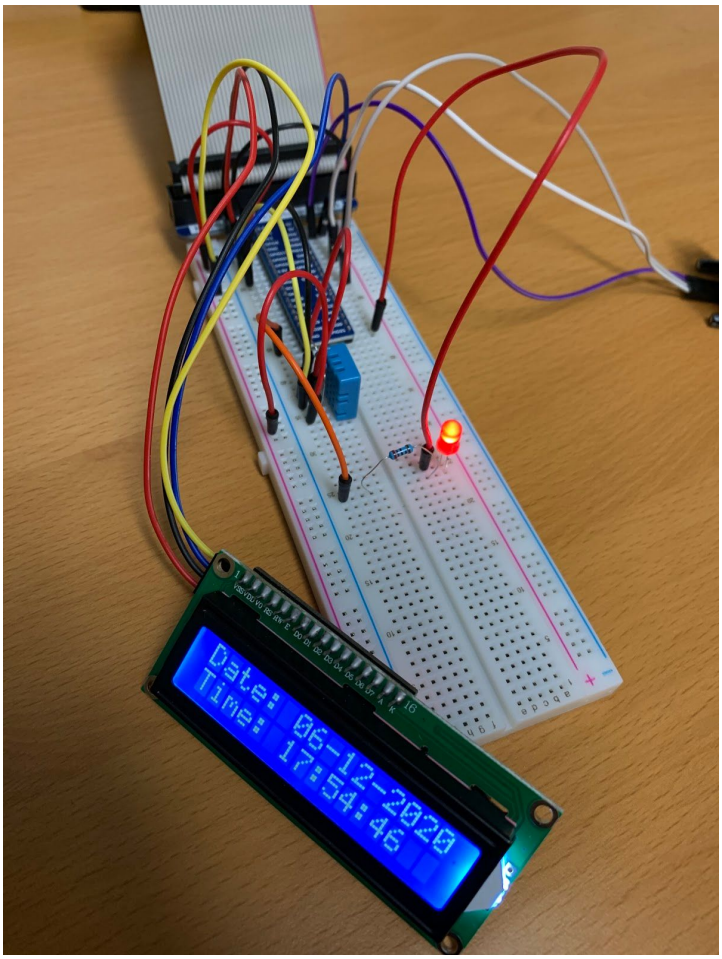


Figure 4: Breadboard configuration

SOURCE CODE

- **main.py**
- **CIMIS.py**
- **LCD.py**
- Adafruit_LCD1602.py
- Freenove_DHT.py
- PCF8574.py

Adafruit_LCD1602.py and **PCF8574.py** were obtained from the Freenove tutorial to drive the I2C LCD display we used to display weather information. **Freenove_DHT.py** is used with the DHT temperature and humidity sensor, this code is also retrieved from the Freenove tutorial.

I included snippets of my code for reference in the event that further explanation is necessary. I also left extensive comments and explanations in the source code.

main.py

main.py is the main, central driver for the entire program as it delegates and operates functions from **CIMIS.py** and **LCD.py.** One of the key role of **main.py** is to handle threads:

1. Local/DHT data
2. CIMIS data
3. Time
4. Update LCD
5. Irrigation / PIR Sensor

It uses the time of program execution as a reference point for the essential functions listed below:

Key Functions

- data_thread()
- mainloop()
- irrigation_calculation()

def data_thread():

```
while ( hours_passed < TOTAL_HOURS ):
    console_data("\n\n******************* LOCAL DHT REPORT ***********************************\n")
    #Collect one hour worth of data
    # "i" represents the index, for our purposes this is the minute
    for i in range (0,ONE_HOUR,1):
        #Get time of pulling data
        pull_time = time.time()
        #Get current local temperature
        local_temp = get_local_temperature()
        #Get current local humidity
        local_humidity = get_local_humidity()


        #Accumulate and store local temperature and humidity
        accum_temp = accum_temp + local_temp
        accum_humidity = accum_humidity + local_humidity

        #Print local data to console
        console_data("  DHT Data #"+ str(i+1) + ":        Temp:" +  str(local_temp) + "       Humidity:" + str(local_humidity))
        #Print local data to LCD
        LCD.display_local_data(i+1, local_temp, local_humidity)

        pull_time = time.time() - pull_time
        #Sleep for one minute
        if i is not (ONE_HOUR - 1):
            time.sleep ( ONE_HOUR - pull_time )


    average_temp = calculate_avg(accum_temp)
    average_humidity = calculate_avg(accum_humidity)

    #After hour has passed, record and store data to arrays
    #We also want to prompt console and LCD that hour has passed
    console_data("\n\n******************* ONE HOUR REPORT ***********************************")
    print("Averages:")
    print("Temperature: " + str(round(average_temp,1)) + " , Humidity: " + str(round(average_humidity)) )
    LCD.display_average_data(current_time , average_temp, average_humidity)

    #Store average temperature and humidity to array to tabulate data
    temperature_array[current_time ] = average_temp
    humidity_array[current_time ] = average_humidity

    #To prep for next hour, stall for one minute since we only record data
    #in increments of 1 minute
    time.sleep ( ONE_HOUR - pull_time )
    current_time  = (current_time  +1) % TOTAL_HOURS
    #Increment hours_passed, loop again
    hours_passed = hours_passed + 1

    #Reset Values
    average_temp = 0
    average_humidity = 0
    accum_temp = 0
    accum_humidity = 0
```

Figure 5: Local data handling from DHT

The purpose of this **while** loop is to, once per minute, read and display local temperature and humidity to both the console and LCD display. To index the amount of time that has passed, I used **hours_passed** to index and increment until 24 hours has passed. This essentially works as a double for loop such that the inner loop increments per minute

while the outer loop increments once an hour. Within the inner loop, the program grabs the local temperature and humidity using **get_local_temperature()** and **get_local_humidity()**, which uses the API from Freenove_DHT.py. Since we want to eventually calculate the average temperature and humidity, I used **accum_temp/accum_humidity** to later calculate the average by dividing that value by minutes in an hour. For every hour passed, we want to calculate and report the average temperature and humidity values we received from the DHT. Finally, before looping back to the inner loop, we want to reset average and accumulated weather values as well as increment **hours_passed** per cycle.

def mainloop():

```
while ( hours_passed < TOTAL_HOURS ):
    delay_hour = time.time()
    #Pull CIMIS data every hour

    console_data("\n****************** CIMIS REPORT **********************************")
    print("Pulling CIMIS Data. Hour: " + str(current_hour)+"......")

    #Retrieve the cimis data for the past hour
    data = get_cimis_data_for(current_hour )

    #Check to see if there is an error pulling data from CIMIS
    #Data could either not be availible or there was an error pulling
    while( data is None or data.get_humidity() is None or data.get_temperature() is None ):
        if data is None:
            console_main("[ERROR]Failed to pull from CIMIS. Reattempt in one hour... ")
        else:
            console_main("[ERROR]CIMIS Data Not Availible. Try Again in One Hour")
        #wait for next hour before pulling from CIMIS again, for that hour
        time.sleep( HOUR )

        #Repull data
        console_main("Repull for Hour: " + str(current_hour))
        data = get_cimis_data_for( current_hour )

    #If data pulls successfully, print and store

    print("CIMIS Data for Hour: " + str(current_hour))
    #Print CIMIS data
    cimis_date = data.get_date()
    cimis_hour = data.get_hour()
    cimis_humidity = data.get_humidity()
    cimis_temperature = data.get_temperature()
    cimis_eto = data.get_eto()

    #Print CIMIS to console
    print("Date:"+ cimis_date+ " \nTemperature: "+ cimis_temperature + " \nHumidity:"+ cimis_humidity+ " \nETO:"+ cimis_eto)

    #Print CIMIS to display : Hour, Temp, Humidity
    LCD.display_cimis_data(current_hour, data.get_temperature(), data.get_humidity())


    #Access temperature and humidity array to grab data
    #We can use current_hour as our index to access approriate data
    dht_temp = temperature_array[current_hour]
    dht_humidity = humidity_array[current_hour]


    #Clear data from the index so data doesn't overlap
    temperature_array[current_hour] = None
    humidity_array[current_hour] = None
```

Figure 6: CIMIS Data Handling

The main purpose of **def_mainloop()** includes a per hour loop, using the same index **hours_passed** from the local data handling function. With every hour passed we want to pull weather data from the CIMIS, specifically the Irvine station. If program fails to pull data or if there is something wrong on the other end, this thread will sleep for an hour so

it is in sync with the hourly CIMIS report, to do this we use **time.sleep(HOUR) .** Once an hour has passed, we re-pull from the same hour we missed and if successful, store and print data from CIMIS. In addition to temperature and humidity levels, we also grab ETO to later determine if irrigation is necessary. Since we are considering hours that we missed CIMIS data, we can access previous DHT values from **temperature_array** and **humidity_array** so we can compare relative values for the same time of day.

```
#If irrigation calculation returns 0, no need to irrigate
if irrigation_time != 0 :
    GPIO.output(led_pin, GPIO.HIGH)
    console_data("\n\n****************** IRRIGATION REPORT **********************************")
    console_main("[Irrigation]Irrigate for " + str(irrigation_time) + " seconds")

    #Set start and stop time for irrigation
    start_time = time.time()
    stop_time = 0
    total_time = start_time + irrigation_time + stop_time

    #Run irrigation till either PIR sensor detects motion
    #or if timer runs out (total time)
    while( time.time() < total_time ):
        #if PIR sensor detects motion, stop irrigation immediatly
        #otherwise irrigate until timer stops
        if( GPIO.input(pir_pin) == GPIO.HIGH  ):
            if(stop_time < 60):
                GPIO.output(led_pin, GPIO.LOW)
                console_main("[Motion Detected]Irrigation Stopped")
                time.sleep(10)
                stop_time = stop_time + 10
#If no irrigation required
else:
    console_main("[No irrigation required]\n\n")
```

Figure 7: Irrigation Thread if Required

Since irrigation process is only necessary based on calculations from **Appendix B: Irrigation**. If **irrigation_calculator()** returns 0, that indicates that irrigation is not necessary, therefore the program will continue its DHT data routine. However, if **irrigation_calculator()** returns a value, we mimic a virtual irrigation by setting a timer based on that value. There are two scenarios where irrigation will halt, that is if either **irrigation_time** is reached or if motion is detected from the PIR sensor**.** In the event that **GPIO.input(pir_pin)** == **GPIO.HIGH**, we immediately stop irrigation and loop back to DHT routine. Since we want to keep our threads in sync, we want to make sure that irrigation takes no longer than 60 seconds. Once irrigation is done, prompt the user before returning to the local DHT thread. As a bonus, I also implemented an LED such that if motion is detected, LED will shut off.

```
#returns the time to irrigate in seconds based on the cimis data and local average temperature and humidity
def irrigation_calculation(data, local_temp, local_humidity):
    #Declare constants to calculate appropriate time for irrigation
    #These include plant factor, area of irrigation, irrigation efficiency, and water debit per hour
    #These values are given per instruction PDF
    plant_factor = 1.0
    area_irrigate = 200
    irrigation_eff = 0.75
    water_debit = 1020
    conversion_constant = 0.62

    #Modify ETO for our purposes
    eto_humiditiy = float(data.get_humidity()) / local_humidity
    eto_temp = local_temp / float(data.get_temperature())
    modified_eto = float(data.get_eto()) * eto_humiditiy * eto_temp

    #Calculate how many gallon needed per hour for irrigation
    num_gallons = modified_eto * plant_factor * area_irrigate * conversion_constant / irrigation_eff
    gallon_per_hour = num_gallons / float(24)
    #This will be the time needed to fully irrigate
    irrigation_time = gallon_per_hour / water_debit

    return irrigation_time * HOUR
```

Figure 8: Irrigation Calculation

Given the constant values and equation from **Appendix B: Irrigation**, this function calculates the amount of time needed to irrigate. For our temperature and humidity values, we used the data collected from both the CIMIS and local DHT.

CIMIS.py

This module serves as our local CIMIS API to retrieve data from the CIMIS - Irvine Station. To do so, we need to address **et.water.ca.gov** as well as the specific station number, for our purposes Irvine station is number 75. For this module I decided to create a **cimis_data** class to better store weather information - temperature, humidity, and eto. This allows for ease of access such that if I want to grab the humidity for a given hour I can call **get_humidity()** which will then return appropriate data from the CIMIS. The heart of this API lies in **cimis_hour(current_hour)** which is shown below:

```
#This will be called from the main function to index the current hour that
#we need to use to get CIMIS data from. Current_hour is the same index during runtime
#that we used in the main program. Since we are within 24 Hour formatting, there is a special case
#when current_hour = 0. In this case it is midnight so we want to grab CIMIS data from the previous day.
#If this isn't the case we grab data from time as is.
def cimis_hour (current_hour):
    #If current_hour indicates midnight, we want to grab data from the previous day
    #datetime.now() - timedelta(1) handles this case
    if (current_hour == 0 ):
        date = datetime.strftime(datetime.now() - timedelta(1), '%Y-%m-%d')
        if(time.localtime(time.time()).tm_hour < current_hour):
            date = datetime.strftime(datetime.now() - timedelta(1), '%Y-%m-%d')

    #else we use current time as index
    else:
        date = datetime.now().strftime('%Y-%m-%d')

    #call cimis_url to get data from url parsed for cimis_url
    data = cimis_url(appKey, irvine_station , date, date)
    #if data received is null or None, return None and main function will prompt
    #user that something has gone wrong or data is not avalible
    if data is None:
        return None
    #else return data received from CIMIS using 3D array which indexes the hour of retrieval,
    #category of data, and the actual value itself
    #GlyRelHum, HlyAirTmp, HlyEto are protocols used to address data in CIMIS
    else:
        data_received = cimis_data(
                        data[current_hour-1]['HlyRelHum']['Value'],
                        data[current_hour-1]['HlyAirTmp']['Value'],
                        data[current_hour-1]['HlyEto']['Value']
                       )
        #return data to main function
        return data_received

#This function pings and sends a request to et.water.ca.gov
#since we want our request to be dynamic such that we only want
#certain data from a specific time frame we need to parse our own url
#once url is generated, send request and listen for response
#if sucessfull, extract and return data, else simply return none and main funciton
#will prompt user accordingly
def cimis_url(appKey, irvine_station , start, end):
    request_data = ['hly-air-tmp',
                    'hly-eto',
                    'hly-rel-hum']

    dataItems = ','.join(request_data)

    url = ('http://et.water.ca.gov/api/data?appKey=' + appKey + '&targets='
           + str(irvine_station ) + '&startDate=' + start + '&endDate=' + end +
           '&dataItems=' + dataItems +'&unitOfMeasure=M')
    try:
        content = urlopen(url).read().decode('utf-8')
        data = json.loads(content)
    except:
        print("CIMIS Request Failed....")
        data = None

    if(data is None):
        return None
    else:
        return data['Data']['Providers'][0]['Records']
```

Figure 9: cimis_hour() function to retrieve data from et.water.ca.gov API

cimis_hour(hour) takes a given hour as an argument before delegating the appropriate time to request data from CIMIS. **request_data[]** holds the type of information we want to request from the CIMIS using their protocol notation **hly-air-tmp, hly-eto,** and **hly-rel-hum**. We then want to join that array list with the **dataItems** protocol which we then used in our **cimis_url** to pull data. Once we have our URL correctly parsed with the information we need, we can use that URL to pull data and store them into an array **data_receieved**. Data_received is a 3D array which indexes the hour of data, type of data, and data itself that we received from the CIMIS. This array gets returned to the main function to calculate irrigation time and to display on the LCD.

LCD.py

LCD.py is used to delegate what is displayed on the I2C LCD during runtime. Key functions include setting up the LCD itself - PCF8574 addressing, LCD backlight, dimension initialization, and displaying information from the main program.

```
#LCD Thread
#For most of the runtime we want to constantly display current time
#To do this we check if there is any incomin display request from main program,
#stop displaying current time for 3 seconds, during that 3 seconds we want to display
#the message from main.

def lcd_thread():
    global incoming_message
    while not on_off_toggle:
        lcd.clear()
        lcd.setCursor(0, 0)
        if(incoming_message is not None):
            lcd.message(incoming_message)
            incoming_message = None
            sleep(3)
        #Per second update to LCD to replicate live time update. Therefore, we want to delay the display
        #update once per second to replicate a real timer that increments by seconds
        #First line of LCD shows date, which only updates day every 24 hours
        #Second line of LCD shows current time, which updates once per second
        else:
            lcd.message(datetime.now().strftime('Date: %m-%d-%Y') + "\n" + datetime.now().strftime('Time: %H:%M:%S'))
            sleep(1)

#This function facilitates what gets printed to the LCD based on the message we
#receive from main program. "Type" refers to the source of data (i.e DHT or CIMIS)
def display_data(counter, temperature, humidity, type) :
    global incoming_message
    message = type + " Data #"+ str(counter) + "\nT:" + str(temperature) + " H:" + str(humidity)
    while(incoming_message is not None):
        sleep(1)
    incoming_message = message
```

Figure 10: lcd_thread() to handle LCD updates and incoming messages from main

lcd_thread(): constantly displays the current date and time of runtime unless there is a **display_request** - DHT update or CIMIS update.

The display will only have 4 states:

1. Display current date and time (updated every during runtime)
2. Display per minute update from DHT
3. Display per hour update from DHT (average values)
4. Display per hour update from CIMIS

Figure 10: Display current Date and Time (Time is in 24 Hour format)

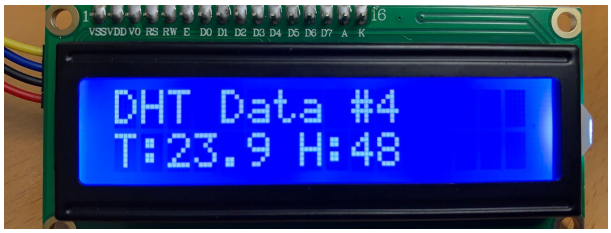


Figure 11: Display per minute temperature and humidity update from DHT (#4 indicates the 4th minute of the hour interval)

Figure 12: Display per hour average temperature and humidity

Figure 13: Display per hour data from CIMIS - Irvine Station

DATA

Since data is redundant for the 24 hours of runtime, here is a sample output from console:

```
******************** LOCAL DHT REPORT ************************************

[14:37]  DHT Data #1:        Temp:22.4       Humidity:48
[14:38]  DHT Data #2:        Temp:11.1       Humidity:54
[14:39]  DHT Data #3:        Temp:22.5       Humidity:48
[14:40]  DHT Data #4:        Temp:22.4       Humidity:48
[14:41]  DHT Data #5:        Temp:22.4       Humidity:54
.
.
.
[15:31]  DHT Data #55:        Temp:22.4       Humidity:54
[15:32]  DHT Data #56:        Temp:22.4       Humidity:54
[15:33]  DHT Data #57:        Temp:22.4       Humidity:55
[15:34]  DHT Data #58:        Temp:22.5       Humidity:48
[15:35]  DHT Data #59:        Temp:22.3       Humidity:48
[15:36]  DHT Data #60:        Temp:22.4       Humidity:48
[15:36]

******************** ONE HOUR REPORT ************************************
Averages:
Temperature: 20.5 , Humidity: 51
[15:37]
******************** CIMIS REPORT ************************************
Pulling CIMIS Data. Hour: 14......
CIMIS Data for Hour: 14
Date:2020-06-11
Temperature: 31.6
Humidity:18
ETO:0.84
[15:37]

******************** IRRIGATION REPORT ************************************
[15:37][Irrigation]Irrigate for 4.640263760346885 seconds
[15:37]
[15:37][Irrigation]Irrigation Done
```

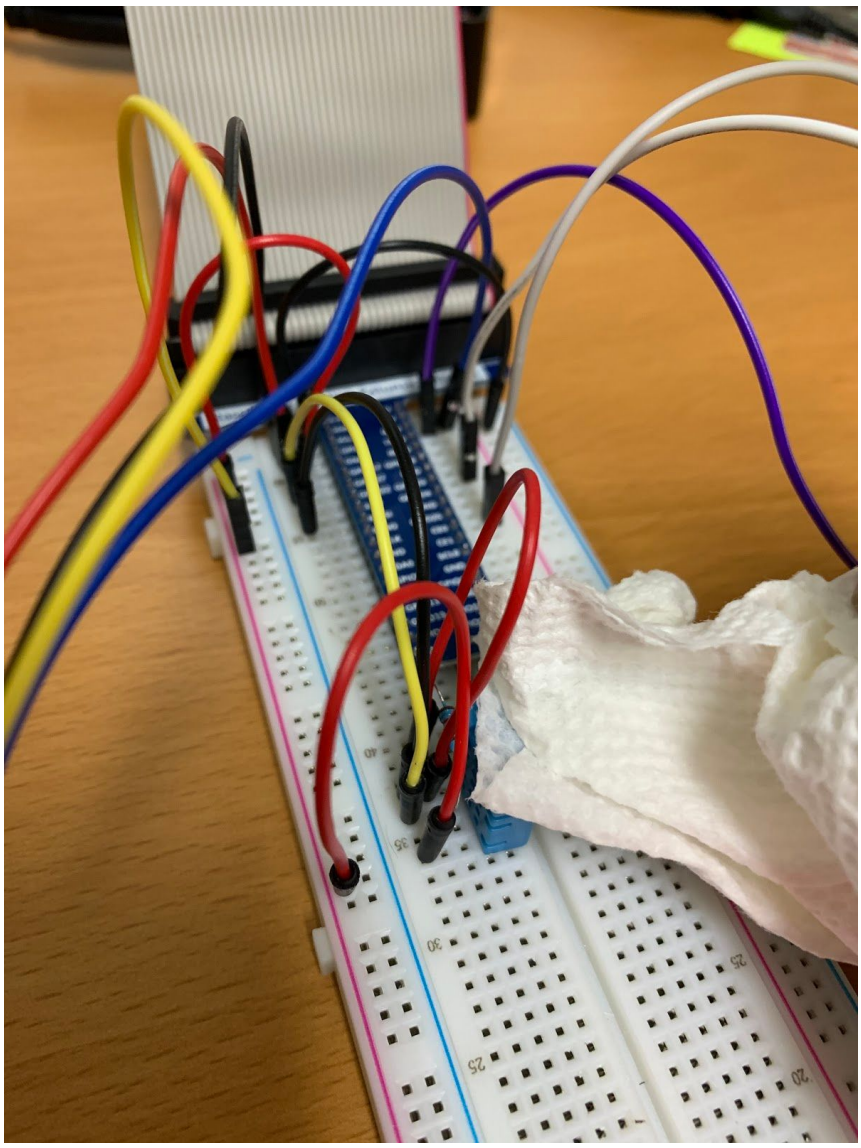
Figure 12: Sample Data of per minute and per hour update to console

To save space, I replaced data from minutes 6 - 54 with "…" however the data is still stored in the attached **output.txt** file. As you can see, we print local DHT statistics per minute. Once an hour has passed, we accumulate the data collected to calculate the average temperature and humidity for that hour interval. We then do our hourly pull from the CIMIS and compare their data with our DHT. Based on calculations the program will then determine if irrigation is necessary. If so, irrigation stalls the program for the appropriate amount of time before continuing to the next cycle of identical operations.

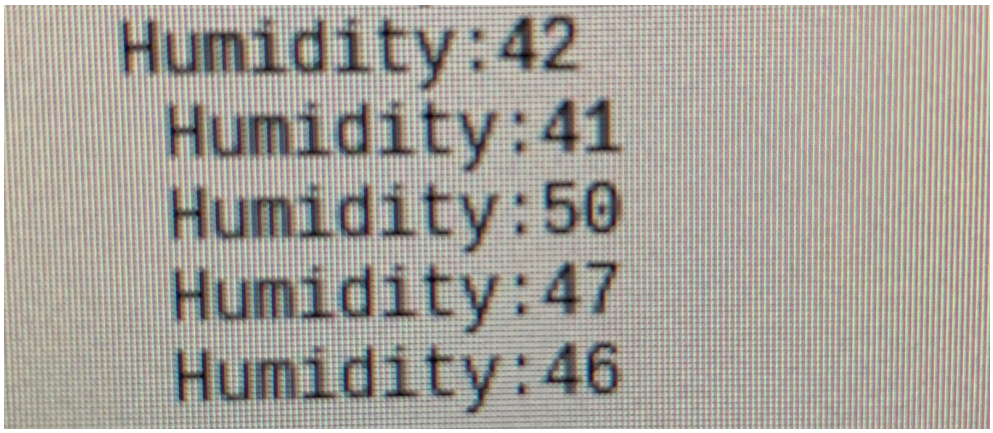
TEST CASES

To test the accuracy and responsiveness of both the DHT-11 and PIR sensor, I carried out three test cases:

1. Lay a damp paper towel gently on the DHT sensor to test humidity:



Waiting for the next per minute cycle, the humidity reading spiked up to 9 units above the normal, as shown:



2. To test accurate temperature reading, I used the same technique such that I used a refrigerated water bottle and placed it gently on top of the DHT sensor.

3. To demonstrate that the PIR sensor works, such that it stops irrigation upon detecting movement, I created a special case where irrigation time was for about 15 seconds. Once PIR senses motion, the irrigation should stop and the program should continue on with it's process. This will be demonstrated in the video attached along with my submission files.

Deliverables:

Included in the submission zip file:

- Source Code Folder
    - Main.py
    - CIMIS.py
    - LCD.py
    - Adafruit_LCD1602.py
    - Freenove_DHT.py
    - PCF8574.py
- 113_Project_Report.pdf
- output.txt (includes captured data of 24 hour span)
- FinalProjectDemo.mp4