

# PROJECT A - ENVIRONMENT LOGGER

Ronak Mehta<sup>†</sup> and Vikyle Naidoo<sup>‡</sup>  
EEE3096S Embedded Systems II, 2019  
University of Cape Town  
South Africa  
<sup>†</sup>MHTRON001 <sup>‡</sup>NDXVIK005

**Abstract**—This project was aimed at creating and deploying an IOT device with the intention of developing a representative embedded system. The project objective was to design and build an environment logger for monitoring a private greenhouse through using a Raspberry Pi and other sensing modules.

## I. INTRODUCTION

An environment logger is an embedded system device which interacts with its surroundings (a greenhouse for this project) by measuring a number of factors ranging from moisture content to GPS location. For this project, the sensing factors were light intensity, temperature and humidity readings and a constant monitoring of the time of day was also conducted. This report explains the design choices made for the project starting from the sensor types, to the use of a Real Time Clock (RTC), an Analog to Digital Converter (ADC), Digital to Analog Converter (DAC), Blynk Application and the output for alarm notification.

The project used an LDR to detect the light levels, a MCP9700A temperature sensor for measuring temperature and a potentiometer to mimic the humidity VPD sensor which outputs an analog voltage between 0V and 3.3V. A MCP3008 ADC was used to convert analog readings from the three sensors into digital for it to be interpreted by the RPi. It also made use of MCP7940M RTC to record time as it is often hard to hold the system time correctly in a Raspberry Pi because the Pi uses Network Time Protocol (NTP) which needs constant internet connectivity to hold time. Thus, the project interfaced with the RTC using I2C communication bus and set the time using buttons and interrupts.

The calculated output voltage from the RPi was then connected to a DAC so that an analog value is being read at the output. An output LED was also added using a PWM signal to notify if the output voltage went below 0.65V or above 2.65V. These output readings notifying an alarm going off alongside sensor values from the ADC were remotely monitored and displayed on the Blynk Application (an IOT platform which provides graphical interface to remotely control hardware.)

This report then describes the UML Use Case diagram, a State Chart diagram together with a UML Class diagram and a circuit diagram for indicating the structuring of different modules in the system. The report then discusses code snippets, provides validation and performance of the system and summarizes the success of the environment logger project.

## II. REQUIREMENTS

As seen from Figure 5 below which represents the UML Use Case diagram, the core requirements for the system were:

1. Four push buttons each with different functionality.
  - a. To start or stop the monitoring of sensors without affecting the system timer.
  - b. To dismiss the alarm when the output voltage goes below 0.65V or above 2.65V
  - c. To reset the system and clean the console. This button press was further extended to affect the RTC module which was responsible for resetting the system time.
  - d. To change the frequency of monitoring. The frequencies used in this project were 1s, 2s and 5s and looped between these values per event occurrence. This alteration is further extended to the RTC module.

Interrupts were also created for all four button functionalities and the inputs were debounced.

2. The MCP3008 ADC took in three analog readings from the three sensors and output a digital voltage to the RPi. The value read from the temperature sensor was then converted to degrees Celsius using the formula found in the temperature sensor datasheet [5]:

$$Ta = (V_{out} - V_{oc})/Tc \quad (1)$$

Where:

Ta = Ambient temperature in celsius

Vout = Sensor output voltage

Voc = Sensor output voltage at 0 degrees celsius (0.5V)

Tc = Temperature coefficient (10mV/degree celsius)

3. The MCP7940M RTC read time with an oscillator crystal frequency of 32.768kHz and then wrote it to the RPi using the I2C serial communication bus.
4. Raspberry Pi 3B+ was the central connecting device in this system. It took in inputs from the push buttons, the ADC and RTC and output a correctly calculated voltage over the DAC using the formula:

$$V_{out} = (LightReading)/1023 * (HumidityVoltageReading) \quad (2)$$

5. The MCP4911 DAC converted the digital output from RPi into analog. Here the analog output voltage could be of one of the two ranges. Nothing happened if the output was in the range between 0.65V and 2.65V but as soon as it went beyond or above this specified range, then an LED flashed 'ON' to indicate the user for an issue.
6. The output LED was used as an indication whenever the voltage went beyond the range mentioned above and it flashed an LED 'ON' using a PWM signal. The LED however only turned ON if the previous alarm dismissal was more than 3 minutes ago; and this three minute timer started when the alarm was ON and not when it was dismissed.
7. The Blynk App provided remote monitoring of the system and also displayed the ADC readings from all the three sensors, the system time, and state of the alarm (if the LED had gone on or off)

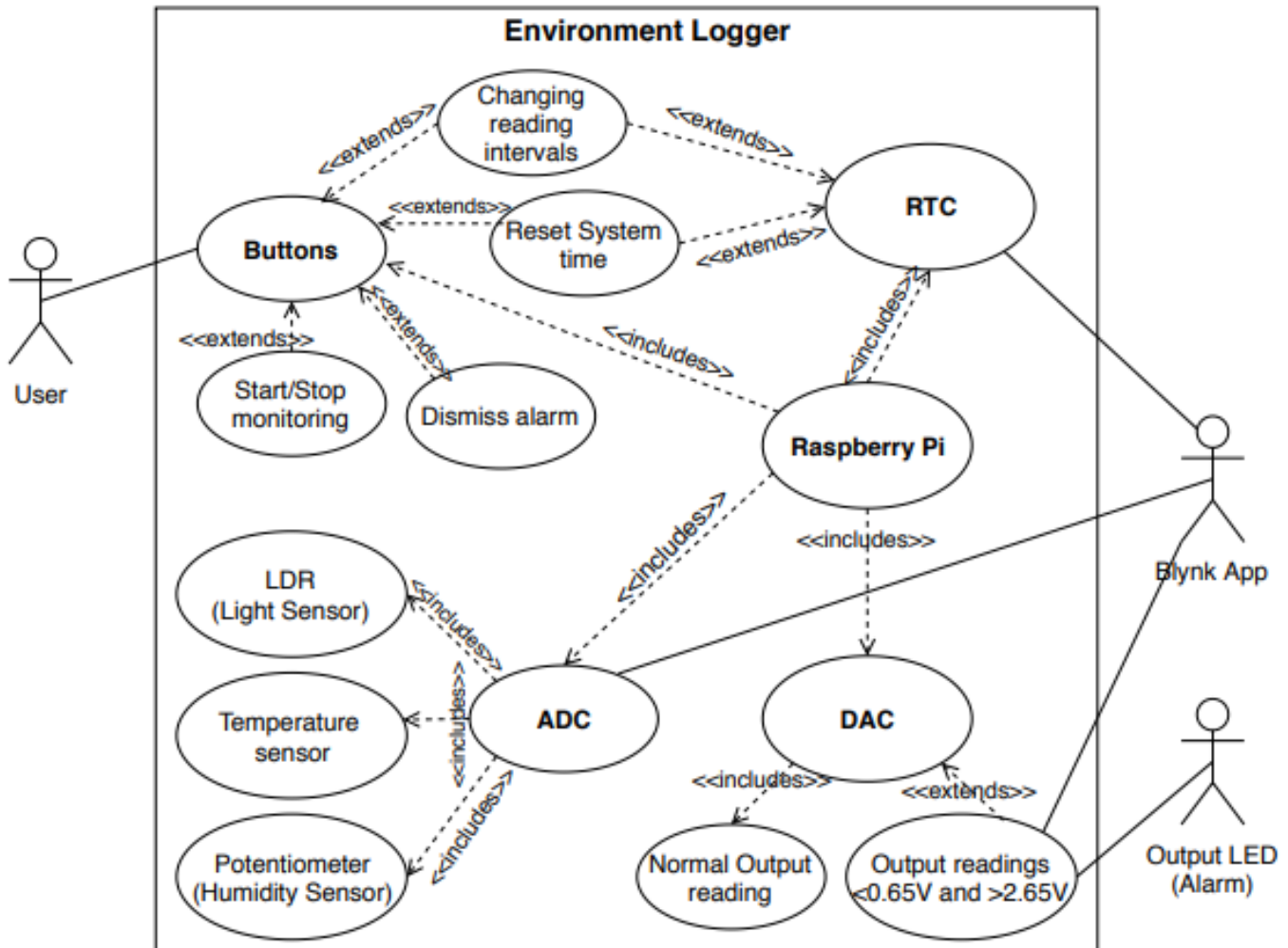


Fig. 1: UML Use Case diagram of the system

### III. SPECIFICATION AND DESIGN

For the design of the system, Figure 2 which displays a UML State Chart describes all the main operations of the Environment Logger with trigger events and guard conditions.

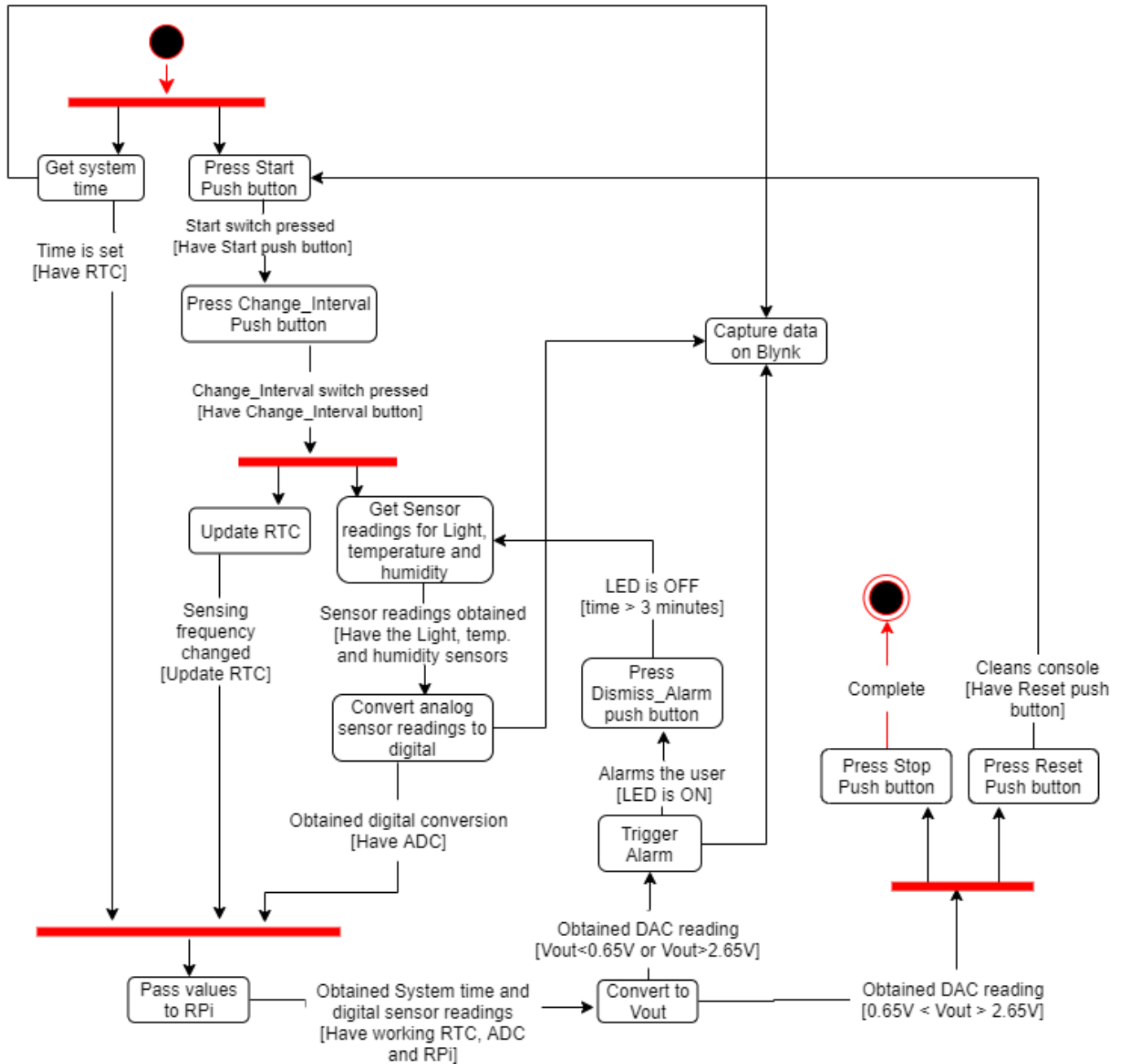


Fig. 2: UML State Chart of the system

Figure 3 below shows the UML Class diagram indicating the structuring of the code implementation alongside the different class modules used with their functions.

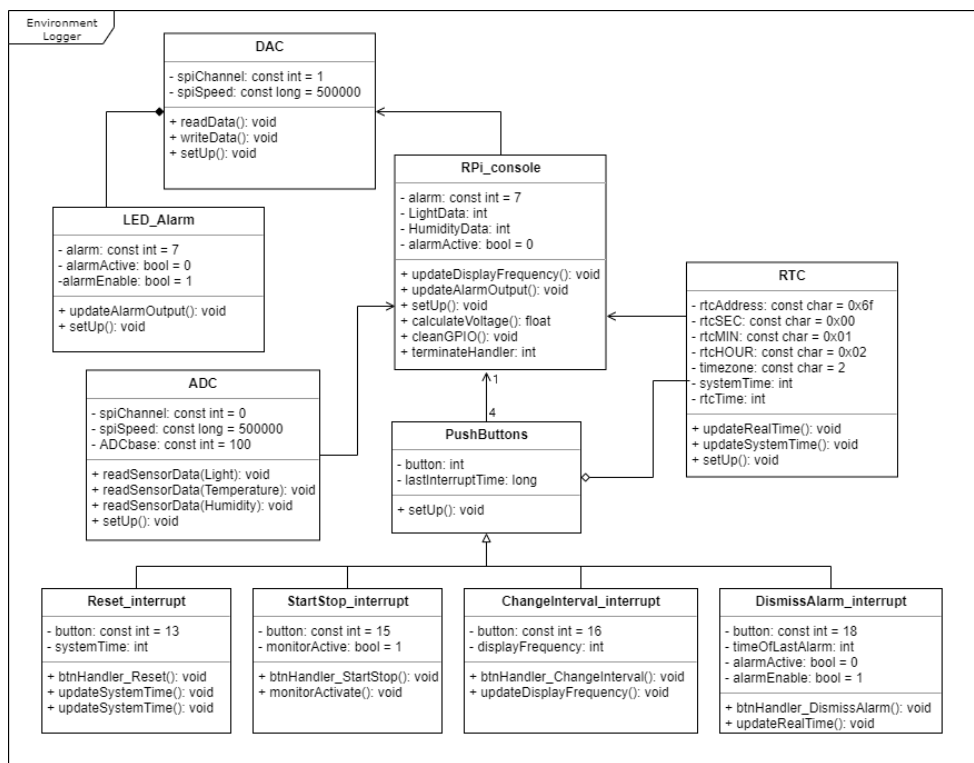


Fig. 3: UML Class Diagram indicating structuring of the code modules

Figure 4 shows the Circuit Diagram and the connections between the different components making up the larger system.

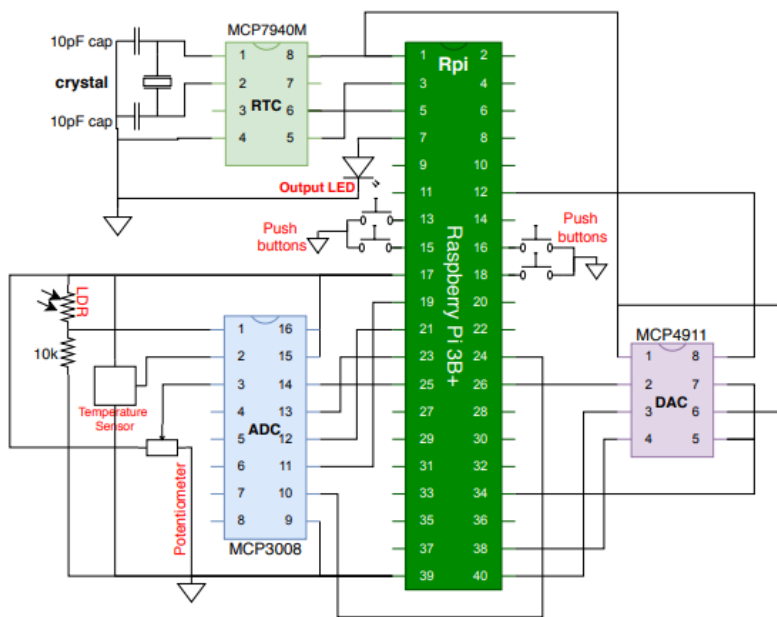


Fig. 4: Circuit Diagram of the system

## IV. IMPLEMENTATION

Some of the important code snippets used for implementing the Environment Logger system are mentioned and discussed briefly in this section.

### A. I2C and SPI Communication:

Wiring Pi includes a library which makes it easier to use Raspberry Pi's on-board I2C and SPI interface.

- The SPI bus is a synchronous data bus which uses separate lines for data and clock. These lines are SCLK (Clock generated by master), MOSI (Master Out Slave In), MISO (Master In Slave Out) and !SS (Slave Select active low). An SPI Clock speed of 500kHz was used in this project. SPI bus was used to send data over the ADC and DAC Channels. To send data, it is required to use the Read/Write command which performs a simultaneous read/write transaction over the selected SPI bus.
- The I2C bus consists of two bidirectional open drain lines which are pulled up with resistors. These lines are Serial Data Line (SDA) and Serial Clock Line (SCL). I2C bus was used to send data over the RTC. This has two separate Read and Write commands unlike SPI which writes/reads data values into/from the register.

```
#include <wiringPi.h>           //To use Wiring Pi GPIO library
#include <wiringPiI2C.h>        // To use I2C library

int RTC, MM, mins;
const char RTCMIN = 0x01;      //Address of RTC minutes register
const char RTCAAddr = 0x6f;    //Hardware address of the slave

RTC = wiringPiI2CSetup(RTCAAddr); //Initialize and set up RTC
wiringPiI2CWriteReg8(RTC, RTCMIN, MM); //Writes 8-bit MM value into RTCMIN register
mins = wiringPiI2CReadReg8(RTC, RTCMIN); //Reads 8-bit value from RTCMIN register.

#include <wiringPiSPI.h>        //To use SPI library
#include <mcp3004.h>            //To use ADC

//Global constants
#define SPI_CHANNEL_ADC 0
#define SPI_CHANNEL_DAC 1
#define SPI_SPEED 500000
#define ADC_BASE 100

wiringPiSPISetup(0, 500000);    //Initialize and set up ADC channel
mcp3004Setup(ADC_BASE, SPI_CHANNEL_ADC);
wiringPiSPIDataRW(SPI_CHANNEL_DAC, outputData, 2); //Send Data over DAC channel
```

Fig. 5: Some commands indicating the initialization, set up and sending of data via I2C and SPI

### B. Main Function:

The main function as seen from Figure 7 below, started by initializing and set up variables and constants and also created a thread for reading from the ADC (shown in Figure 6). It then called up the functions **updateRealTime()** and **updateSystemTime()** to update Real time and system time respectively when the monitoring was Active (a boolean 1). It then calculated Vout to determine if a call to trigger the alarm was necessary or not depending on the DAC Vout voltage. If Vout was less than 0.65V or more than 2.65V then and if three minutes had passed since the last alarm, then the alarm would be active and an update would be made to the alarm output and real time. It also displayed the RTC time, the system time, DAC output, Alarm detection and temperature, light and humidity readings in the terminal.

```
thread init_ADC(void){
    // wiringPiSPISetup(SPI_CHANNEL, SPI_SPEED);

    wiringPiSPISetup(0, 500000);
    mcp3004Setup(ADC_BASE, SPI_CHANNEL_ADC);
    std::thread thread_ADC(readADC2); // run read_ADC() on a new thread
    cout<<"init_ADC done, Thread started\n";
    return thread_ADC;
}
```

Fig. 6: Thread function for reading from ADC

```

int main(void){
    //setup and initialisation
    wiringPiSetup ();
    init_GPIO ();
    thread thread_ADC = init_ADC ();

    Sleep (1);
    btnHandler_Reset ();

    while(true) {
        while(monitorActive) {
            std::this_thread::sleep_for (std::chrono::milliseconds(displayFrequency*1000)) //monitor frequency

            float humidity = calculateVoltage (humidityData, 3.3);
            //temp: Vout = Tc-Ta + V0 = 10m*Ta+0.5 ==> Ta = (Vout-V0)/Tc
            float temp = (calculateVoltage (temperatureData, 3.3)-0.5)/10;
            //calculate DAC voltage
            float vout = (lightData/1023) *humidity;

            //display readings
            cout<<"RTC Time = "<<rtcTime [2] <<":"<<rtcTime [1] <<":"<<rtcTime [0] <<"\t";
            cout<<"System Time = "<<systemTime [2] <<":"<<systemTime [1] <<":"<<systemTime [0] <<"\t";

            alarmActive = 0; //switch alarm off
            if (vout<0.65 || vout>2.65) {
                if(alarmEnable){ // 3mins have passed since last alarm
                    alarmActive = true;
                    timeOfLastAlarm [0] = rtcTime [0];
                    timeOfLastAlarm [1] = rtcTime [1];
                    timeOfLastAlarm [2] = rtcTime [2];
                }
            }
            updateAlarmOutput ();
        }
    }
    thread_ADC.join ();
    return 0;
}

```

Fig. 7: Main function code

### C. Update Functions:

All update functions required and used in this project were:

1. Updating the monitoring frequency: Here the frequency for monitoring the sensor readings could be altered to be either at 1s, 2s or 5s with a default monitoring frequency at 1s. Code in Figure 8
2. Updating system time and real time: The Real time gets updated from the RTC using the Read command from the I2C bus and the system time gets updated based on the rtcTime and timeOfStart Code in Figure 9
3. Updating the Output LED (Alarm): The LED goes High everytime the alarm is active and vice versa. Code in Figure 10

```

void updateDisplayFrequency(void){
    switch (displayFrequency){
        case 1:
            displayFrequency = 2;
            break;
        case 2:
            displayFrequency = 5;
            break;
        case 5:
            displayFrequency = 1;
            break;
        default:
            displayFrequency = 1;
            break;
    }
}

```

Fig. 8: Code for updating the monitoring frequency

```

/*update the time from the RTC */
void updateRealTime(){
    rtcTime[0] = getSecs();//wiringPiI2CReadReg8(RTC, RTCSEC);
    rtcTime[1] = getMins();//wiringPiI2CReadReg8(RTC, RTCMIN);
    rtcTime[2] = getHours();//wiringPiI2CReadReg8(RTC, RTCHOUR);
}

/*update the system time based on the rtcTime and timeOfStart*/
void updateSystemTime(void){
    int second1 = rtcTime[0];
    int minute1 = rtcTime[1];
    int hour1 = rtcTime[2];
    int second2 = timeOfStart[0];
    int minute2 = timeOfStart[1];
    int hour2 = timeOfStart[2];

    if(second2 > second1) {
        minute1--;
        second1 += 60;
    }
    systemTime[0] = second1 - second2;

    if(minute2 > minute1) {
        hour1--;
        minute1 += 60;
    }
    systemTime[1] = minute1 - minute2;
    systemTime[2] = hour1 - hour2;
}

```

Fig. 9: Code for updating system time and real time

```

void updateAlarmOutput(void){
    if(alarmActive){
        digitalWrite(ALARM, HIGH);
        cout<<"*";
    }
    else
    {
        digitalWrite(ALARM, LOW);
    }
}

```

Fig. 10: Code for updating the Output LED (Alarm)

#### D. CleanUp Function:

The cleanupGPIO(void) function cleans up all general purpose input output pins by setting them back to a LOW value using a Pull Down resistor. The terminateHandler() function exits the console and terminates the code. This function is called up in the main loop.

```
/* cleanup the gpio pins by setting them all back to input with a LOW value*/
void cleanupGPIO(void){

    pinMode(ALARM, INPUT);
    pullupDnControl(ALARM, PUD_DOWN);

    printf("exiting gracefully\n");

    exit(0);

}

void terminateHandler(int sig_num){
    if(sig_num==SIGINT){
        cleanupGPIO();
    }
    printf("error");
    exit(0);
}
```

Fig. 11: Code for updating the Output LED (Alarm)

#### E. Interrupt Handler:

The functions for each push button to provide an interrupt were as follows:

1. The Reset button handler updates the real time and system time and sets the system time to zero and also clears the alarm output.
2. The StartStop button handler either activates the monitoring of data or stops the monitoring.
3. The ChangeInterval button handler calls the updateDisplayFrequency() to alter the frequency of monitoring
4. The DismissAlarm button handler changes alarmActive to boolean False to dismiss the alarm and updates the Real time. It also checks if the duration since the alarm went off is more or less than three minutes.

The code for each of the interrupt handler is shown in Figures 12 and 13 below.

```
void btnHandler_Reset(void){
    long now = millis();

    if(now-lastInterruptTime>200){

        updateRealTime();
        timeOfStart[0] = rtcTime[0];
        timeOfStart[1] = rtcTime[1];
        timeOfStart[2] = rtcTime[2];
        systemTime[0] = 0;
        systemTime[1] = 0;
        systemTime[2] = 0;
        updateSystemTime();
        alarmEnable = 1;
        alarmActive = 0;
        system("clear");
        cout<<"system reset\n";
        cout<<"\n\tRTC Time\tSystem Time\tHumidity\tTemperature\tLight\tDAC Out\tAlarm";
        cout <<endl;
    }
    lastInterruptTime = now;
}

void btnHandler_StartStop(void){
    long now = millis();

    if(now-lastInterruptTime>200){
        monitorActive = !monitorActive;
    }
    lastInterruptTime = now;
}
```

Fig. 12: Code for Reset and StartStop interrupt Handler



```

void btnHandler_ChangeInterval(void){
    long now = millis();

    if(now-lastInterruptTime>200){
        updateDisplayFrequency();
        cout<<"\n*display frequency changed to every "<< displayFrequency<< "seconds*\n";

    }
    lastInterruptTime = now;
}

void btnHandler_DismissAlarm(void){
    long now = millis();

    if(now-lastInterruptTime>200){
        alarmActive = false;
        cout<<"\n*Alarm dismissed*\n\n";
        updateRealTime();
        //total seconds since last alarm
        int alarmSecs = rtcTime[0] - timeOfLastAlarm[0]; //seconds difference
        alarmSecs = alarmSecs + rtcTime[1] - timeOfLastAlarm[1]; //mins
        alarmSecs = alarmSecs + rtcTime[2] - timeOfLastAlarm[2]; //hours

        alarmEnable = false;
        if(alarmSecs>180){ //more than 3mins has passed
            alarmEnable = true;
        }

    }
    lastInterruptTime = now;
}

```

Fig. 13: Code for ChangeInterval and DismissAlarm interrupt Handler

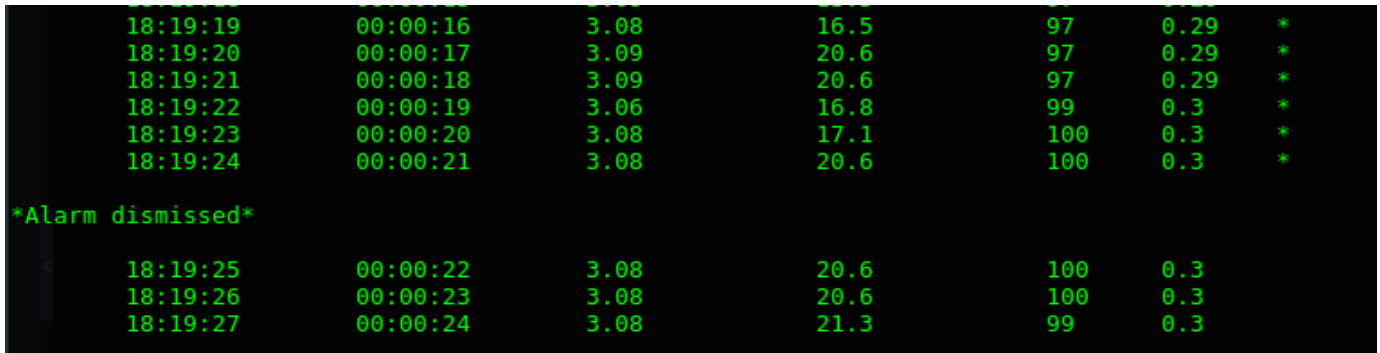
## V. VALIDATION AND PERFORMANCE

The system worked as intended, with 3 inputs (light, temperature, humidity) being correctly detected by the ADC and output to the terminal. The change in any of these variables was correctly reflected and the Vout calculated and outputted to the DAC.

RTC Time	System Time	Humidity	Temperature	Light	DAC Out	Alarm
18:19:04	00:00:01	3.26	17.1	91	0.29	*
18:19:05	00:00:02	3.27	20.3	83	0.27	*
18:19:06	00:00:03	3.27	20.6	105	0.34	*
18:19:07	00:00:04	3.26	20	108	0.34	*
18:19:08	00:00:05	3.27	17.4	48	0.15	*
18:19:09	00:00:06	3.27	20.3	22	0.07	*
18:19:10	00:00:07	3.26	20.3	39	0.12	*
18:19:11	00:00:08	3.26	20	92	0.29	*
18:19:12	00:00:09	3.27	16.1	93	0.3	*
18:19:13	00:00:10	3.21	19.7	93	0.29	*
18:19:14	00:00:11	2.72	20	92	0.24	*
18:19:15	00:00:12	2.38	21	92	0.21	*
18:19:16	00:00:13	3.01	20	92	0.27	*
18:19:17	00:00:14	3.09	20	99	0.3	*
18:19:18	00:00:15	3.09	13.5	97	0.29	*
18:19:19	00:00:16	3.08	16.5	97	0.29	*
18:19:20	00:00:17	3.09	20.6	97	0.29	*
18:19:21	00:00:18	3.09	20.6	97	0.29	*
18:19:22	00:00:19	3.06	16.8	99	0.3	*
18:19:23	00:00:20	3.08	17.1	100	0.3	*
18:19:24	00:00:21	3.08	20.6	100	0.3	*

Fig. 14: Monitoring of system behaviour

The alarm was successfully dismissed after pressing the button.

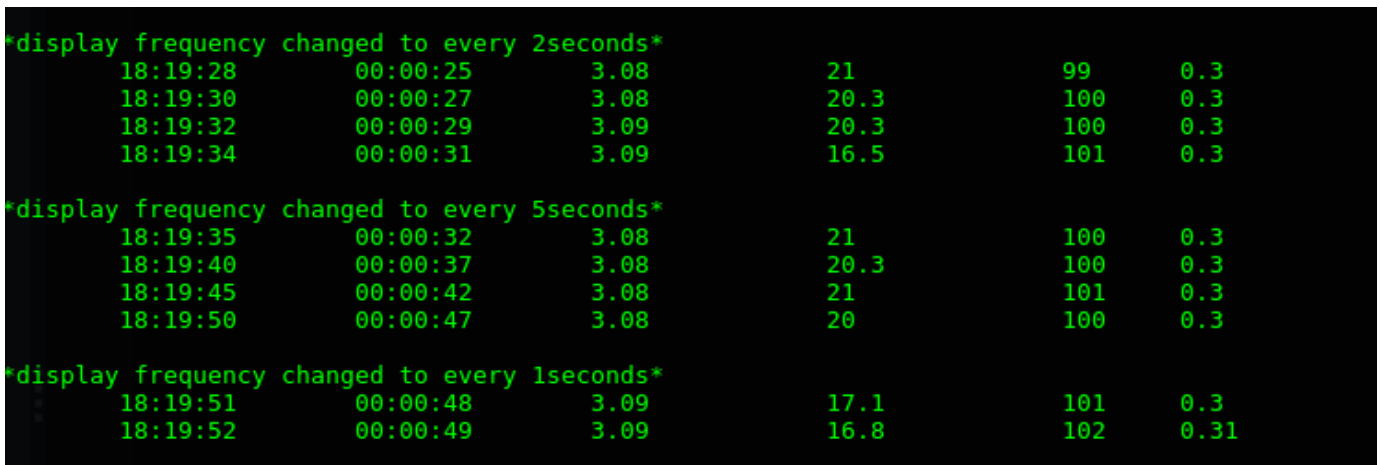


The screenshot shows a terminal window with green text on a black background. It displays a series of sensor readings (time, RTC time, and three numerical values) followed by a message indicating the alarm was dismissed. The data is as follows:

System Time	RTC Time	Value 1	Value 2	Value 3	Value 4
18:19:19	00:00:16	3.08	16.5	97	0.29
18:19:20	00:00:17	3.09	20.6	97	0.29
18:19:21	00:00:18	3.09	20.6	97	0.29
18:19:22	00:00:19	3.06	16.8	99	0.3
18:19:23	00:00:20	3.08	17.1	100	0.3
18:19:24	00:00:21	3.08	20.6	100	0.3
*Alarm dismissed*					
18:19:25	00:00:22	3.08	20.6	100	0.3
18:19:26	00:00:23	3.08	20.6	100	0.3
18:19:27	00:00:24	3.08	21.3	99	0.3

Fig. 15: Dismissal of Alarm

The frequency was successfully changed between 1Hz 0.5Hz 0.2Hz with another button.



The screenshot shows a terminal window with green text on a black background. It displays three sections of sensor data, each preceded by a message indicating a change in display frequency. The data is as follows:

*display frequency changed to every 2seconds*					
System Time	RTC Time	Value 1	Value 2	Value 3	Value 4
18:19:28	00:00:25	3.08	21	99	0.3
18:19:30	00:00:27	3.08	20.3	100	0.3
18:19:32	00:00:29	3.09	20.3	100	0.3
18:19:34	00:00:31	3.09	16.5	101	0.3
*display frequency changed to every 5seconds*					
System Time	RTC Time	Value 1	Value 2	Value 3	Value 4
18:19:35	00:00:32	3.08	21	100	0.3
18:19:40	00:00:37	3.08	20.3	100	0.3
18:19:45	00:00:42	3.08	21	101	0.3
18:19:50	00:00:47	3.08	20	100	0.3
*display frequency changed to every 1seconds*					
System Time	RTC Time	Value 1	Value 2	Value 3	Value 4
18:19:51	00:00:48	3.09	17.1	101	0.3
18:19:52	00:00:49	3.09	16.8	102	0.31

Fig. 16: Change in display frequency

The reset button cleared the terminal and started the system time from 00:00:00. The start/stop button started/stopped the logging to the screen successfully. When the alarm was active, an LED turned on to indicate as such. The alarm was prevented from activating within 3 minute period of last alarm.

The ADC readings from all the three sensors, alarm indication, RTC time and system time, all were logged remotely and viewed live in the Blynk app.

1. Figure 17 below shows the Blynk console when Vout is between 0.65V and 2.65V and thus, the alarm is OFF (indicated by black circle)
2. Figure 18 below shows the Blynk console when Vout is less than 0.65V and thus, the alarm is ON (indicated by Red circle)

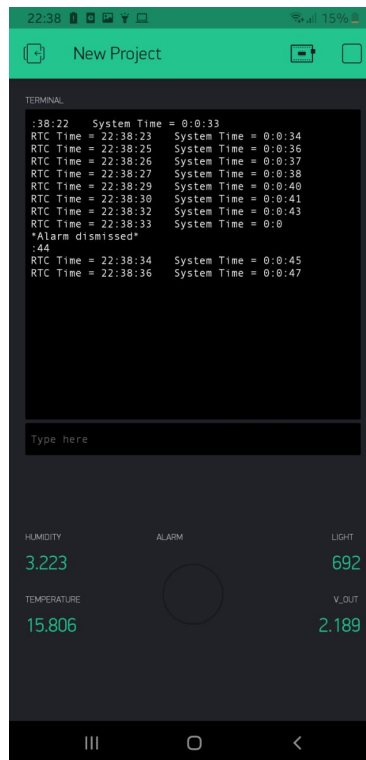


Fig. 17: Blynk console showing all readings with the alarm being OFF

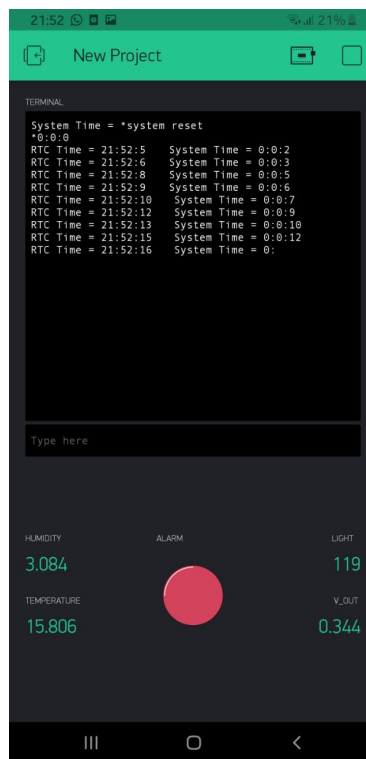


Fig. 18: Blynk console showing all readings with the alarm being ON

## VI. CONCLUSION

This project successfully fulfilled its objective of designing and building an environment logger for monitoring temperature, humidity and light levels using a Raspberry Pi and relaying these information to the Blynk App for remote monitoring. It was successful in creating and deploying an IOT device with the intention of developing a representative embedded system. The environment logger system worked as intended with the monitoring starting with a button press and all three of its sensor inputs being correctly detected by the ADC which used threads for reading these values. The RTC was correctly setup indicating the system time accurately and the output voltage from DAC was used to determine the triggering of the alarm. The alarm was successfully initiated with an LED turning ON when Vout was either less than 0.65V or greater than 2.65V and also being dismissed after a button press. The alarm was also prevented from activating within a three minute period of the last alarm. The frequency change of monitoring was successful as it managed to change from 1s to 2s to 5s with a button press. Pressing the reset button cleared the terminal and ensured that the system time was reset to 00:00:00. As intended, the system timer was updated when reset button was pressed and not affected at all by the press of the Start/Stop switch. The ADC readings from all the three sensors, alarm indication, RTC time and system time, all were logged remotely and viewed live in the Blynk app. An Environment logger system working this way is definitely a potentially useful product as not only does it give accurate readings of time and sensor data but it also provides them remotely to the users' mobile device which makes it extremely useful as the user does not have to constantly be in the greenhouse for monitoring. The one drawback to this system could be during a loss of internet connectivity, which would mean that no data could be relayed to the app making it visible for risks. All in all, an IOT system like this is extremely useful and user friendly which definitely makes it a potentially useful product in the market.

## REFERENCES

- [1] Vula briefing  
[Project A Document](#)
- [2] Keegan's Github  
<https://github.com/kcranky/EEE3096S>
- [3] Raspberry Pi pin configurations  
<https://pinout.xyz/>
- [4] MCP3008 ADC Datasheet  
<https://cdn-shop.adafruit.com/datasheets/MCP3008.pdf>
- [5] MCP9700A Temperature sensor Datasheet  
<http://ww1.microchip.com/downloads/en/devicedoc/20001942g.pdf>
- [6] Guide to Blynk  
<https://blynk.io/>
- [7] maxEmbedded: I2C Basics  
<http://maxembedded.com/2014/02/inter-integrated-circuits-i2c-basics/>
- [8] SPI WiringPi Details  
<http://wiringpi.com/reference/spi-library/>