EEE3096S: Embedded Systems II

13 October 2019

Remote Environmental Logger

Mini Project A

Iviwe Malotana (MLTIVI001)

Matthew Lock (LCKMAT002)

1 Introduction

This report serves to demonstrate the inception, design and implementation of an Environmental Logger. The task at hand is to develop a monitoring system for the conditions in a private greenhouse. These conditions are time of day, system run time, humidity, light, temperature, a DAC voltage output and an alarm to signal an exceeded DAC voltage. This data must be sent to a display on an application, so it may be monitored remotely. In order to do this, the particular hardware and software needed is chosen wisely.

The hardware capabilities of a Raspberry Pi 3B + have proven successful in multiple projects thus far, so it is no surprise that the pi is the IoT device of choice in this scenario. It has multiple pins through which components from DACs, ADCs and RTCs can all connect to. The breadboard is generously packed with a few components to monitor the analog signals of the greenhouse. Temperature is monitored using a temperature sensor, an LDR voltage divider is used for light input and a potentiometer is used to mimic a humidity sensor. These analog inputs must be converted into digital form and this is established by connecting all the three devices to the input of an ADC which send the output to the raspberry pi. The raspberry pi is also connected to a DAC. This DAC functions to read the digital light and humidity reading and convert it into a voltage which it outputs on its eighth pin back to the pi. It is highly necessary to have an RTC running to monitor the current time, so it also connected to the raspberry pi to fulfil its function of keeping true time. The pi is accompanied by four push buttons and an alarm. The buttons each perform one of these functions; starting/stopping the running program, increasing the time interval between reading values in a set, dismissing the alarm should it go off and resetting system run time.

On a software level, C++ code is used to create the program. This program sets up wiringpi and establishes the connection between the pi and the multiple devices it is connected to. Among those devices, an android device with the Blynk application is used to display the data on screen.

The sections in this report aim to qualify the design process and the implementation of this project. The hardware and software requirements are noted. This is followed by the depiction of a UML use case diagram which is used to illustrate an overview of the interaction between the user, pi, DAC, ADC, RTC and Blynk. Specification and design includes a state chart, UML class diagram and a circuit diagram for further diagrammatic elaboration of the hardware and software connections. The implementation section will discuss the process of designing the system and an in depth qualification of the state chart accompanied with code snippets. And lastly, a validation and performance section is included where the overall performance and test cases are evaluated for the optimal functioning of the monitoring system.

2 Requirements

Table 2.1 Showing Hardware Requirements

Function	Requirement	ID
INPUT HARDWARE	Sensors for light, temperature, humidity and RTC must be included. Input devices to control the start/idle state of the program, changing the frequency of sensor sampling, dismissing the alarm and performing system run time reset should be present.	R1
PROCESSING HARDWARE	A device to communicate with the software and other hardware must be used. This will run the program and communicate with both the input and output sources.	R2
OUTPUT HARDWARE	Alarm output - A device to output an alarm signal on the hardware should exist, like a buzzer.	R3
	Display output- An android device to run Blynk to display output is necessary.	R4

Table 2.2 Showing Software Requirements

Function	Requirement	ID
SETUP	RTC - The RTC must be set up using a kernel driver or the I2C protocol used in practical 3.	R5
	DAC and ADC - Communication to the DAC and ADC must be facilitated using the correct protocol.	R6
	Buttons - All push buttons must be configured with interrupts and debouncing.	R7
INPUT	ADC - A thread to read from the ADC must be created and the temperature reading from the thermistor must be converted to degrees Celsius.	R8
PROCESS	DAC output voltage- The DAC must calculate a DAC output voltage according to the following formula:	R9

	$V_{out} = \frac{LightReading}{1023} \times HumidityVoltageReading$	
	Alarm signal - An alarm must be triggered when the voltage is below 0.65V or over 2.65V.	R10
OUTPUT	Blynk output -The output must be sent to Blynk for display.	R11

Use Case Diagram

Diagram 2.3 is the use case figure and it shows the relationship between the user, Blynk, DAC, ADC, RTC and the raspberry pi. The use case diagram does omit some information, particularly about where the ADC receives the humidity, temperature and light values. The humidity signal is sent from a potentiometer. The temperature reading is linked through the ADC to a temperature sensor and the light from an LDR.

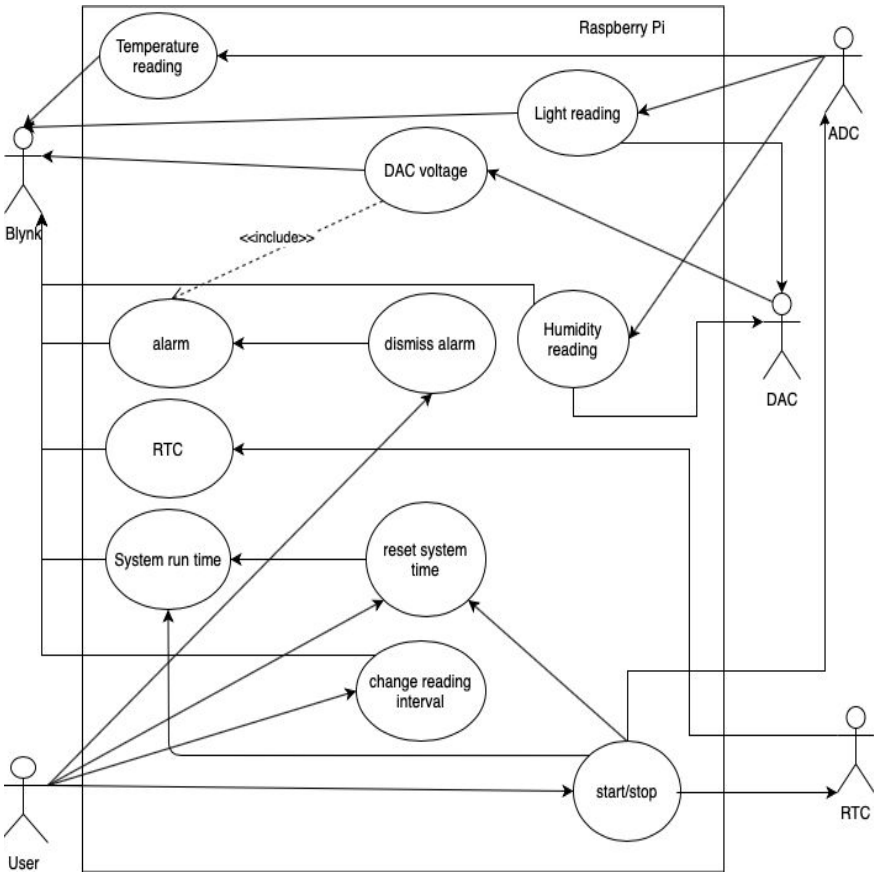


Diagram 2.3 Showing Use Case UML Diagram for the System

3 Specification and Design

UML State Chart

Diagram 2.2 is a state chart and its main effort is to show the behavior of the system in response to stimuli. The path of the program is followed from when it is running to its disturbance and decision stages and through to termination.

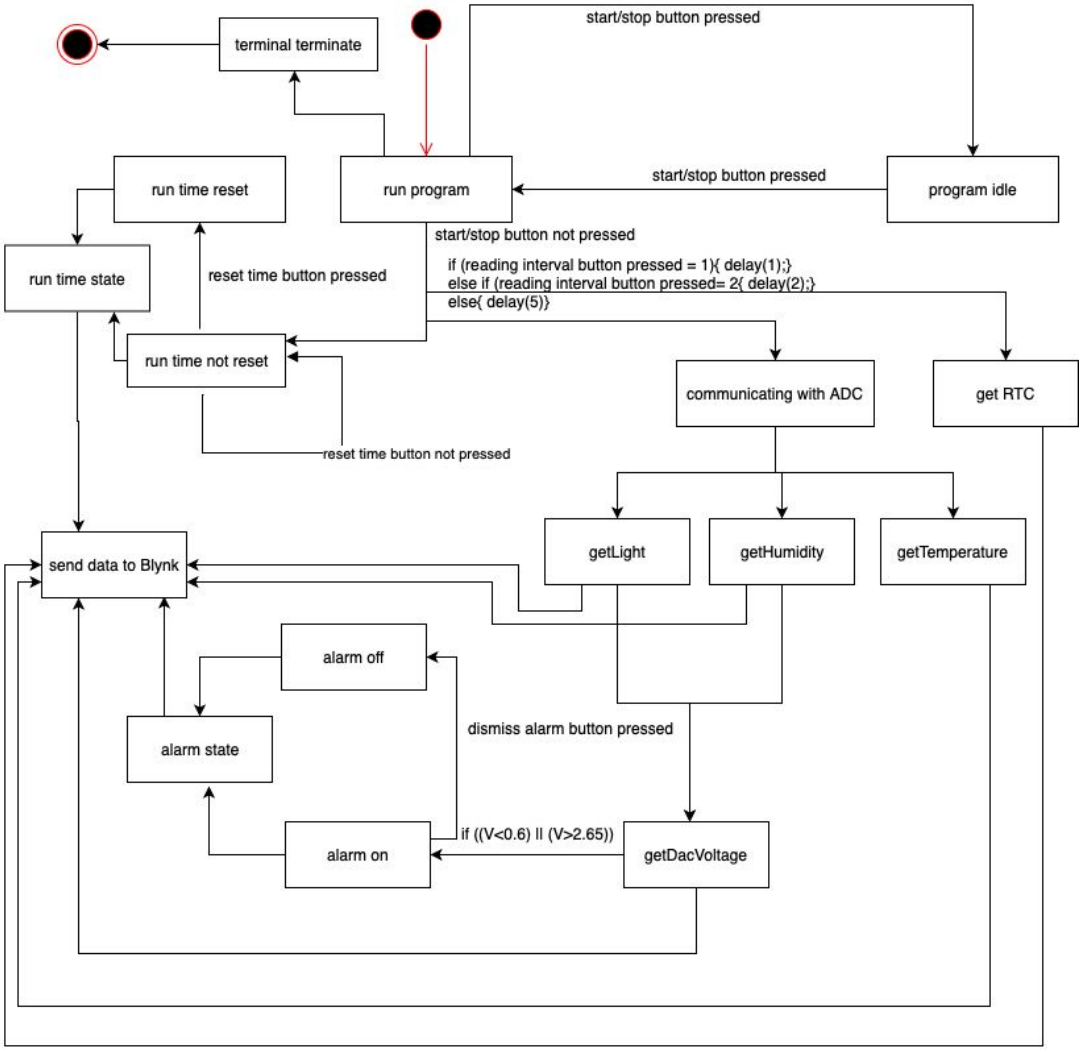


Diagram 2.2 Showing UML State Chart

UML Class Diagram

Diagram 2.3 is a class diagram illustrating the relationship between the CurrentTime and the main class of the program.

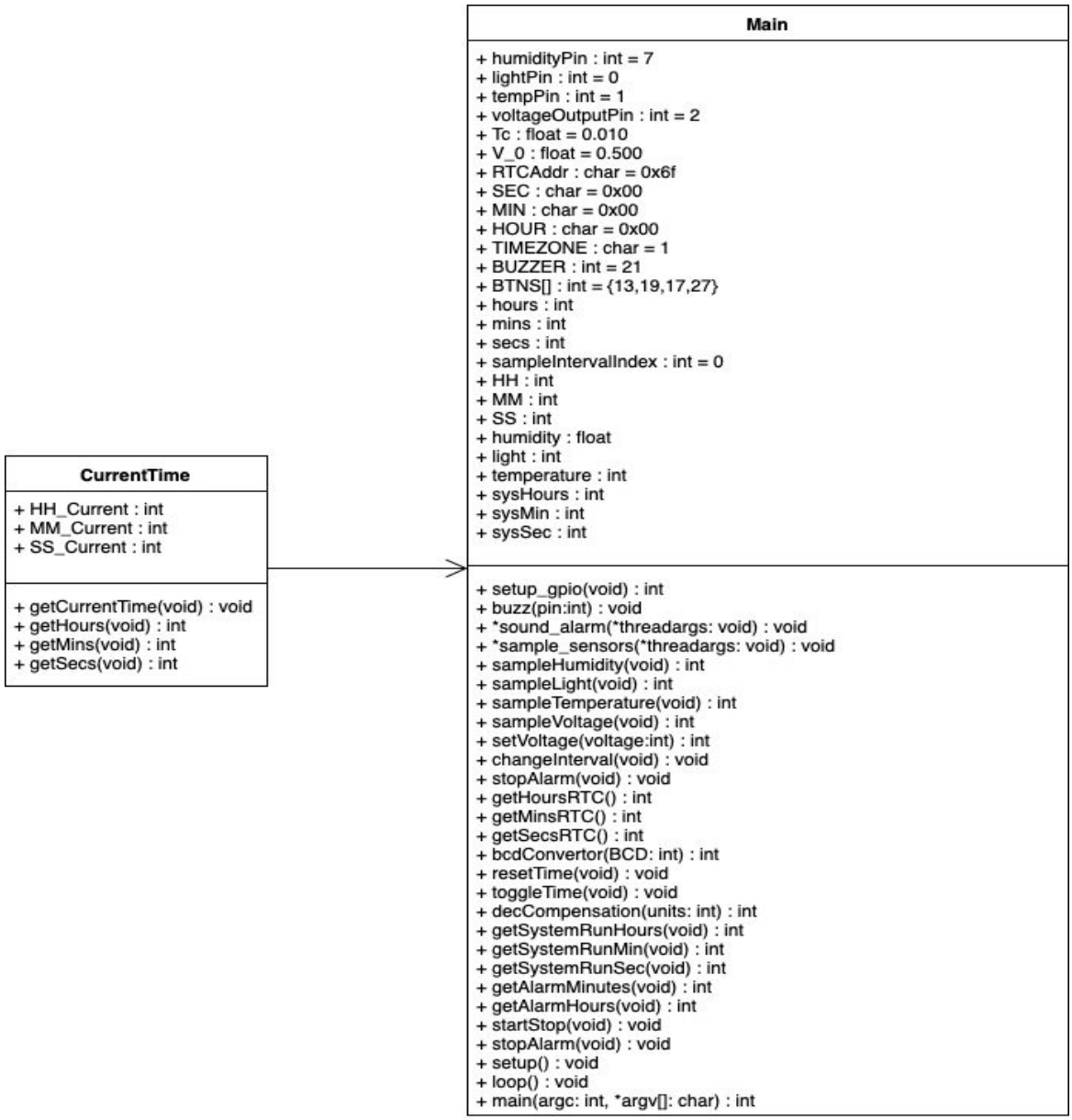


Diagram 2.3 Showing UML Class Diagram

Circuit Diagram

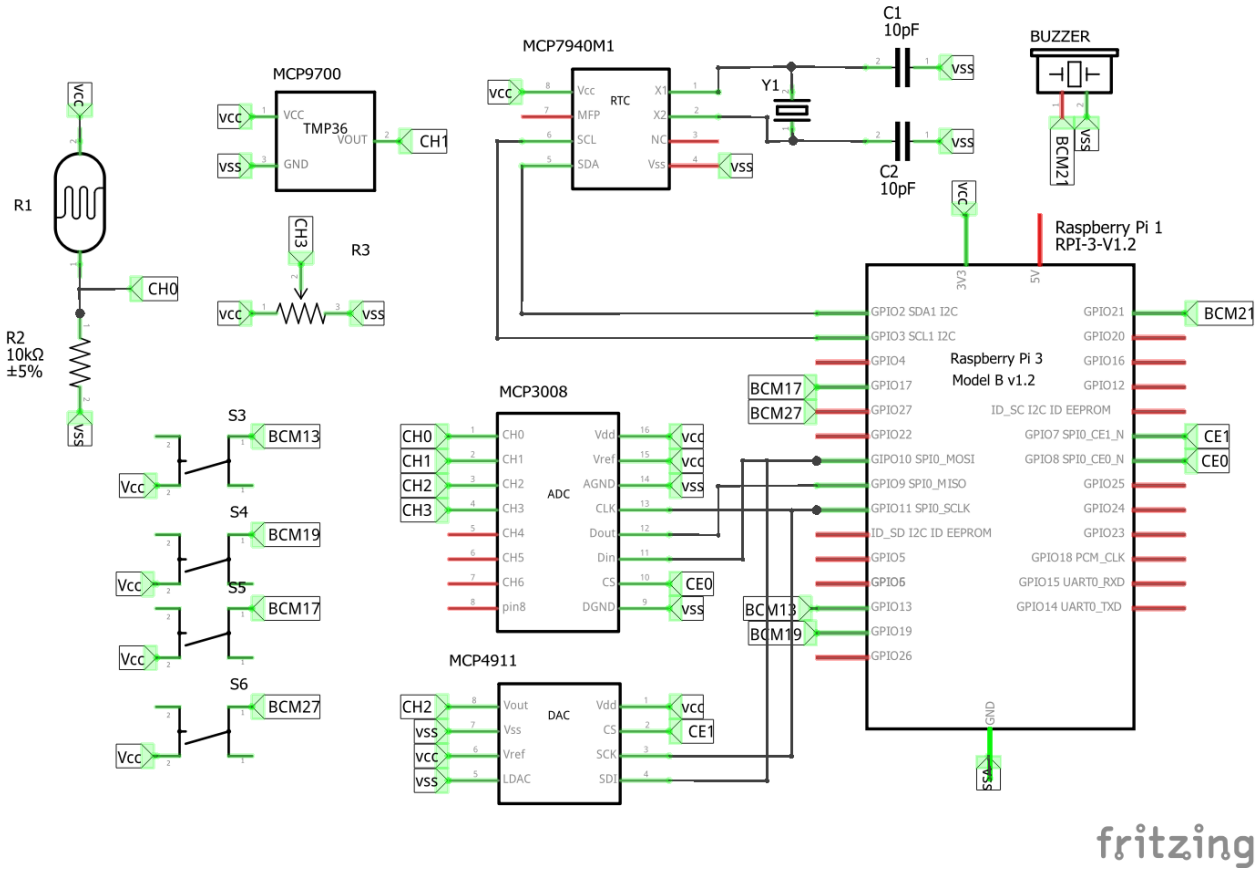


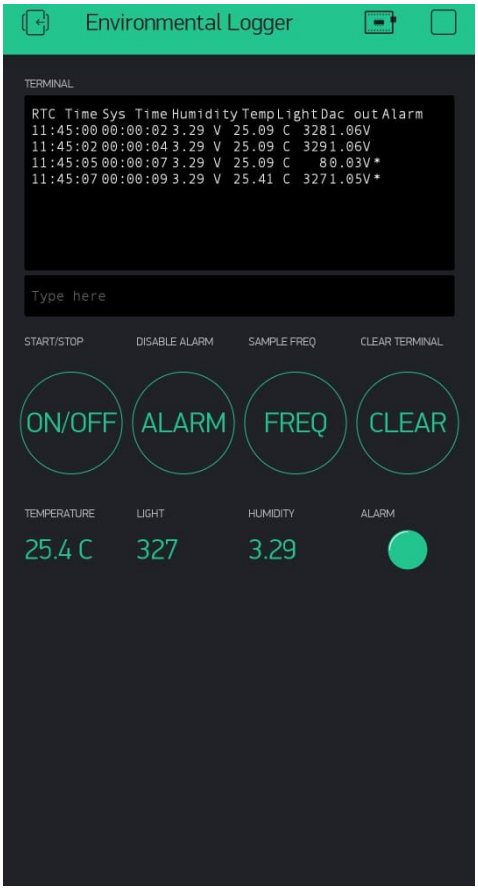
Diagram 2.3 Showing Circuit Diagram

4 Implementation

In order to implement the necessary functionality, design decisions were made to balance and maximise efficiency, as well as simplicity. These decisions include the number of threads to be implemented, methods used to poll environmental sensors, methods for remote monitoring of polled data, and the methods used to control timing of the system.The following section details why each decision was taken, and includes code snippets to show how they were implemented.

Remote Monitoring

As per the system requirements, any and all environmental sensors implemented need to be capable of remote monitoring. Furthermore, the remote monitoring system needs to be capable of controlling the system in the same manner that the system is controlled by hardware buttons. Thus the functionality of the remote monitoring system would include starting or stopping the polling of environmental data, disabling the alarm, changing the polling frequencing, and resetting the system time.



For these needs it was decided that an IoT platform known as **Blynk** would be used**.** Fundamentally, Blynk is a Platform with IOS and Android apps to control the Raspberry Pi over the Internet, and contains a digital dashboard where you can build a graphic interface for any project by simply dragging and dropping widgets. This would make designing a functional and aesthetically pleasing remote control and monitoring interface simple. Additionally, Blynk makes use of WiringP pin based GPIO access library which would make integration with existing C++ code from previous practicals seamless.

Diagram 4.1 alongside shows the user interface design of the environment logger in the Blynk application. As per the requirements a terminal has been implemented to display detailed information pertaining to timing and sensor information, while labeled text fields and an LED indicator allows for easy monitoring of the system state.

Large labeled buttons were selected to enable remote control of the system as they provide an intuitive way to interact with the system and provide the user with an internal locus of control.

Diagram 4.1 Screenshot of Blynk Application

In order to integrate our application and interface with the designed blynk application, the blynk library (https://github.com/blynkkk/blynk-library) was downloaded and our source code integrated with the **main.cpp** file. Adjustments had to be made to our original source code as WiringPi within the Blynk application made used of the BCM pinout scheme while we had previously made use of the WiringPi pinout scheme.

Terminal

The terminal was implemented by means of dragging a terminal widget into the Blynk application and then creating a terminal widget our C++ code assigned to Virtual Pin 0 :

```
WidgetTerminal terminal(V0);
```

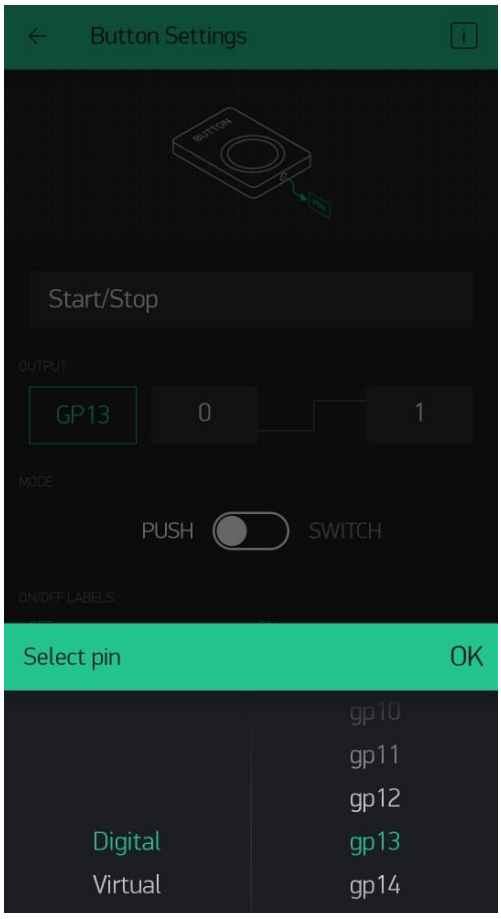
The terminal widget could now be written to by means of a virtualWrite command and cleared by calling a console clear command:

```
Blynk.virtualWrite(0,buffer);//Writes contents of buffer to blynk terminal
terminal.clear(); // Clears the contents of blynk terminal
```

Buttons

Button integration with existing source code was seamless. Since the Blynk application made use of wiringPi and could directly control the state of GPIO ports, buttons in the user interface were assigned to the appropriate BCM port where they would trigger the appropriate interrupt if pressed.

The snippet of code (shown below) and screenshot (shown alongside) detail the method used to create button functionality in the C++ code and assign the appropriate buttons in the Blynk application to the various functions.




```
const int BTNS[] = {13,19,17,27}; // B0, B1, B2, B3
//Set up the Buttons
for(unsigned int j=0; j < sizeof(BTNS)/sizeof(BTNS[0]); j++){
    pinMode(BTNS[j], INPUT);
    pullUpDnControl(BTNS[j], PUD_DOWN);
}

wiringPiISR (BTNS[0], INT_EDGE_RISING,  &startStop );
```

Sensor Readings

Sensor readings were pushed to the Blynk application at regular intervals selected by the user. Much like the terminal widget, a Value Display Setting widget was dragged into the blynk terminal and assigned to a Virtual Pin. A value was then pushed to the display in code with a virtual write as follows:

```
Blynk.virtualWrite(1,environmental_temperature);
```

Threads Implemented

The implementation of Blynk for remote monitoring meant that to achieve time controlled events, a method for polling sensor values from the ADC would have to run without interrupting the connection to Blynk, as calling a sleep or delay function from within the blynk connection thread would cause the connection to be dropped. This either means that the separate threads can be created with one thread continuously keeping the Blynk connection alive, or creating a Blynk timer event to perform regularly scheduled task.

Instead of performing all checks sequentially with a Blynk timer event, it was decided to create independent threads to handle the different tasks. This would allow for regular scheduled communication with the Blynk application, continuous affirmation of the Blynk connection, as well as performing other tasks such as controlling of the alarm in parallel.

Sensor Thread

A thread was created that is responsible for sampling the sensor values by polling the MCP30078 (ADC) for the relevant integer values. Since this thread plays no part in keeping the Blynk connection alive, it was decided to place the polling frequency functionality inside this thread by placing a sleep() function at the end of the thread. This sleep function would sleep for a period of time selected by the user by a bottom press. This thread is also responsible for checking whether the alarm needs to be activated and updating the system time. The following code shows how this was implemented:

```
int sampleInterval[3]={1,2,5}; //Regular intervals
unsigned int sampleIntervalIndex = 0; //Initial index of sampleInterval
/*
* Thread to sensor sampling
*/
void *sample_sensors(void *threadargs){
   printf("Starting sensor thread\n");
   …  //Example of continuous thread with while loop
sleep(sampleInterval[sampleIntervalIndex]); //Sleep for selected time
...
```

```
/*
* changeInterval
* Change the interval for which measurements are made
* Software Debouncing used
*/
void changeInterval(void){
   //Debounce
   long interruptTime = millis();

   if (interruptTime - lastInterruptTime>200){
       printf("\nInerval Changed!\n");
       sampleIntervalIndex++;
    if (sampleIntervalIndex>(sizeof(sampleInterval)/sizeof(sampleInterval[0])-1)) {
           sampleIntervalIndex=0;
       }

   }
   lastInterruptTime = interruptTime;
}
```

The advantage of introducing the delay in this specific thread is that the RTC, ADC and DAC are all interfaced with at regular and necessary intervals. This means that communication between the devices is only done when necessary and does not take up any unnecessary CPU time. Furthermore pushing the updates to Blynk within this thread using virtual writes prevents the Blynk application from blocking the connection. Having a thread continuously pushing data would cause the Blynk app to block the connection due to spam.

Since the values acquired from the ADC are in the form of an integer representation of a voltage, conversions are required to extract the appropriate measurements. These conversions can be found below:

```
//Conversions
float dac_output = light/(float)1023 * humidity;
float environmental_temperature= ((temperature*3.3/1024)-V_0)/Tc;
```

The last task this thread is responsible for changing the voltage level on the (MCP4911) DAC using SPI protocol. The snippet of code below shows how the setVoltage function is called within the sampling thread, as well as how the setVoltage function communicates with the DAC.

```
//Set DAC voltage
dac_output=(int)(dac_output/3.3*1024);
setVoltage(dac_output);
```

```
int setVoltage(int voltage){
   //Set control and data bits
   unsigned char dacBuffer[2];
   dacBuffer[0]=48 | (voltage>>6);
   dacBuffer[1]=252 & (voltage<<2);
   //Send buffer
   return wiringPiSPIDataRW (SPI_CHAN_DAC, dacBuffer, 2);
}
```

Alarm thread

If the alarm activated boolean has been set to **True** by the sensor thread, a separate thread has been created to handle the buzzer that signals that an alarm event is ongoing. The reason a seperate thread has been created is that the buzzer has delay functions within it to replicate the sound of an alarm. If this thread were to perform any other function is would stop timing critical events such as sensor sampling from occurring. The thread functionality is shown below:

```
/*
* Thread to handle alarm activation
*/
void *sound_alarm(void *threadargs){
   printf("Starting alarm thread\n");
   fflush(stdout); // Make sure printf works
   for (;;){
       //Check for alarm activation
       fflush(stdout); // Make sure printf works
       while (alarmActive){
           softToneWrite (BUZZER,2200) ; // Turn buzzer on
           usleep(400*100);
           softToneWrite (BUZZER,0) ; // Turn buzzer off
           usleep(400*100);
       }
   }
   pthread_exit(NULL);
}
```

Timing Critical Events

As described in the aforementioned section on the sampling thread, time critical events such as sampling the sensors are controlled by the sampling thread. The user has the option of starting or stopping the system. While this option stops the logging of environmental data, it should not affect the system time in any way.

To achieve this, the user was allowed to set the state of a boolean through a button press of the “Start/Stop” button. This causes the sampling thread to sit in a waiting state (as shown below) where it does not sample the ADC (sensors) and does not display to the screen. Since the RTC is not stopped in any way and the information pertaining to the start time of the system has not been changed, the run time of the system is unaffected.

```
void *sample_sensors(void *threadargs){
   printf("Starting sensor thread\n");
   fflush(stdout); // Make sure printf works
   for (;;){
       while(start!=1){
           sleep(1);
           //wait
       }
...
```

5 Validation and Performance

In order to validate that the system performs correctly, each specification was allocated a test case. These test cases were then run, with the results of these tests tabulated below. The final test results show that the system all of the determined design and user specifications.

ID	Functional Requirement	Design Specification	Test Case
R1, R6, R8, R9	System must measure light conditions, temperature and humidity levels.	The system must contain a LDR, potentiometer, thermistor (MCP9700A), and an ADC (MCP3008)	TC1 ,TC6, TC7, TC8
R1, R5	The system must keep track of the current time and system time.	RTC must be included	TC2, TC3,
R1, R7	USer must be able to control the start/idle state of the program, frequency of sensor sampling, alarm state and system time.	There need to be 4 hardware buttons and 4 Iot buttons implemented	TC2, TC3, TC4, TC5
R2	The device must be able to communicate with IoT software and other hardware used	The Raspberry Pi 3B will be used to run a C++ application	-
R3, R10	The system must sound an alarm when output voltage falls outside of threshold.	A DAC (MCP4911) and piezo electric buzzer must be used	TC6
R4, R11	The system should send all sensor readings to an IoT application	A smart device capable of running the Blynk Application is needed	TC2, TC3, TC4, TC5

Table 3.2 Showing Acceptance Test Protocol

Test Case ID	Test Procedure	Requirement in order to pass	Test Result
TC1	1) Start the application on the raspberry pi by running: ***sudo ./blynk --token=YOq372w339cskEXQHy4l6mdy1yH8VFxI*** 2) Press the start button once	1) The system shows ***Ready (ping: 222ms).*** to indicate a successful connection to blynk 2) The system starts	Pass

	(physical hardware button) 3) Press the start button once (physical hardware button)	outputting sensor data to the pi terminal in the required format at an interval of once per second after the first press 3) The system stops printing to the raspberry console after the second press	
TC2	1) Start the application on the raspberry pi 2) Press the start button once (physical hardware button) 3) Let the system run for exactly 1 minute and 30 seconds	1) The system starts correctly and replicated the results shown in TC1 2) Sys time shows the value ***00:01:30***	Pass
TC3	1) Start the application on the raspberry pi 2) Press the start button once (physical hardware button) 3) Let the system run for exactly 1 minute and 30 seconds 4) Press the reset button once (physical hardware button)	1) The system starts correctly and replicated the results shown in TC1 2) The system clears the raspberry pi console when the reset button is pressed and the sys time now shows ***00:00:00***	Pass
TC4	1) Start the application on the raspberry pi 2) Press the interval button once (physical hardware button) 3) Press the interval button once (physical hardware button) 4) Press the interval button once (physical hardware button)	1) The system starts correctly and replicated the results shown in TC1 2) The system prints to the console at a regular interval of 2 seconds after the first button press 3) The system prints to the console at a regular interval of 5 seconds after the second button press 4) The system prints to the console at a regular interval of 1 seconds after the third button press	Pass
TC5	1) Start the application on the raspberry pi 2) Repeat TC1-5 but instead of pressing hardware buttons, press buttons on the Blynk application	1) All results from TC1-5 are seen, but they observed from the Blynk terminal as well as the raspberry terminal.	Pass

		2) Raspberry terminal shows addition button information (which button has been pressed) where Blynk terminal does not	
TC6	1) Start the application on the raspberry pi 2) Press the start button once (physical hardware button) 3) Place finger directly over the LDR	1) The system starts correctly and replicated the results shown in TC1 2) The DAC output shown falls below a value of 0.65V 3) Both the raspberry and Blynk terminal show an additional '*' character 4) The Alarm LED on the Blynk application lights up 5) The Buzzer can be heard beeping loudly	Pass
TC6	1) Start the application on the raspberry pi 2) Press the start button once (physical hardware button) 3) Place thumb and index finger over temperature sensor	1) The system starts correctly and replicated the results shown in TC1 6) The DAC output shown falls below a value of 0.65V 7) Both the raspberry and Blynk terminal show an additional '*' character 8) The Alarm LED on the Blynk application lights up 9) The Buzzer can be heard beeping loudly	Pass
TC7	1) Start the application on the raspberry pi 2) Press the start button once (physical hardware button) 3) Place light source directly over the LDR 4) Measure output voltage on DAC	1) The system starts correctly and replicated the results shown in TC1 2) Voltage measured on DAC output is within 0.1V of DAC output displayed in terminal	Pass
TC8	1) Start the application and press the start button 2) Remove pot from circuit and used a voltage supply to supply the ADC humidity pin with 2.2 V.	3) The system starts correctly 4) Output humidity in terminal is within 0.1V of 2.2 V	Pass

6 Conclusion

The design process followed throughout this report shows a successful implementation of a remote environmental logger. This was proved through multiple condition testing with well lit environments, badly lit environments, environments with increased temperature and simulated humidity levels. The user specifications proved to be well within the capabilities of the selected hardware (Raspberry Pi) as supports all the needed communication hardware, as well as IoT features. Although not accurate to more than a tenth of a degree, the output DAC voltage means this system can be successfully implemented with other systems looking to take this voltage in as a signal. The system in its current state could be expanded further to output voltage levels specific to light and temperature readings to devices in the greenhouse that could effectively lower light and temperature levels. This leads us to believe that the system has a lot of potential for further expansion.

It must be noted that while the system holds great potential to future applications and expansion, in its current state the system only supports a very specific use case. The system would not be suitable as a product to anybody other than the specific customer for which the system was tailored for, or clients with very similar needs. Reasons for this include the limitation that the system must be connected to a computer with an internet connection. In its current state the system has powered by long lasting power source (such as a battery) and has not been configured to connect to a wireless network for it's IoT functionality. Another pressing issue is that while the system has been integrated with an IoT platform such as Blynk, no easy way of replicating the Blynk application on other devices has been implemented. Furthermore, installation of the system would require the customer to have an intimate understanding of electronics as they would have to setup the circuitry required themselves.

If the product were setup to accept wireless connection, be powered by a long lasting battery, provide a web page for IoT monitoring, and with the circuitry being self contained on a PCB within an enclosure, we believe the product would be suitable for a wide range of environmental logging applications.