EEE3095S Project A - An Environment Logger using a Raspberry Pi and Blynk

Hendri Vermeulen[†] and Kyle du Plessis[‡]
EEE3095S Class of 2019
University of Cape Town
South Africa
[†]VRMHEN004 [‡]DPLKYL002

***Abstract*—A report detailing the implementation of an environment logger using a Raspberry Pi and Blynk mobile application to measure and report the temperature, humidity and light of an environment the Raspberry Pi is placed in both through a terminal and online Blynk mobile application for remote access to the information. To obtain this information about the environment an ADC, MCP3008, is used to convert voltages from sensors: MCP9700A for temperature, LDR (Light Dependent Resistor) for light and Potentiometer to mimic VPD sensor for humidity. Finally there is a DAC, MCP4911, to output a voltage obtained with equation, containing the light and humidity values, to trigger an alarm, flashing PWM LED, when the voltage is too low or high. All this information is reported to a Blynk mobile application (for remote data access) and terminal (for local data access). There are also 4 buttons: reset - resets system, frequency - changes frequency of reading in of data, stop - stops/resumes reading in of data, dismiss - dismisses alarm. Due to the Raspberry Pi only using the Internet for time, a RTC was also added to the system to track time accurately.**

I. Introduction

This report provides a detailed systems specification for a single iteration of the environment logger project, which can be regarded as a representative embedded system (an IoT device). This project had arisen from the user requirements specification for a particular scenario which involves monitoring a private greenhouse. The task, according to the client request, comprises of monitoring the time of day, time since the system has been running, light levels, temperature and humidity. Furthermore, they also wish to be able to monitor this data remotely through a Blynk mobile app.

The system was implemented on a Raspberry Pi using Python as a fixed coding environment. Python was chosen due to particular libraries available in Python which made development quicker and improved code quality through code legibility. The user already has a humidity monitoring device that outputs a voltage level between 0 and 3.3V, thus we will mimic this with a Potentiometer, which at a later stage can be replaced with the humidity device. Furthermore to insure accuracy and performance we made use of threading as far as possible, for reading from the ADC, displaying log information, talking to the Blynk mobile application and updating the flashing PWM LED alarm was all done in separate threads.

The report structure follows a logical sequence covering various sections which include: Requirements, Specification and Design, Implementation, Validation and Performance as well as a Conclusion section.

The Requirements section includes a refined UML Use Case diagram which clarifies the functional requirements according to the project description and provides a high-level overview of the system's functionality. This section also serves as a platform on which we will be building the proposed system during the implementation stage.

The Specification and Design section includes a UML State Chart diagram which describes the system's main operation, as well as a UML class diagram which describes the structuring of the implementation. It also includes a circuit diagram which corresponds to the physical arrangement of the finished system and shows all the hardware components used.

The Implementation section includes important code snippets and explanations to clarify the implementation of the system.

The Validation and Performance section describes the performance of the system. It also includes test cases which tests that the system works reliably and meets the specified functional requirements.

The Conclusion section describes the extent that the system was found to be successful. It also includes a discussion of the system working in this way being considered a potentially useful product.

II. Requirements

In this section we'll be describing physical requirements (hardware) and user requirements, making use of an Use Case Diagram, to describe the requirements of this project

A. User requirements

A biology student, referred to now on as the user, want to monitor their private green house. He needs to monitor the time of day, time sine system has been running, light levels, temperature and humidity - already has a device that outputs a voltage between 0 and 3.3V, of the green house. This data should be accessible through a local terminal and remotely using a aesthetically pleasing Blynk application on a mobile device. The user also requested that the system should output a voltage equal to Vout = (LightReading/1023)*HumidityVoltageReading. This voltage level is used to trigger an alarm, if it goes under 0.65V or above 2.65V. The alarm should only sound if there was no alarm dismissal more than 3 minutes ago - this is timed when the alarm is sounded. Furthermore he needs to be able to stop/resume the monitoring, reset the system time, dismiss the alarm and changing the reading interval.

B. Hardware requirements

To be able to implement all the functions the following hardware will need to be required: Raspberry Pi with Networking Capabilities - for Remote connection with the Blynk application, an ADC for reading the values from different sensors - MCP3008, light sensor - LDR (Light Dependent Resistor), temperature sensor - MCP9700A, DAC - MCP4911, a device that can mimic the voltage of the humidity sensor - Potentiometer, RTC to monitor time as the raspberry pi can only get time from the network - MCP7940M, and LED to display alarm.

C. Use Case Diagram

An Use case diagram, see figure 1, depicting the interactions between the user, Blynk application and the system - raspberry pi and additional hardware:

D. Departures/Additions

For testing and output purposes we used two LED's, see 3: one to be an alarm - flashes when alarm is signaled, and another for the DAC output voltage to be visible. The Pi also only keeps track of time via the network, but due to the real time nature of the user requirements we used a RTC to track time. Other wise we kept to user requirements as much as we could.

III. Specification and Design

From the requirements we will move on to the system specification and design including an UML state diagram to describe the main operation of the system and an UML Class diagram to indicate the structuring of our implementation as well as a circuit diagram.

A. State Diagram

An state diagram, see figure 5, depicting the states in which the python program can be. The program starts off sequential then goes into a fork where it is in 4 different states at the same time. It is logging the information, updating the ADC sensor values and DAC output voltage, handling Blynk requests and updating the alarm. When the program closes all of the join again into one thread where it cleans up and exits. The main thread which handles logging of the data can be interrupted and change into a button handling state.

B. Class Diagram

An class diagram, see figure 2, depicting the structure of out python program. The program has four threads: Main Thread, Blynk Thread, ADC/DAC Thread and Alarm Thread. Each thread except Main Thread is a python threading.thread. They program is mostly structured by threads, each thread has its own function to perform: the main thread handles information logging, converts information from ADC threads saved sensor values to readable values, get alarm status from Alarm Thread, outputs logged information to terminal - each readingInterval second, starts the other threads and waits for the other thread to finish before closing the program with a quick clean up if the program needs to be ended as well as handles button interrupts; Blynk Thread handles communication with the blynk application and transfers the logged information from main thread to blynk app; ADC/DAC Thread reads and stores the sensor values as well as updates the DAC output voltage; lastly the Alarm thread check to see when an alarm should be sound, sounds the alarm and also updates the alarm status.

C. Circuit Diagram

A circuit diagram, see figure 3, depicting how communication between the raspberry pi, ADC and its sensors, DAC and alarm is handled as well as how each component is connected to one another. We used two LED's for output signals: one to show when alarm is activated - flashes, and another to show the value of the DAC voltage. The ADC is connected to the Pi via SPI channel 0 CE 0 and the sensors are connected to it via its input channels: the humidity is connected to channel 0, temperature sensor to channel 1 and light sensor to channel 7. The DAC is connected to the Pi via SPI 0 CE 1 and the RTC is connected via I2C1. All hardware is supplied with the Pi's 3.3V Output for power. There are also four buttons: first, from left, - reset (GPIO BCM 17), second: reading interval frequency change (GPIO BCM 27), third: stop (GPIO BCM 22) and last, from left, is the alarm dismiss button (GPIO BCM 4).

IV. Implementation

In this section we are going to give a simplified explanation of the implementation as well as explanation and highlights of important code used. We used Python for out implementation due to the extent of libraries available to us on the platform over C.

A. Overall Implementation

The program is split into 5 parts: Part 0: Setting up GPIO, SPI -DAC/ADC, I2C- RTC, PWM - LED Alarm and threads; Part 1: displaying and saving information log - done by main thread; Part 2: Reading, saving sensor values and updating DAC voltage, Part 3: Updating and sounding the Alarm, Part 4: Communication with Blynk App. Parts 0 and 1 is done by the Main thread, all other parts are done in concurrency with the main thread by use of Python threading.Thread's. These part are all split up to insure the best real-time performance is kept. Part 2, reading of ADC's connected sensors, converting to correct values if need be and saving of the values for use by other threads is done in concurrency with other parts this insure values are always up to date and reading in of values aren't delayed by other parts. Part 1, displaying and converting ADC values to string log values is done separately from part 2 as convertion from int to string can be long and output to terminal is also a time consuming process which can delay real-time performance. Part 3 is done in concurrency with the other parts to insure the alarm is triggered as soon as the value of the DAC goes out of range - putting this else where may cause delays in sounding alarm. Part 4 has to be done concurrently as Blynk runs until the end of the program and waits for interrupts from the app - putting this with another part will cause the program to stall.

B. The Main Thread

The main thread is responsible to setup the program before starting all the other threads and finally waiting for the threads to finish before cleaning up and closing the program if needed:

Starting of threads

```
# only run the functions if
if __name__ == "__main__":
    #Create Trheads
    print("Creating_threads...")
    valuesUpdator = threading.Thread(target=updateValues)
    alarm = threading.Thread(target=updateAlarm)
    blynkThread = threading.Thread(target=blynkFunction)

    # make sure the GPIO is stopped correctly
    try:
        #Start Threads
        print("Starting_threads...")
        valuesUpdator.start()

        #Wait for ADC Thread to get first values
        while (not valuesUpdatorIsReady):
            time.sleep(float(readingInterval) / 20.0)

        #Start other threads
        alarm.start()
        blynkThread.start()

        #Ready to display log
        print("Ready...")
        os.system('clear')

        while True:
            main()
```

Here you can see its important for the ADC thread to be ready before starting the other threads to insure the ADC has read values for the other threads to use first.

Notifies program about to close, wait for threads and cleans up:

```
programClosed = True

# wait for threads
valuesUpdator.join()
alarm.join()
blynkThread.join()

# turn off GPIOs
GPIO.cleanup()
```

The setup involves creating the threads and setting up GPIO, SPI for DAC/ADC - used by Part 2 Thread, I2C for RTC - first the main thread updates the time of the RTC using python datetime library then starts the RTC to oscillate and gives it over to Part 2 thread for saving.

ADC Setup, used Adafruit MCP3008 library for ADC communication:

```
# SPI0 ADC pins
MOSI = 10
MISO = 9
CLK = 11
CS = 8
adc = Adafruit_MCP3008.MCP3008(clk=CLK, cs=CS, mosi=MOSI, miso=MISO)
```

Then it goes on to convert the ADC Values to readable strings, saves them to a log - for use by Blynk Thread, and outputs the log to terminal. Main Thread repeating function:

```
def displayLoggingInformation():
    global systemTimer
    lastUpdated = systemTimer

    print("{:<15}{:<15}{:<15}{:<15}{:<15}{:<15}{:<15}".format(
        "RTC_Time", "Sys_Timer", "Humidity", "Temp", "Light", "DAC_out", "Alarm"))

    while (not programClosed):  # only continue if parent thread is running
        if (monitoringEnabled):
            if (systemTimer - lastUpdated > readingInterval - 0.1):
                lastUpdated = systemTimer
                loggingInformationLine = getCurrentLoggingInformation()
                # print out current logging information line
                print("{:<15}{:<15}{:<15}{:<15}{:<15}{:<15}{:<15}".format(
                    loggingInformationLine[0],
                    loggingInformationLine[1],
                    loggingInformationLine[2],
                    loggingInformationLine[3],
                    loggingInformationLine[4],
                    loggingInformationLine[5],
                    loggingInformationLine[6]
                ))
        time.sleep(float(readingInterval) / 5.0)
```

Here you can see the program only logs if the last log was taken less than the reading interval ago, but also sleeps for a 5th of the reading interval - this is to save CPU time and allow other thread CPU time, it sleeps for a 5th of the reading interval so that is doesn't miss the interval.

C. The ADC/DAC Thread

This thread only reads in and saved the ADC values and outputs the DAC value:

```
values["rtcTime"] = (hrs * 3600 + min * 60 + sec)

            systemTimer = values["rtcTime"] - (startHour * 3600 + startMin * 60 + startSec)

            values["humidity"] = getADCValue(potentiometer) * (3.3 / 1023)
            values["temp"] = ((getADCValue(temperatureSensor) * (3.3 / 1023)) - V0) / Tc
            values["light"] = getADCValue(lightSensor)
            values["dacOut"] = (values["light"] / 1023.0) * values["humidity"]
            writeToDac(values["dacOut"]);

            valuesUpdatorIsReady = True
        time.sleep(float(readingInterval) / 10.0)
```

Here you can see it get the value from adc, converts if necessary, and saves it - nothing else done to insure real time performance. It sleeps to save CPU time and give other threads CPU time and also for 10th of the reading interval to insure it doesn't miss the interval and updates faster than the rest of the threads to insure real time values.

D. Conclusion of Implementation

Only a high level over view is described of the program implementation and most important code snippets shown to keep report brief. See here for full code: Github Repo Link

V. Validation and Performance

In this section we explain how we validated our system and the performance outcome of the system.

A. Validation

To validate the sensor we used the following tests: Testing of the light sensor: To check that the light sensor is working and giving correct output, we covered it with our thumb to check that it output its highest value 1023 and shined a phone torch light to see that it get to 0 or close to zero, lowest value. Testing of the Potentiometer: To test this was easy we moved the sliding all the way up and all the way down and check that the value picked up by the system goes up and down, 0V lowest and 3.3V highest, as the slider moves. Also used a volt meter to check accuracy of intermediate values. Testing of the Timer: We used a phone to check that the time was correct and a stopwatch to check that the system timer was keeping up. Testing of the DAC: The DAC out we calculated the value and check that it match the system's value and use a volt meter on the DAC to see that it co-response. Also increase/decrease Potentiometer and light sensor reading to insure the DAC changed it value accordingly. Testing of Alarm: To test the alarm we decrease the threshold to 3 seconds instead of three minutes then changed the DAC out value up and down, below and above the thresholds to insure the alarm goes on when it should and stays off when it should. For the timer we dismissed the alarm in different time interval: dismissed alarm straight before previous dismissal to see that it doesn't go on till 3 seconds have passed and also dismissed more than 3 seconds apart to insure it does go on immediately again. Testing of Blynk App: We tested the Blynk mobile application by checking that it connects to the system and displays the same values as the terminal, see figure 4

B. Performance

The system performed as expected due to the amount of concurrency put in. It responds immediately to change, within the interval. Also no part of the system effect another, due to concurrency, and the system keeps up with real time. It is also very responsive: immediately responses to buttons and isn't effected by debouncing. The alarm also triggers immediately when the DAC out value goes below or above thresholds and dismisses immediately when dismiss button is pressed. Overall we are very pleased with the performance of the system.

VI. Conclusion

In conclusion we met all the user requirements whilst keeping the system within real-time performance without any issue. The system keeps track of the changing environment and provides a lot of functionality at high performance. The system was found to be successful in terms of passing all the tests during the validation process. The hardware components are relatively cheap making it a potentially useful product in terms of affordability and follows a simple set up process. This system was also intended for use in a green house but can be used in many other places: for instance in solar energy applications one can switch to utility power when sun is set and use solar panels on a warm day.

VII. Figures

Fig. 1: Use Case Diagram

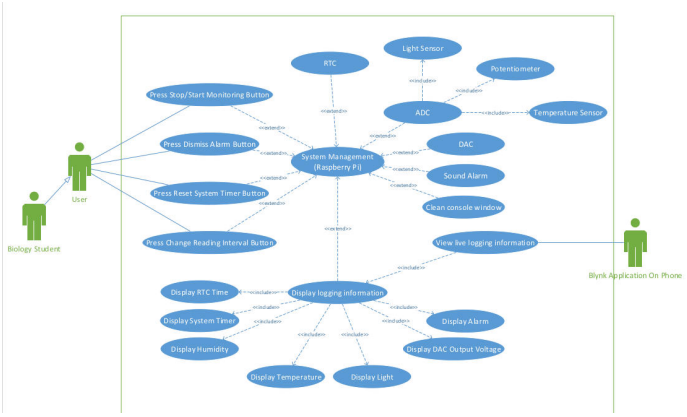

Fig. 2: Class Diagram

Diagram.jpg

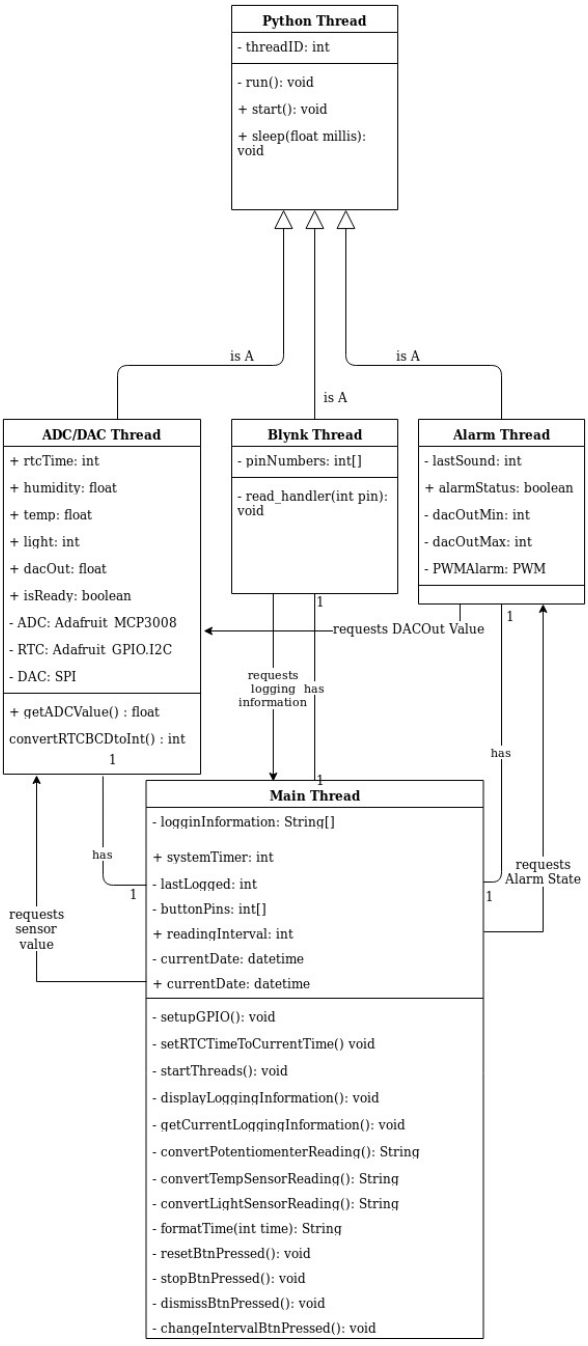

Fig. 3: Circuit Diagram

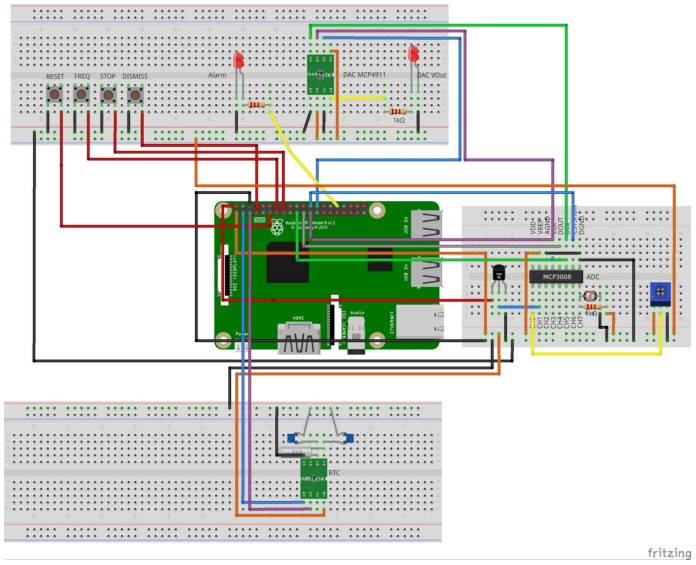

Fig. 4: Blynk App Screenshot

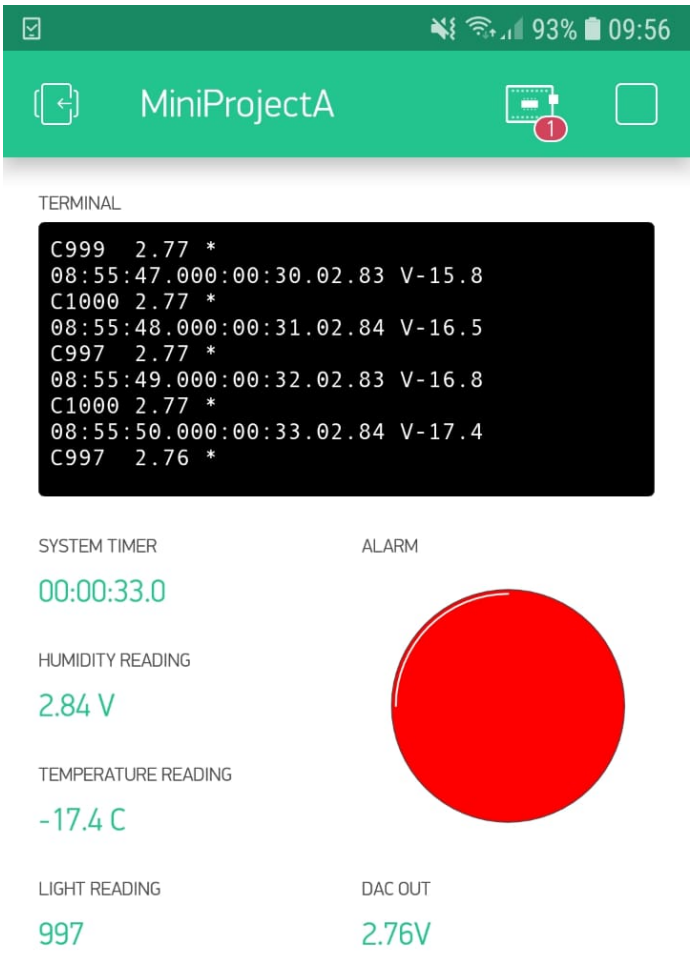

Fig. 5: State Diagram