Project A - Greenhouse Monitor

David Moore[†] and Keagan Chasenski[‡]
EEE3096S Class of 2019
University of Cape Town
South Africa
[†]MRXDAV015 [‡]CHSKEA001

***Abstract*—A design and implementation project of an environment logger for a private Greenhouse. Using specific sensors and inputs the system will control light and humidity inside while reporting environmental factors and system time to a Blynk App for user control.**

I. Introduction

This report is aimed to explain and prove the workings of the Project in conjunction with a formal presentation and demonstration. It will follow the structure stated in the Project brief, where the design choices will be explained below. A UML Use Case and State Charts will be provided in the *Requirements* section to further substantiate these. In the *Design and Specification* section a class diagram will further break down the code implementation of the project. A circuit diagram will be provided to show all the connections of the final circuit. An *Implementation* section will show and explain all important code snippets, followed by a *Validation and Performance* section where the working system will be shown with examples of test cases passed and debugging. Finally a *Conclusion* will end off the report with final words and affirmations.

Using a Raspberry pi 3B+ we will monitor the time of day, time since the system has been running (both using a RTC we will set up on the Pi) , light levels (Using a LDR), temperature (Temp sensor MD9700), and humidity for a Biology student's greenhouse. To mimic the device that monitors humidity we will use a potentiometer to get an analog voltage between 0 and 3.3V.

Through a computer in the greenhouse the Raspberry Pi will be accessed through terminal. However, the data will also be monitored remotely through an app we created using Blynk.

The Pi will output a voltage calculated as follows for the custom recording equipment provided in the greenhouse:

$$Vout = \frac{LightReading}{1023} \cdot HumidityVoltageReading \quad (1)$$

For this voltage, triggers will be set to cause an alarm to sound should the output voltage exceed 2.65V or drop below 0.65V. For this alarm, a button will then be added to dismiss the alarm. For sanity sake a requirement has been made that the alarm will not sound more than once in a 3 min period. is dismissed.

II. Requirements

With the design stated, the following components were needed:

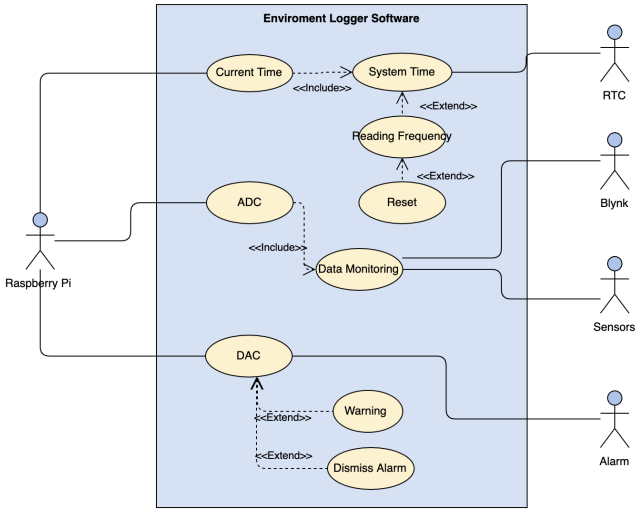
A. Hardware

- Raspberry Pi 3B+
- MCP4911 DAC
- MCp7940M RTC
- MCP3008 ADC
- MCP9700A Thermistor
- Push Buttons
- LDR
- Potentiometer
- Alarm
- Blynk

B. Implementation

With the project brief being very descriptive and the outcomes clear, we did not vary from the initial plan, the use case diagram below shows a clear implementation of the Pi control software for the logger.

Fig. 1: Use case UML for the Project



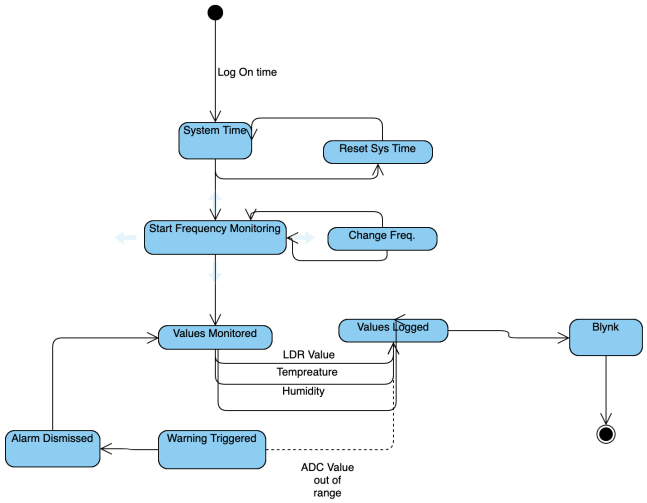
III. Specification and Design

To further break down and specify the project state and class diagrams have been included to show a detailed interaction code and state for the project to run.

A. State Chart

The state chart shows the main operation of the project and the flow of states needed to successfully implement the system.

Fig. 2: State Chart UML for System



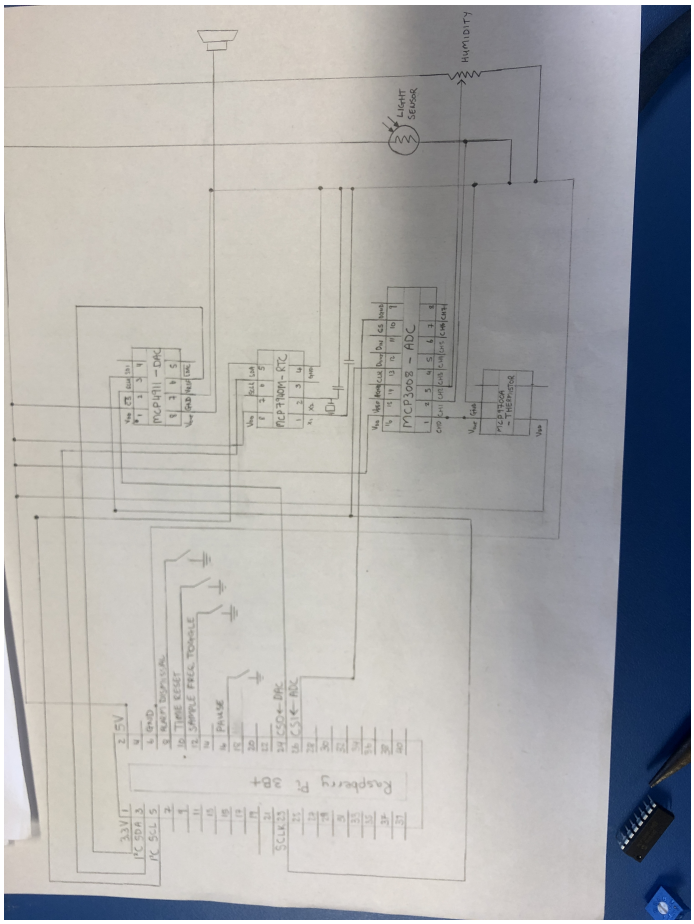
B. Class Diagram

The class diagram shows the Linking of the two files used to run the system. For simplicity in development we only used two class files, and included in those are all the necessary methods. In the class diagram the important methods have been included. The main.c is the C code that runs the system and the main.h file is the header file that contains all of the C values and methods needed.

C. Circuit Diagram

With the specific Pin lay out of the DAC,ADC,RTC and Temperature sensors, the circuit diagram needed to be draw by hand so all the necessary pin connections can be shown.

Fig. 3: Circuit Diagram draw by hand



IV. Implementation

An I2C connection was made to connect to the RTC to the pi. C code was then implemented to calibrate the correct local time using the ¡time.h¿ library. With this C code implementation, the following was able to be done:
A global variable (sec) was used to keep track of the seconds passing by retrieving hex values from the RTC, to observe active system time. This variable was also able to reset at a moment's notice.

Sampling the data from the ADC could be done to the specification of 1, 2 or 5 seconds required thanks to sec. This ensured that the appropriate values stayed consistent and were displayed.

Sec was also able to ensure that the alarm didn't go off more than once in the allowed 3-minute interval

Fig. 4: Time function using sec counter

```c
void *fetchTime(void *arg)
{
    int *fetchingTime = arg;
    prevSecs = hexCompensation(wiringPiI2CReadReg8(RTC, SEC));
    int secs;
    int min;
    int hours;
    while(true)
    {
        secs = hexCompensation(wiringPiI2CReadReg8(RTC, SEC));
        min = hexCompensation(wiringPiI2CReadReg8(RTC, MIN));
        hours = hexCompensation(wiringPiI2CReadReg8(RTC, HOUR));

        if(secs > prevSecs)
        {
            prevSecs = secs;
            sec ++;
            printf("%d:%d:%d\n", hours, min, secs);
        }
    }

    pthread_exit(NULL);
}
```

The ADC was connected to the pi through an SPI connection. A Light Dependent Resistor (LDR), Thermistor and a potentiometer mimicking as the humidity was connected to the ADC. From there, a thread was created to sample the aforementioned analogue signals from the ADC and converted them to digital values that could be manipulated with the C code. These values were displayed to the user at either 1, 2 or 5 second intervals. A total voltage was then calculated

Fig. 5: Main while Loop for the system to continuously run

```c
while(true)
{
    if(sec>prevSecs)
    {
        if(resume == true)
        {
            printf("Active time: %d seconds\n", sec);
        }
    }

    if(sec%SF == 0)
    {
        lightIntensity = readings[0];
        humidity = readings[1];
        temperature = readings[2];
        printf("Light Intensity: %d | Humidity: %d | Temperature: %d \n",lightIntensity,humidity,temperature);
        voltage = (1/1023)*lightIntensity*humidity;
        if(voltage > 2.65 || voltage < 0.65)
        {
            if(sec - TSLA > 180 || firstChance == true)
            {
                alarm = true;
                while(alarm == true)
                {
                    playAlarm();
                    delay(6000);
                }

            }
    }
}
```

Fig. 6: Creating the threads

```c
//creating threads
pthread_t readRTC, readADC;
pthread_create(&readRTC, NULL, fetchTime, sec);
pthread_create(&readADC, NULL, fetchADC, readings);
phtread_join(readRTC, NULL);
pthread_join(readADC, NULL);
```

Fig. 7: ADC Thread creation

```c
void *fetchADC(void *arg)
{
    int buffer[] = arg;

    while(true)
    {
        //read from ADC
        buffer[0] = wiringPiSPIDataRW (ADC_channel, 0b10000000, 1);
        buffer[1] = wiringPiSPIDataRW (ADC_channel, 0b10010000, 1);
        buffer[2] = wiringPiSPIDataRW (ADC_channel, 0b10100000, 1);
        //update appropriate values
    }

}
```

Fig. 8: ADC and DAC init

```c
void initSPI()
{
    DAC = wiringPiSPISetup(0, 500000);
    ADC = wiringPiSPISetup(1, 10000);
}
```

The DAC was connected to the pi via an SPI connection. The DAC was used to play the alarm tone when Vout exceeded its threshold values. The alarm sound file is loaded into an array at the beginning of the program so that it is ready to play immediately when the alarm needs to go off.

Fig. 9: Initializing the Alarm function

```c
void initAlarm(FILE *filePointer)
{
    char ch;
    FILE *filePointer;
    printf("%s\n", FILENAME);
    filePointer = fopen(FILENAME, "r"); // read mode

    if (filePointer == NULL)
    {
        perror("Error while opening the file.\n");
        exit(EXIT_FAILURE);
    }
    //load sound file contents into an array
    int i = 0;
    while(ch != EOF)
    {
        ch = fgetc(filePointer);
        soundFile[i] = ch;
        i++;
    }

    fclose(filePointer);
}
```

Fig. 10: Function call for Alarm to sound

```
void playAlarm()
{
      TSLA = sec;
      firstChance = false;
      printf("*ALARM*\n");
      //play audio through DAC
      for(int i = 0; i < 172284; i++)
      {
         wiringPiSPIDataRW(DAC_channel,soundFile[i],1);
      }
}
```

V. Validation and Performance

The compilation of the main.c file takes a while because the program needs to load in 172284 bytes of data into an array for the alarm sound file. The startup of the program also takes a couple of seconds before it starts displaying anything as the pi needs to make the appropriate SPI and I2C connections and to start up the threads. But once the programmed has warmed up, everything runs smoothly and acts in real time. The system isn't very power efficient and isn't recommended to run on a small battery system, but a trade-off needs to be made when providing effectiveness over power efficiency.

VI. Conclusion

Working on this project provided a viable solution to the design task and we feel that the implementation works well, with the provided components. Linking to Blynk provides much needed functionality however technical knowledge with a Pi and terminal will be needed to maintain the system.

References