Mini-Project A: Environment Logger

Devon Pugin[†] and Callum Tilbury[‡]
EEE3096S Class of 2019
University of Cape Town
South Africa
[†]PGNDEV001 [‡]TLBCAL002

Contents

List of Figures

Plagiarism Declaration

1) We know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2) We have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this project report from the work(s) of other people, has been attributed and has been cited and referenced.
3) This project report is the collective work of the team members of this group.
4) We have not allowed, and will not allow, anyone to copy our work with the intention of passing it off as their own work or part thereof.

Signed:

Devon Pugin

Devon Pugin

Callum Tilbury

Callum Tilbury

I. Introduction

The aim of this project was to prototype a simple environment logger – one that could record ambient temperature, humidity, and light, and report these values to a user. Two interfaces were designed: one to be viewed in the Raspberry Pi terminal, and another on a phone running a simple Blynk application.

Since the project was a prototype, it was built on a breadboard. This enabled fast debugging, and easy restructuring when required. Moreover, for convenience, rather than using an actual humidity sensor, a simple potentiometer was used. That way, all three 'input' parameters of the system could easily be altered for testing (since light and temperature were already fairly easy to adjust).

All code for the Raspberry Pi was written in a single Python 3 script. Python is a massively popular, simple language that has great support online, as well as extensive libraries for GPIO interfacing, SPI and I2C communications, and Blynk. Due to the relative straightforward nature of the project, an object-orientated approach was *not* taken; instead, global variables were simply shared around the program. This is not necessarily a scalable approach, but is justifiable for the context of the project.

This report details key factors involved in the design of this project. After this brief introduction, the project's requirements are presented, primarily using a UML Use Case diagram. Thereafter, a high-level design overview is given – showing important components and their respective interconnections. A state chart, deployment diagram, and schematic are all given here. Moving on, sections of important code are provided, and discussed briefly. Subsequently, the system is evaluated and validated, and the performance of the resulting prototype is considered. Finally, some concluding remarks are made.

II. Requirements

Figure 1 shows a UML Use-Case Diagram which describes the requirements of the system.

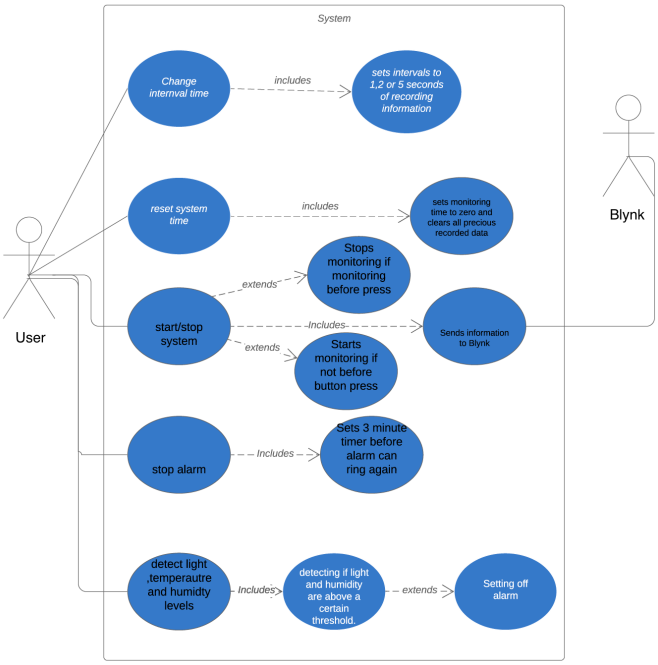

Fig. 1: UML Use-Case Diagram for System

There are some key points that must be made which are not immediately obvious from the Use-Case Diagram alone. They are listed below:

- The 'interval time' button iterates through the values: [1, 2, 5] seconds. If the time is set to 5 seconds and the button is pressed, it moves back to 1 second, and so on.
- The code is scalable to add/remove as many interval times as desired
- Information is sent to Blynk at the same rate at which the text is printed to the terminal.
- Information is sent to Blynk on a separate thread as to prevent network requests blocking the main thread.
- Resetting the system time does not affect any alarm functionality.
- Starting/stopping the system begins/ends the thread on which the ADC reads the data.
- When the alarm is triggered, the system starts a 1kHz PWM signal on a pin connected to a buzzer. The alarm is thus monotonic.
- Once an alarm is dismissed, it cannot sound for another 3 minutes. This period begins when the alarm first activates, not when it is dismissed.
- The Real-Time Clock (RTC) module is not shown here. This is because there is nothing in the Python code that is causing the RTC to tick; instead, a kernel driver is used such that the Pi uses the RTC time as its own time.

III. Specification & Design

A. UML State Chart

The state chart in figure 2 describes an overview of the main functionality of the system. Each depicted state or connection between states fulfils a particular requirement given in section II.

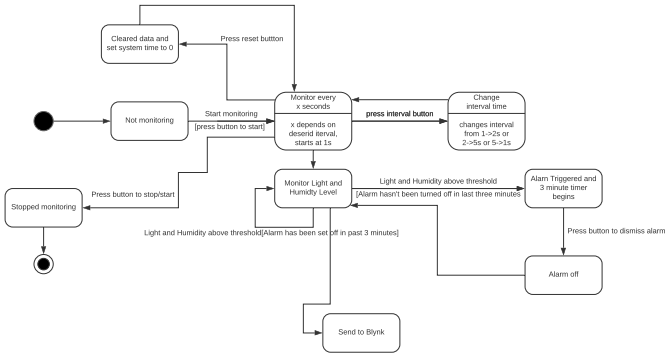

Fig. 2: UML State Chart for System

B. Deployment Diagram

To supplement the state chart shown above, a deployment diagram is shown in figure 3. This enables one to get a high-level, broad understanding of the system components – specifically important software and hardware, and how they interact.

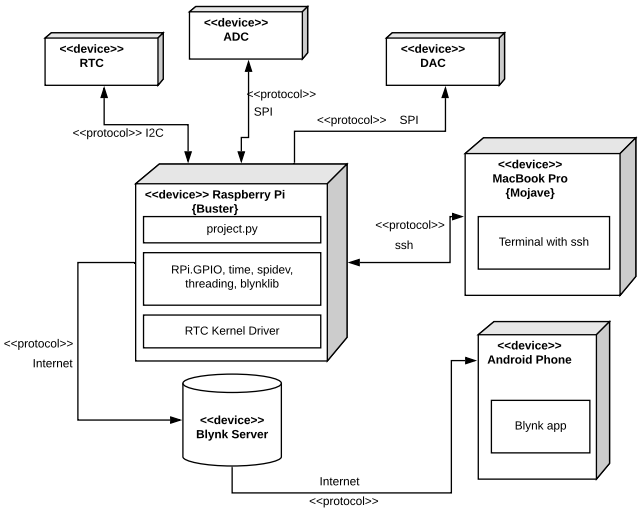

Fig. 3: Deployment Diagram for System

C. Circuit Diagram

Figure 4 shows the full circuit schematic for the project. This diagram entails all that was built on the breadboard, and the respective connections to the Pi. As a short aside, when comparing the schematic to the deployment diagram in figure 3, it is clear that there are additional connections made off the breadboard – these are network related connections between the various devices.

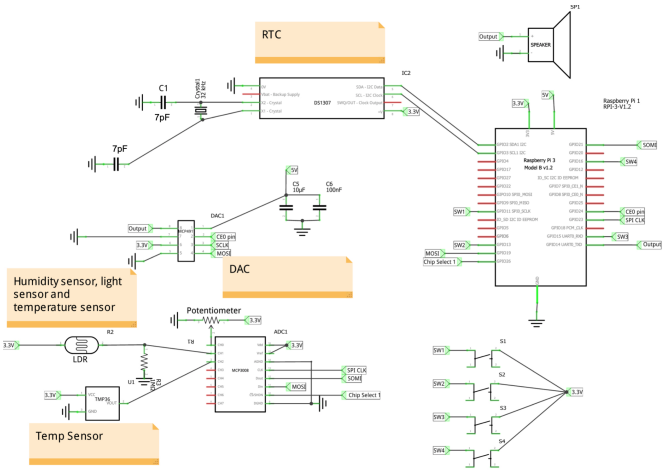

Fig. 4: Schematic for System

IV. Implementation

This section presents some key code[1] snippets that are essential to the project.

A. Reading from the ADC

```
# Read a [0, 1023] value from a chosen ADC channel
def ADC_read(chan):
  adc = spiADC.xfer2([1,8 + chan << 4,0])
  data = ((adc[1] & 3) << 8) + adc[2]
  return data
```

This code simply performs a read of the ADC registers, for a desired channel. The logic used is based off the register positions of the MCP3008 chip.

B. Writing to the DAC

```
# Write a [0,1023] value to the DAC for [0, 3.3]V
def DAC_write(value):
  lowByte = value << 2 & 0b11111100
  highByte = ((value >> 6) & 0xff) | 0b00110000
  spiDAC.xfer2([highByte, lowByte])
```

This code writes a 10-bit number to the DAC for conversion to a voltage. The bitshifts and logic operations manipulate the data such that it suits the DAC's register structure, as shown in figure 5.

REGISTER 5-2: WRITE COMMAND REGISTER FOR MCP4911 (10-BIT DAC)

W-x	W-x	W-x	W-0	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x
0	BUF	$\overline{GA}$	$\overline{SHDN}$	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	x	x
bit 15															bit 0

Fig. 5: Write register structure for the DAC

[1]Actually pseudocode as certain variable names and function calls have been simplified

C. Voltage Output with Alarm

```
# Function to put desired voltage on DAC output;
# also signals the alarm
def output_voltage(voltage):
  global alarm, alarm_on, last_alarm_time_secs,
    output_Volts
  output_Volts = voltage
  if(voltage outside threshold):
    if (currentTime())
      > (last_alarm_time_secs + 3*60)):
      alarm_on = True
      alarm.start(50) # 50% duty cycle
      last_alarm_time_secs = currentTime()
  output_val = int((voltage/3.3)*(2**10-1))
  DAC_write(output_val)
```

This code is called each time the ADC has finished reading the three inputs. It writes the voltage value to the DAC, and most importantly, starts the alarm if appropriate. Notice the two checks in place for turning on the alarm: both checking if the voltage is outside of the threshold, and checking if it has been over 3 minutes since the last alarm.

D. Print to Terminal

```
# Function to print values to monitor
def print_monitor():
  curr_time = time.localtime()

  monitor_time = currTime
  monitor_sys_time =
      secs_to_string(currTime-startTime)
  monitor_humidity =
      (humidity_val*3.3/1023)
  monitor_temp = convert_temp
      (temp_val + ADC_TEMP_CORRECTION)
  monitor_light = light_val
  monitor_outputV = output_Volts

  # Terminal:
  monitor_string = ( ... )
  print(monitor_string)
```

The purpose of the above code is to display the log information in the terminal of whatever is running the Python script. The global variables storing log information are loaded into the function, and are manipulated into the correct format. The appropriate output string is built and formatted, and is then printed. This occurs every time the display is printed.

E. Push to Blynk

```
blynk.virtual_write(BLYNK_TIME, monitor_time)
blynk.virtual_write(BLYNK_HUM, monitor_humidity)
blynk.virtual_write(BLYNK_TEMP, ...)
blynk.virtual_write(BLYNK_LIGHT, ...)
blynk.virtual_write(BLYNK_ALARM, 255)
blynk.set_property(BLYNK_ALARM, 'color', ...)
blynk.virtual_write(BLYNK_OUTPUTV, ...)
```

This code actually exists within the print_monitor() code from above, but is presented separately for clarity. It uses a convenient Blynk library for Python, and requires a simple set-up of virtual pins on the Blynk app. Then, once setup, those pins can be addressed easily using the virtual_write() function.

BLYNK_TIME, etc. are simply the constants holding the virtual pin numbers for the various parameters.

V. Validation & Performance

It is fundamentally important to validate the results of the project – both on a system level, as well as each component individually.

A. Overall Performance

On the whole, the project performed well, and met all the initial specifications. All of the desired functionality was included, and thorough system testing was performed – including arbitrary actions and alternative orders of operation. The codebase is believed to be reliable enough to handle anomalies appropriately.

B. Analog-to-Digital Converter (ADC)

To test the accuracy of the ADC, the potentiometer (simulation of humidity) was varied through samples of its full range, and the resulting digital value – converted to Volts – was compared to the actual input voltage. The results of this experiment are shown in the graph in figure 6.

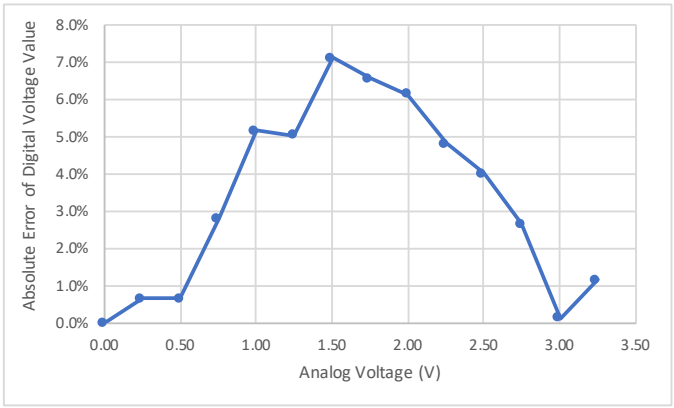

Fig. 6: Absolute error achieved on ADC

Notice that in the central range of voltages, the ADC became fairly inaccurate – with an absolute error as high as 7%.

The effect of the ADC's inaccuracy was most noticeable in the temperature sensor reading. For example, consider the situation at room temperature ($25°C$). The relationship between the ambient temperature and the sensor's output voltage [] is given by:

$$V_{out} = T_A \cdot T_C + V_{0°C}$$

where T_A is the ambient temperature, and, for the MCP9700A, $T_C = 0.01\frac{V}{°C}$; $V_{0°C} = 0.5V$. Hence, the desired output voltage is $V_{out} = 0.75V$. This voltage value was indeed observed on a multimeter at room temperature. However, the ADC error causes the digital value to be recorded as 198 – that is $\frac{198}{1023} \times 3.3V = 0.64V$. Now, working backwards, instead of reading the correct temperature of $25°C$, the temperature is reported as $14°C$ on the digital device – which is a big error for this context. This problem is aggravated by the fact that most of the temperature readings will be towards the middle of the supply's range. This is exactly where the ADC performs badly.

To rectify the error, an offset can be applied to the read ADC value. Of course, the error is clearly non-linear (see figure 6) over the input voltage range, and hence a linear shifting is not a complete solution. However, within the context of an environment logger, it is these so-callled 'middle' values that matter most (it is unlikely that a greenhouse will reach temperatures of greater than $100°C$).

Note that the ADC inaccuracy is not as severe for the light reading (from the LDR), as this setup needs to be calibrated either way. The effect on the humidity readings depends on the sensor used.

C. Digital-to-Analog Converter (DAC)

The DAC, on the other hand, performed well. Throughout the range of output voltages [0, 3.3] V, the DAC output was always within ± 0.01V of the desired output.

D. Threading Performance

Two threads were created within the project code – one to read the ADC values, and another to push data to the Blynk server. Both of these performed well, and it resulted in a smooth user interaction experience. The data in Blynk was never more than one sample behind the Pi itself – this is acceptable given the context.

VI. Conclusion

The prototyping of this system went well. As to be expected, several problems were encountered – such as the RTC not ticking (incorrect capacitor values), SPI not working (using the wrong SPI overlay), and so on. Fortunately, however, due to the 'breadboard' approach, the circuit was easy to adapt and improve as time went on, and this enabled fast and effective prototyping.

Regarding the system at large, the requirements were met and the project operated smoothly. The sensors seemingly worked well, and decent results were achieved. The ADC – being low-cost – had some substantial errors, and the full effect of this must still be considered. However, with some careful calibration, the system should be accurate enough for the context.

As for turning this *project* into a *product*, there is indeed great potential, though some changes are essential for this potential to come into fruition: of course, the system would need to be attached to a well-designed, compact, printed circuit board (PCB). Moreover, an actual humidity sensor would have to be implemented. The codebase would also have to be reworked into something that is more scalable, and possibly object-orientated (though still in Python). The cost of commercial licensing would have to be considered for the Blynk application, and an improved User Interface would need to be developed. More widgets are required, as a greater locus of control should be given to the user from the Blynk app. Finally, much more extensive testing would have to occur.

Once these changes are made, there is definitely great potential for such a product to succeed in the market. There are plenty people – from hobbyist-gardeners to those who are more qualified – who would value the useful and insightful information such an environment logger can provide. It could be assistive in scientific research (though the sensors and ICs should be upgraded), as well as education (where lower quality is acceptable for a lower cost).