# EECS 3215: Embedded Systems Project

## *Air quality based ventilation*

Faculty: Dr. James A. Smith
York University
Due: March 10, 2020

Student:
Sai Varun Ramavath
Student #: 213506936

# *Contents*

# 1. *INTRODUCTION*

This project aimed to create a proof of concept for an embedded system that would adjust the amount of Total Volatile Organic Compounds (TVOCs) in an enclosed space to healthy levels. The sensor of choice is the CCS811 from Sparkfun which unfortunately failed at maintaining a reliable connection with the LPC802 (details in the *Contingency* section). The chosen alternative was to use the LM75 temperature sensor on the LPC802 to showcase the functionality of the rest of the components, but the context, requirements and all the diagrams will assume the CCS811 is being used, since the LM75 is also an I2C sensor.

# 2. *CONTEXT*

Volatile Organic Compounds, VOCs for short, are all around us. They are emitted from various sources, and in certain concentrations prove to be a major health hazard. In addition, work environments that have high concentrations of VOCs suffer from a decrease in worker productivity and cognitive ability.[1] This system is designed to tackle this issue by implementing a ventilation system that is able to detect TVOCs and begin exhaust systems when a certain threshold is reached.

# 3. *TECHNICAL REQUIREMENTS*

| | |
|---|---|
| Inputs | TVOC (CCS811), RPM (motor encoder), button presses |
| Outputs | Current PPM levels and fan setting on OLED display |
| | PWM signal to motor |
| Functions | Push button adjusts fan between high, low, and automatic modes |
| | Control fan speed dynamically, based on PPM readings |

# 4. *TECHNICAL SPECIFICATIONS*

The technical specifications will talk about *how* the requirements are going to be implemented.

## 4.1 OLED display

The display driver for the OLED of choice (a 128x32 SSD1306) has to be designed under tight constraints given the rather small storage of the LPC802. This driver was written from scratch, and since I have no experience in glyph rendering on displays, I decided to follow a simple approach.

### *4.1.1 How to render characters?*

1. Define a binary bitmap of each ASCII character, since each column contains 8 pixels that are turned on or off based on the value of the corresponding bits in the display's graphical memory (GDDR). This was done using a bitmap generator, and a bitmap to C header conversion application. The output of this application is a 600 element one-byte array for the chosen font size of 8x8 bits.

[1] Piers MacNaughton, Usha Satish, Jose Guillermo Cedeno Laurent, Skye Flanigan, Jose Vallarino, Brent Coull, John D. Spengler, Joseph G. Allen, The impact of working in a green certified building on cognitive function and health, Building and Environment, Volume 114, 2017, Pages 178-186

2. Create a function that evaluates which bytes to send based on a given character, and retrieves them from the bitmap.

#### *4.1.2 String formatting - the buffer*

The buffer is instantiated as a 256 element (128x2 columns) one-byte array since the strings that would be written for this project were not going to be more than 2 lines long. This sizing allowed for line wrapping, while taking half the space of a full screen buffer. Line wrapping is implemented by checking if the total number of characters in the string will overflow the maximum number of pixel columns in the line, and writing a newline character before the word that causes the overflow.

#### *4.1.3 Supplementary functions*

Some functions were made to make it easier to interface with the display, such as setting position of the next written string, clearing the display, turning the display on, off, or in low power mode, and a function to display numbers.

### 4.2 Motor

The CTimer in PWM mode will be used for speed control with a frequency of 100 Hz, since anything faster will not able to deliver enough power for the motor to start.
The PID controller runs on channel 1 of the MRT (reload value equal to period of 2.25 seconds).

#### *4.2.1 PID*

The output from the control block requires the current speed of the motor in order to generate an appropriate PWM duty factor. The duty factor is then multiplied with the desired frequency of the signal to calculate the number of ticks the CTimer counts before an interrupt is generated. Figure 1 shows the block diagram of the system. The display stays on while the PID block is running, and button presses cannot change the fan setting.

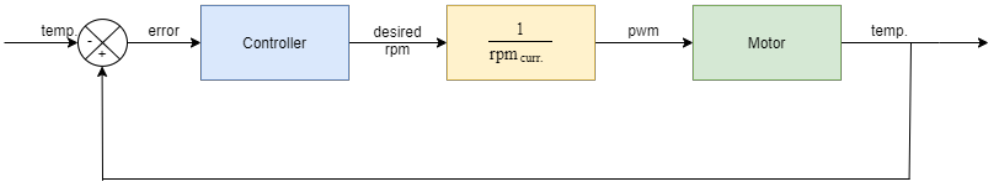



Figure 1: PID block diagram

#### *4.2.2 Push button*

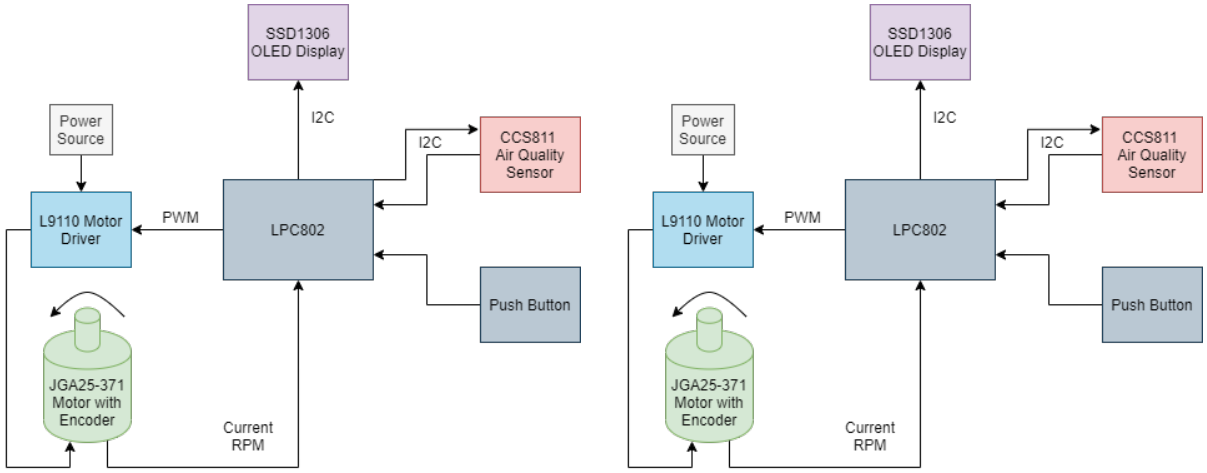
The *User* button on the LPC802 was used - as a form of manual control - and pin interrupts will be enabled on the falling edge to detect button presses. The button allows the user to set LOW, HIGH or AUTO modes for fan speed; AUTO mode turns the fan on when the TVOC reading reaches the threshold value. Key presses turn the display on; the display turns off at the end of a timeout period (around 10 seconds).

### 4.3 CCS811 sensor

The sensor defaults into BOOT mode, therefore the first step in getting data from this sensor is writing to the application verify and application start registers. After these have been written to, the measurement mode is selected to take measurements at 1 Hz, and the sensor readings are taken on channel 0 of the MRT (reload value equal to period of 2 seconds).

## 5. *COMPONENTS*

The hardware components used in this project are shown in Figure 2, followed by the (major) software components in Figure 3.




(a) Architecture (b) Implementation

Figure 2: Hardware components of TVOC ventilator

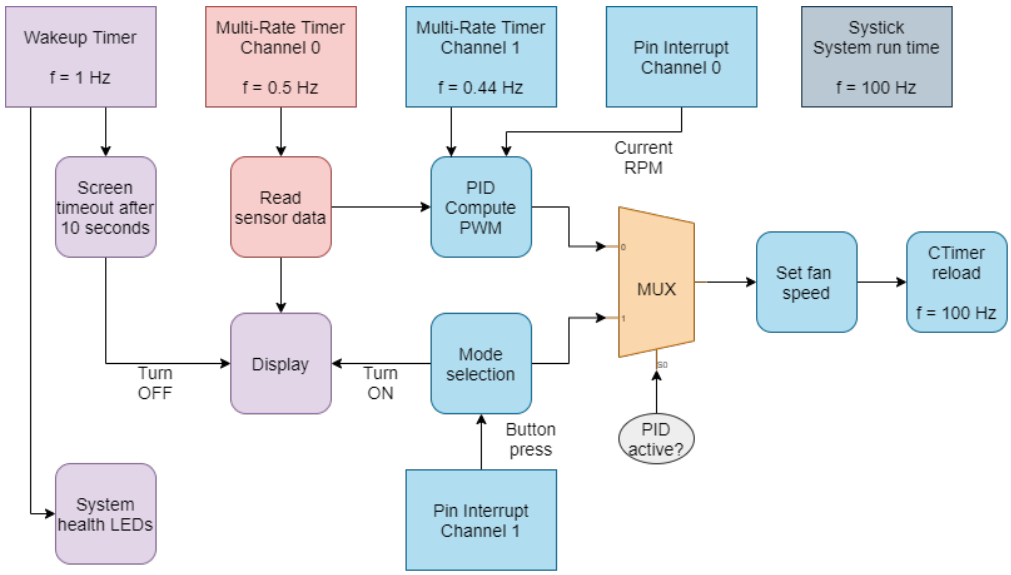



Figure 3: Software architecture of TVOC ventilator

# 6. *PROCEDURE*

## 6.1 Hardware

The hardware connections for this project were relatively straightforward, since the components were purchased as breakout boards, not in their SMD package. The project proposal specified a custom printed case and PCB, but given that I was enforcing self-isolation measures quite strictly, I could not access resources to implement these.

## 6.2 Software

The design process followed for this project was the V Model approach. This model specifies that the developer must first come up with system requirements and specifications (done in the proposal part of this project), and then jump into detailed design and finally coding. These common stops were taken while writing device drivers:

- go through sensor/display/motor data sheet and note down important addresses, write sequences, and timing characteristics,
- expand on the architecture of the system by specifying in further detail what each driver *is* supposed to do, and what it is *not* supposed to do,
- write code and review notes in the case of bugs (there were many).

In some cases, it was possible for me to test out functions (such as the line wrapping that I implemented for the display) in an *Ubuntu* terminal before I implemented it in the driver.

Header files were created for each driver which contained all the #define constants that made code easy to read, and the function prototypes which made it easy to quickly reference a function. C structures were used to store device states and parameters, which made writing to and reading variables quite intuitive; and since the structs were defined as *extern*, these were available in any function as long as the relevant header file was included.

# 7. *TEST*

## 7.1 OLED display

The display driver was tested multiple times throughout development by passing test strings to the *displaystr* function of the driver, and observing the output on the display. The display on, off and low power mode functions were used extensively while turning the system on or off, with good results.

## 7.2 Motor control

Since the LM75 sensor was being used for testing, the PID control block was activated by raising the temperature of the sensor using a heat source (initially by exhaling on the sensor, but this got exhausting) such as a soldering iron. Figure 4 shows the output response: green trace represents temperature (setpoint of 23° C), blue trace is fan speed (RPM), red trace is PID output (RPM).

# 8. *CONTINGENCY*

The contingency options that I had to choose were not on my list of contingencies in the proposal. The backup plan in case I could not get the CCS811 working was to go to an electronics store and pick up a different sensor to detect gases, and I did not anticipate a mismatched motor specification. For the motor, I was able to get the temperature of the LM75 lower by letting the heat dissipate naturally, but the PID block


Figure 4: PID block test run - untuned

still sent out appropriate response to the motor. The next sections are about the CCS811

## 8.1 The problem

I am not able to establish a reliable connection with the I2C for longer than a minute.

## 8.2 Possible reasons

1. The CCS811 is an I2C sensor that requires clock stretching to operate properly, since the on-board IC takes some time to gather data and report that it has data available to send to the primary device (more likely).

2. The CCS811 I got is faulty (less likely).

## 8.3 Attempts to find a solution

1. Introduce blocking delays (even though this is bad practice), but this did not seem to have much of an effect on connection reliability.

2. Try to use the Monitor function (with clock stretching enabled) built into the 802 to take a look at when NACKs were generated from the sensor, but this was quite hard to keep track of, and any solutions that I tried did not seem to have an effect on the connection reliability. An oscilloscope or signal analyzer would have been ideal for this, but I do not have access to one.

3. Try to pull the SCL and SDA pins high (no effect); low (no effect).

4. Try to step through the code in debug mode, which worked quite well, but this is not a permanent or viable solution.

Under normal circumstances, I believe I would have been able to get this sensor working, or replace it with a different sensor. The lesson that I am taking away from this is that I should get better at reading data sheets, since this could have been avoided if I had read the data sheet thoroughly before purchasing the sensor.

## 9. *ADDITIONAL MATERIAL*

The effects of being in an environment with a high TVOC concentration include eye irritation, headache, nausea, fatigue among many others. [2] In a workspace/university/indoor environment with many sources of VOCs, effective detection and ventilation is necessary to ensure that the occupants stay healthy and do not suffer productivity losses.[1]
The PID controller in this system was chosen with the purpose of reducing the overall energy footprint of the system, while effectively fulfilling it's purpose. The interesting thing about VOCs in a given space is that the higher the number of people in the space, the greater the emissions of the VOCs.[3] This fact creates the potential for another way that air quality based ventilation can save energy: a lower number of people in the room will trigger the threshold value of the VOCs less often, thereby saving energy over a conventional ventilator that turns on for x minutes every y hours.

This COVID-19 lockdown has shown us that reducing energy consumption can clear the skies of smog, lakes and rivers of pollutants, and create a better world for us to live in, and we can take one step closer to such a world by reducing our global footprint.

## 10. *CONCLUSION*

The project was a success to me because I was able to achieve most of the requirements.

I have learned some valuable lessons in the design process such as being thorough when reading data sheets, making sure I follow one of the design models laid out in the course, and overall being mindful of the various bugs and issues I had encountered.

[2] "Volatile Organic Compounds' Impact on Indoor Air Quality." EPA, Environmental Protection Agency, 6 Nov. 2017, www.epa.gov/indoor-air-quality-iaq/volatile-organic-compounds-impact-indoor-air-quality.

[3] "HVAC Energy Consumption In Commercial Buildings: 4 Ways To Save " Iota Communications, Inc." Iota Communications, Inc., 18 July 2019, www.iotacommunications.com/blog/hvac-energy-consumption-in-commercial-buildings/.