Smart Hiking Bladder

ECE 445 Fall 2016

Individual Progress Report

Shuchen Song

TA: Kexin Hui

October 31, 2016

Contents

1 Introduction

I am focusing on the software aspect related to the project. Specifically, one important component of the software is the program run on the microcontroller. Since our projects is using flow rate sensor, pressure sensor to measure the input, and using LED, Bluetooth as communication or representation to users, the microcontroller mainly involves many interactions between these sensors and modules. Generally, the microcontroller will take the input from the sensors, do the calculations and transfer data for modules for displaying. The Bluetooth module will also responsible for communication with a phone App, which will tell the distance have already walked and estimation of the water left in the bladder with how far the user could go before it run out.

Currently, most of the bridges between the microcontroller and the hardware are constructed, individually. Therefore, some of the unit tests are also performed for different parts. The LED could display the desired text, flow sensor is reading the input, although with errors, and Bluetooth can communicate with the phone. However, since the fact that the USB serial debugger and the Bluetooth module are using the serial port on the microcontroller at the same time, it would be more difficult to debug if all of the parts are integrated at this stage. Therefore, the integration tests will be performed after the sensors are calibrated, and the USB debugger will not be intensively used.

2 Design

2.1 Microcontroller

In this section some general design decisions about programs on the microcontroller is discussed. This mainly involves two parts, the connection of the microcontroller and the capability of it.

For connection, the problem was that there are ways to connect the microcontroller and programming it. For convenience, we are using the Arduino bootloader on the microcontroller and then update the Arduino programs. This decision is made since the richness of resources for Arduino, and the easiness to program.

For the capability, the concern mainly lies within the interruption. Since the flow sensor is generating pulses as the output, the microcontroller should be able to catch every pulse in order to ensure the first step toward an accurate measurement. There are two ways to detect the pulses, the details will be discussed in the flow sensor section, but here we need to see whether the microcontroller has the capability to take the inputs through interruptions. After measurements, the microcontroller seems to guaranteed accurate catches when the frequency is lower than 16k Hz, which is safe considering the output frequency from the flow sensor. Therefore, from here we know that interrupt based design is possible, and the tentative flow chart is changed correspondingly, as shown in section 2.5, Figure 4.

2.2 LED Module

This module is relatively straight forward. The library provided with the Arduino takes care most of the jobs, and the custom part is the wiring. To not to be conflict with the Bluetooth module, which requires the serial communication, the LED module is using digital pins without the serial functionalities.

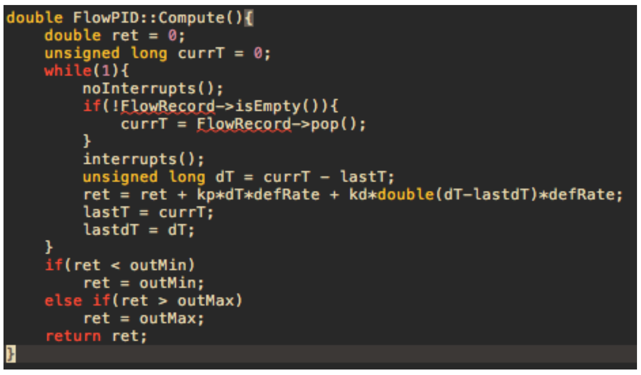
2.3 Bluetooth Module

This module is relatively straight forward as well. The communication between the microcontroller is done by the serial pins. Since this is the only module requires the serial communication in our project, we could use the library comes with the Arduino directly. We just need to decide that the two serial pins are reserved for this module.

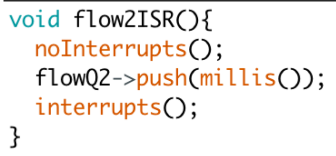
2.4 Flow Sensor

The flow sensor is outputting pulses when some amount of water has flow through it, causing the internal fan rotated a certain angle and activated a hall effect sensor. The problem here is that since there are potential frictions for the fan to rotate, and the fact that fan will be experiencing different forces with different water flow rate, it is hard to determine what is the amount of water flow through it within certain number of pulses. Also, due to the nature of this output signal, we need to determine an effective program scheme which could catch all the input and do the calculations with as less as possible logics. Therefore, the problem is mainly divided into two parts, one is the method to catch the input, and the other one is how to compensate the error to get a water flow amount as accurate as possible.

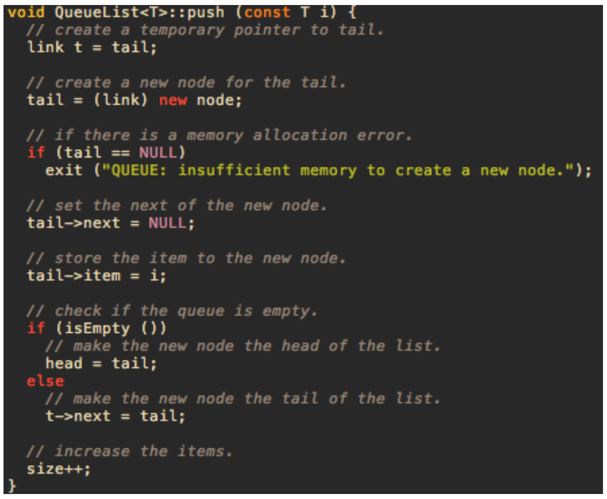
For the measure method, some of the foundations is built in section 2.1, when talking about the interrupt capabilities of the microcontroller. It is proved that this is a valid



```
double FlowPID::Compute(){
    double ret = 0;
    unsigned long currT = 0;
    while(1){
        noInterrupts();
        if(!FlowRecord->isEmpty()){
            currT = FlowRecord->pop();
        }
        interrupts();
        unsigned long dT = currT - lastT;
        ret = ret + kp*dT*defRate + kd*double(dT-lastdT)*defRate;
        lastT = currT;
        lastdT = dT;
    }
    if(ret < outMin)
        ret = outMin;
    else if(ret > outMax)
        ret = outMax;
    return ret;
}
```



```
void flow2ISR(){
  noInterrupts();
  flowQ2->push(millis());
  interrupts();
}
```

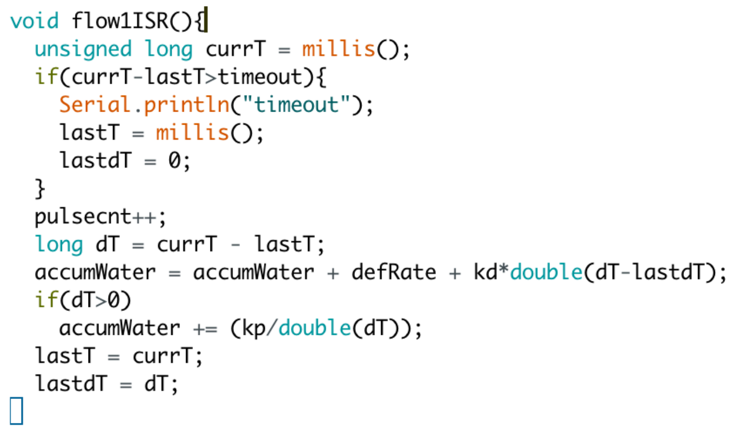


```
void QueueList<T>::push (const T i) {
  // create a temporary pointer to tail.
  link t = tail;

  // create a new node for the tail.
  tail = (link) new node;

  // if there is a memory allocation error.
  if (tail == NULL)
    exit ("QUEUE: insufficient memory to create a new node.");

  // set the next of the new node.
  tail->next = NULL;

  // store the item to the new node.
  tail->item = i;

  // check if the queue is empty.
  if (isEmpty ())
    // make the new node the head of the list.
    head = tail;
  else
    // make the new node the tail of the list.
    t->next = tail;

  // increase the items.
  size++;
}
```

Figure 1 Core Code for Flow Sensor Design Stage 1. Code on right is from library QueueList [2]

design, and it was put into test. In the first stage, the I am trying to use a design similar to the one used in the Linux [1], where the interrupt handler is separated into the top and bottom halves. The top halve is responsible for clear the interrupt and some emergency operations, then let the handler return quickly. After that the bottom half will be executed and the computation intensive code will run. However, in Arduino there is no tasklets or ideas of multitasking, thus, I have to use a data structure similar to the blocking queue in Java to implement this functionality. Some of the core code is show in Figure 1. Through this code it shows the critical section handlings and the two halves architecture. However, the results show that the clear interrupts and set interrupt functions are influencing the clock functionality. The reason is unknown, even after searching documentations on the web. But personally I suspect that the Arduino is using the clock interruptions to determine the system time.

Therefore there is changes to this design, moving all code in the FlowPID::Compute() to the interruption handler itself. I will just call it stage two. Surprisingly, the code works



```
void flow1ISR(){
  unsigned long currT = millis();
  if(currT-lastT>timeout){
    Serial.println("timeout");
    lastT = millis();
    lastdT = 0;
  }
  pulsecnt++;
  long dT = currT - lastT;
  accumWater = accumWater + defRate + kd*double(dT-lastdT);
  if(dT>0)
    accumWater += (kp/double(dT));
  lastT = currT;
  lastdT = dT;
}
```

Figure 2 Core Code for Flow Sensor Design Stage 2

without any trouble and is able to catch all of the interrupts. The core code is shown in Figure 2.

$$u(t) = K_p e(t) + K_i \int_0^t e(t)\,dt + K_d \frac{de(t)}{dt}$$

Figure 3 PID formula [4]

For the design of calculating the measurement, I am using an approach similar to the PID control algorithm. In PID, referenced in Figure 3, the e(t) and de(t)/dt are mainly used to compensate the errors. This approach is used since that from reasoning, we could see that due to the friction, the flow sensor should behave differently when it just starts and when it about to stop. Because of this reasoning, the flow sensor's calculation is based on a steady rate, which every pulse will add on, a compensation related to the time difference between two pulses, and the differences between two differences between two pulses. Also, after experiments we discover that when the water flow more quickly, the flow sensor tends to output a water flow lesser than the real amount. Therefore, we are adding the inversed difference. Overall, several definitions are $\mathrm{dt} = \mathrm{currT} - \mathrm{lastT}, \mathrm{ddt} = \mathrm{currdT} - \mathrm{lastdT}, \mathrm{defRate} = \mathrm{const.}$ And the final equation is: $\mathrm{accumWater} = \mathrm{accumWater} + \mathrm{defRate} + \frac{\mathrm{kp}}{\mathrm{dT}} + kd * ddT.$

2.5 Overall Software Flowchart

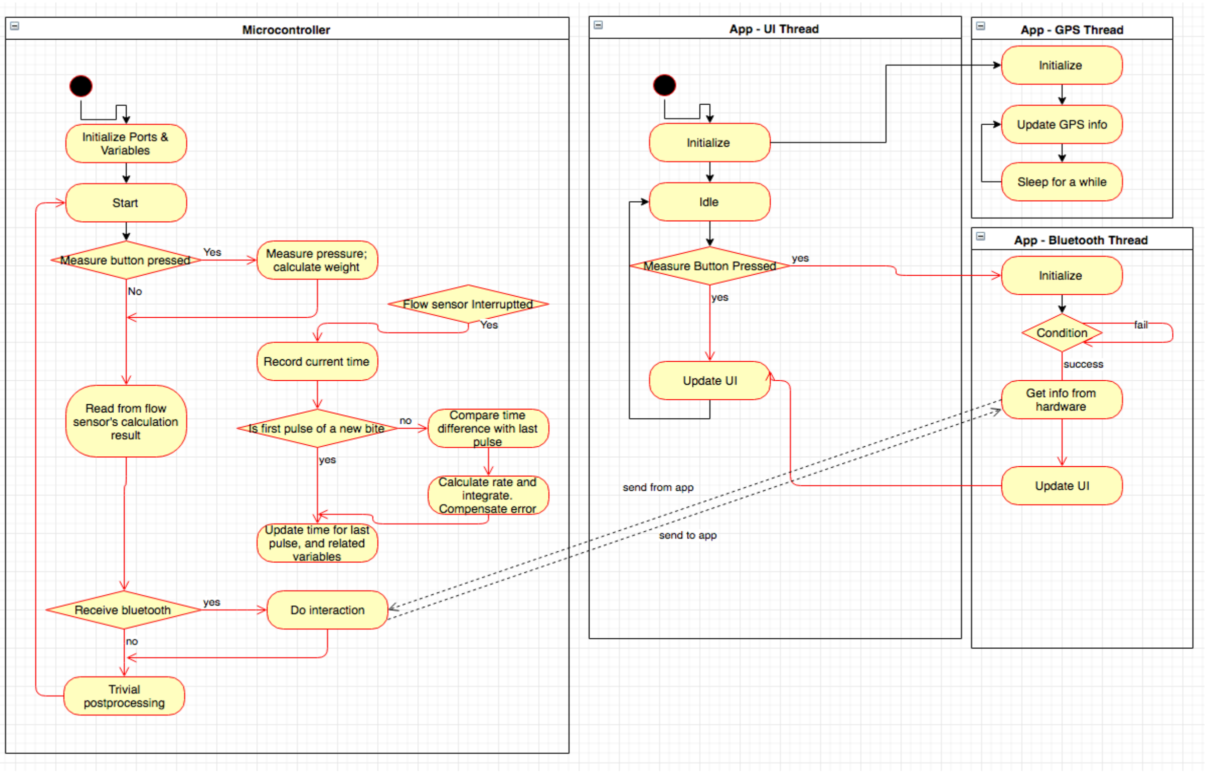


Figure 4 Overall Software Flowchart

3 Verifications

3.1 Microcontroller

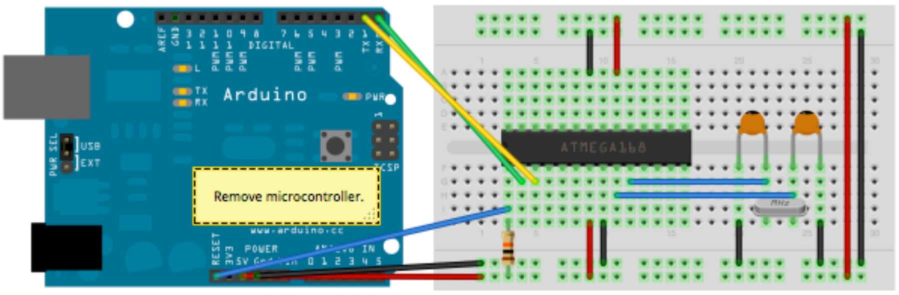
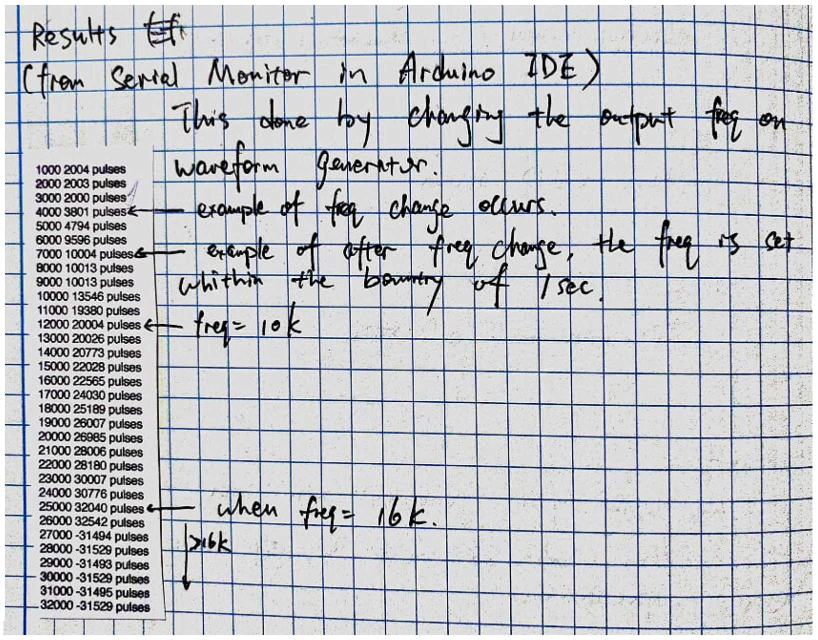


Figure 5 Connection to Microcontroller [3]

As mentioned above, we are using the Arduino bootloader, hence using the Arduino programs. The verification is done by connect the microcontroller to a programmer, which will transfer the programs onto it, and run a simple Arduino code. The connection is based on the Figure 5. The code to test this basically involves the blinking of a LED, and the LED works.

```
const byte ledPin = 13;
const byte interruptPin = 2;
volatile byte state = LOW;
int intcnt = 0;
void setup() {
  Serial.begin(9600);
  pinMode(ledPin, OUTPUT);
  pinMode(interruptPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE);
}
long lastsec = 0;
long time = 0;
void loop() {

  time = millis();
  if(time/1000 > lastsec){
    lastsec = time/1000;
    Serial.print(time);
    Serial.print(" ");
    Serial.print(intcnt);
    Serial.println(" pulses");
    intcnt = 0;
  }
}

void blink() {
  intcnt++;
}
```



Results [illegible]
(from Serial Monitor in Arduino IDE)
This done by changing the output freq on waveform generator.

```
1000 2004 pulses
2000 2003 pulses
3000 2000 pulses
4000 3801 pulses   ← example of freq change occurs.
5000 4794 pulses
6000 9596 pulses
7000 10004 pulses  ← example of after freq change, the freq is set whithin the boundry of 1sec.
8000 10013 pulses
9000 10013 pulses
10000 13546 pulses
11000 19380 pulses
12000 20004 pulses ← freq= 10k
13000 20026 pulses
14000 20773 pulses
15000 22028 pulses
16000 22565 pulses
17000 24030 pulses
18000 25189 pulses
19000 26007 pulses
20000 26985 pulses
21000 28006 pulses
22000 28180 pulses
23000 30007 pulses
24000 30776 pulses
25000 32040 pulses ← when freq= 16k.
26000 32542 pulses
27000 -31494 pulses   >16k
28000 -31529 pulses
29000 -31493 pulses
30000 -31529 pulses
31000 -31495 pulses
32000 -31529 pulses
```

Figure 6 Test Code for Interruption and Result

The other test performed is about the test for interruptions. The idea is simple, to maintain a counter, and it is incremented whenever an interrupt happens. For convenience it is set to change of the input logic voltage. Then we will check when do this counter value lost correspondence to the wave we are sending as the input. The test code and result shown in Figure 6. In short, the result shows that the interrupt is usable when the interrupt frequency is lower than 16k Hz.

3.2 LED Module

```
#include <LiquidCrystal.h>

// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(7, 8, 9, 10, 11, 12);

void setup() {
  // set up the LCD's number of columns and rows:
  lcd.begin(16, 2);
  // Print a message to the LCD.
  lcd.print("hello, world!");
}

void loop() {
  // set the cursor to column 0, line 1
  // (note: line 1 is the second row, since counting begins with 0):
  lcd.setCursor(0, 1);
  // print the number of seconds since reset:
  lcd.print(millis() / 1000);
}
```

Figure 7 Test Code for LED Module

As shown in Figure 7, the verification of the LED is fairly straight forward. The purpose is to see whether the LED would display the correct texts. It is divided into two parts, one is the static text in the setup(), and the other is the changing number of seconds in loop(). The result shows that the LED can display both of them correctly.

3.3 Bluetooth Module

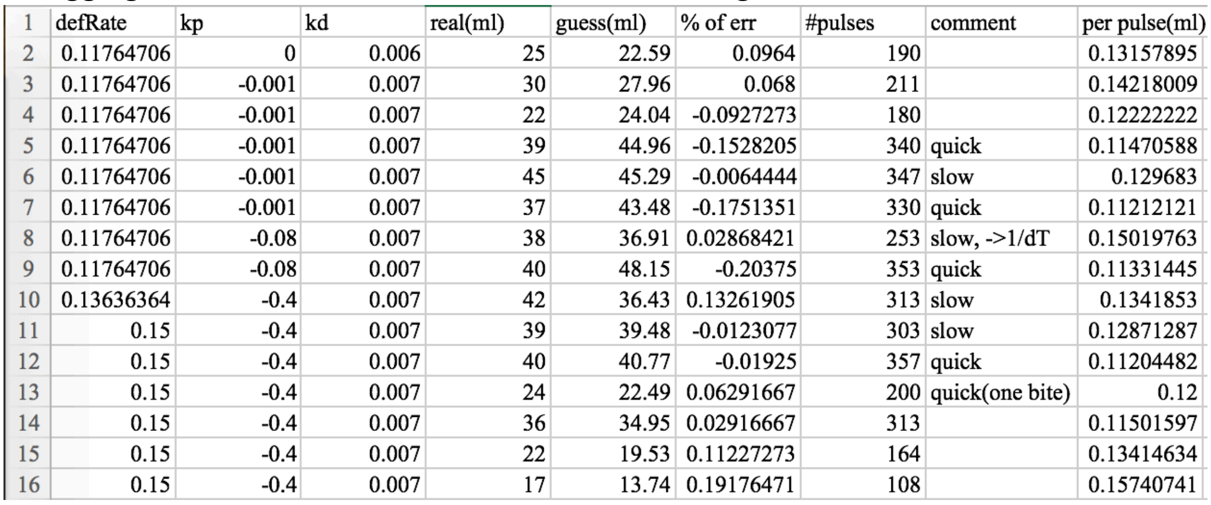
```
char data = 0;
void setup()
{
  Serial.begin(9600);
  pinMode(13, OUTPUT);
}
void loop()
{
  if(Serial.available() > 0)
  {
    data = Serial.read();
    Serial.print(data);
    Serial.print("\n");
  }

}
```

Figure 8 Test Code for Bluetooth

As shown in Figure 8, the verification of the Bluetooth is basically the serial communication. The test is built as follow: the program will be load onto the Arduino first, without the Bluetooth, then connect the Bluetooth module to the serial pins and reset the Arduino. Finally the phone is connected to the Arduino through Bluetooth and start sending and receiving the messages with this Bluetooth module. When sending the string “a” from phone, it will return 97 13 03, where 97 is the Ascii number for “a”, 13 for return and 03 for end of text, thus showing the correct result. Sending other strings will also result in the Ascii number for every character plus the return and end of text.

3.4 Flow Sensor

The verification of the flow sensor is using the core code as shown in Figure 2. Other part of the code is basically checking the variable accumWater constantly and print out its value, corresponding with the pulse number received and the time stamp as debugging info. The summarized result is shown in Figure 9.



1	defRate	kp	kd	real(ml)	guess(ml)	% of err	#pulses	comment	per pulse(ml)
2	0.11764706	0	0.006	25	22.59	0.0964	190		0.13157895
3	0.11764706	-0.001	0.007	30	27.96	0.068	211		0.14218009
4	0.11764706	-0.001	0.007	22	24.04	-0.0927273	180		0.12222222
5	0.11764706	-0.001	0.007	39	44.96	-0.1528205	340	quick	0.11470588
6	0.11764706	-0.001	0.007	45	45.29	-0.0064444	347	slow	0.129683
7	0.11764706	-0.001	0.007	37	43.48	-0.1751351	330	quick	0.11212121
8	0.11764706	-0.08	0.007	38	36.91	0.02868421	253	slow, ->1/dT	0.15019763
9	0.11764706	-0.08	0.007	40	48.15	-0.20375	353	quick	0.11331445
10	0.13636364	-0.4	0.007	42	36.43	0.13261905	313	slow	0.1341853
11	0.15	-0.4	0.007	39	39.48	-0.0123077	303	slow	0.12871287
12	0.15	-0.4	0.007	40	40.77	-0.01925	357	quick	0.11204482
13	0.15	-0.4	0.007	24	22.49	0.06291667	200	quick(one bite)	0.12
14	0.15	-0.4	0.007	36	34.95	0.02916667	313		0.11501597
15	0.15	-0.4	0.007	22	19.53	0.11227273	164		0.13414634
16	0.15	-0.4	0.007	17	13.74	0.19176471	108		0.15740741

Figure 9 Test Result for Flow Sensor

In Figure 9, as the before the line 8 shows, the equation used in the flow sensor is showing error corresponding to the speed of drinking the water, and it seems to have the reversed relationship. Therefore, starting from line 8, the kp part is changed to be reversed with dT. After some parameter adjustments, it achieves a relatively good result in line 11~14.

The last two lines are corresponding to a much smaller bite, which is not compensated correctly right now. We are planning on collecting more data and pulse patterns with the small bites, to observe the differences between the normal bites, and adjust the current parameters in the program and test again. This should happen in the week of Oct. 31.

4 Conclusion

In summary, most of the programming related to the microcontroller is finished. But there still remains the problem with the flow sensor, and the specific hardware for the pressure sensor is still in testing and have the potential to change. Thus I am a little fall behind the schedule. However, the remaining work for different parts in the microcontroller should be simpler since most of the programming schemes are figured out and the formulas are already built. In terms of workload, I think I am about the average workload compares to the group. Although there is less testing involved in the software, figure out what is the right way to do and how to do takes most of the time. Also, when the error occurs in the interaction between software and hardware, it needs more adjustment and those are relatively not straight forward compares to pure software, for example the parameters in the flow sensor.

The remaining work mainly lies on two parts, one is the phone App, and the other is the flow sensor. Those two will be the most time consuming parts, since the App needs more complicated programs and the flow sensor's parameter adjustment requires great amount of experiments. The pressure sensor, regardless of the hardware we will use, the input should be easy to convert into the digital results and only requires minimum level of error corrections. The assemble of all the code related to the microcontroller will be straight forward as well, since each unit is already working.

Therefore, in the following weeks I think it is more reasonable for me to start working on the two parts at the same time, both developing the App and continuing on the microcontroller. Since I still need some help with the hardware side, also need the PCB to do the integration test for programs on microcontroller, it would be wise to program the App when I cannot move forward on the microcontroller. As a result, I am planning to concentrate on basic functionalities in the App more first and after that try to balance the work in microcontroller and the App. It summarizes as follow

Week of	Main Task
Oct 31	Basic functions in App & flow sensor
Nov 7	Integration test for microcontroller & pressure sensor
Nov 14	Remaining parts in App

5 Bibliography

[1] G. K.-H. A. R. Jonathan Corbet, "10.4. Top and Bottom Halves," O'Reilly, [Online]. Available: http://www.makelinux.net/ldd3/chp-10-sect-4.

[2] E. Chatzikyriakidis, "QueueList Library For Arduino," [Online]. Available: http://playground.arduino.cc/Code/QueueList.

[3] "From Arduino to a Microcontroller on a Breadboard," [Online]. Available: https://www.arduino.cc/en/Tutorial/ArduinoToBreadboard.

[4] "PID_controller," [Online]. Available: https://en.wikipedia.org/wiki/PID_controller.