6.115 Final Project
Pranav Murugan

Introduction

This project provides several features designed to aid in monitoring, regulation, and quality-of-life for hospital patients that require the use of a ventilator. Ventilators are expensive medical devices that use pumps to help patients breathe if they are not able to using their own power. The price and scarcity of ventilators were recently highlighted with the COVID-19 pandemic, and this project explores a possible way to provide low cost breathing tracking infrastructure and support with additional features for ease of communication while intubated.

Broadly, this project is divided into two sections. The breath-tracking portion of the project uses a MPRLS pressure sensor to measure the output pressure from the lungs. In combination with an aperture of known diameter, this enables the system to calculate volume flow rate and display the measurement as a graph visually for the user. Another mode uses the pressure sensing to modulate the pressure output of a pump via feedback control from the pressure sensor, which can provide respiratory support during inhalation.

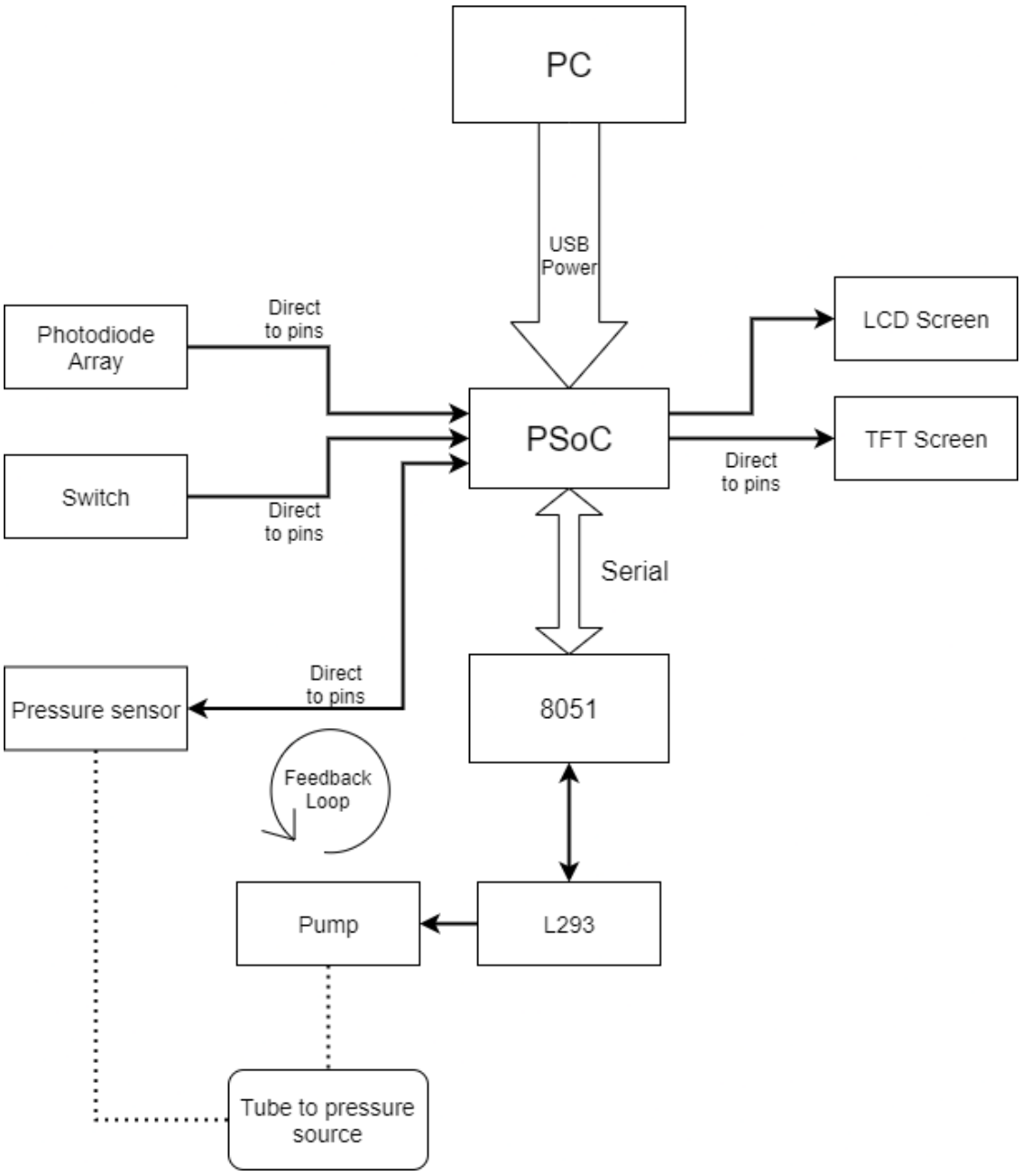
The communication portion of the system involves using the breath sensor and a photodiode array as a simple gesture-based control of a text-writing system, with which the user can write words and communicate with the aid of an autocomplete system. This contactless method of written communication is aimed at improving the quality of life for patients that are unable to speak or lose fine motor controls during their stay in the hospital.

The following sections cover the overall hardware and software schematics, as well as the intended use and expected output of the system. Two appendices containing commented code is provided at the end of the report.

Hardware Summary

The components of this project are controlled and powered by both the PSoC 5LP Big Board and the R31JP microprocessor and kit. Thus, there are a combination of independent ICs used as well as the programmable components of the PSoC 5. This section of the report focuses on the high-level structure of the project, following the block diagram on the next page; a more in-depth component-by-component description is in the next Hardware Schematics section.

The cornerstone of this project is the pressure sensor, wired directly to the PSoC as shown in the block diagram. It is connected to a 3/32” ID vinyl fuel line tube to facilitate pressure application to the sensor. For safety reasons, this tube is connected to a balloon instead of to a users mouth as would be the ideal use case. The tube is also slightly modified such that there are two lengths running in parallel, taped together, but only one is connected to the pressure sensor. This open tube is to facilitate airflow through a hole of



Hardware block diagram for the final project

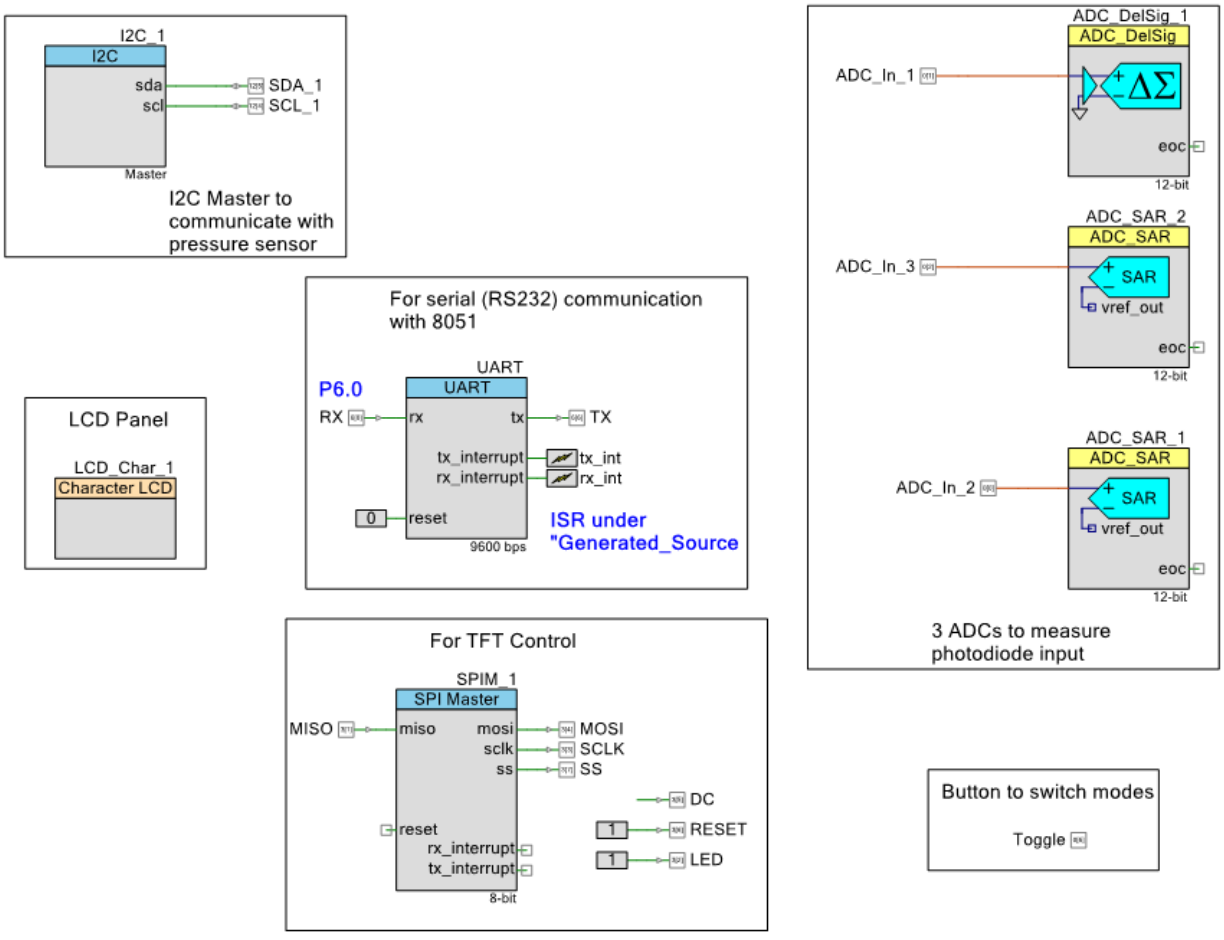
known diameter, which will aid in flow rate calculations as we will discuss in the software section. This tube also connects to the pump output. To drive the pump, the 8051 is connected to the L293. This ensures that adequate wattage can be delivered to the pump, which may be beyond the power the 8051 can sustain at one of its output pins. A button switch to cycle between system modes is wired directly to the PSoC. The photodiode array, in this case just a set of independent IR photodiodes, is wired directly to the PSoC as well. The exact circuit is covered in the next section but this enables the use to provide input to the system without making physical contact with any of the components.

The graphical output of the system is handled by a TFT screen and an LCD screen that are wired directly to the PSoC. The LCD provides more of the raw data output by the pressure sensor (i.e. volume flow rate updated every tenth of a second or the control byte sent to the 8051 to determine the duty cycle at which it runs the pump). The TFT holds more user-friendly information and presents the graphical data collected, such as the plot of volume flow rate over time.

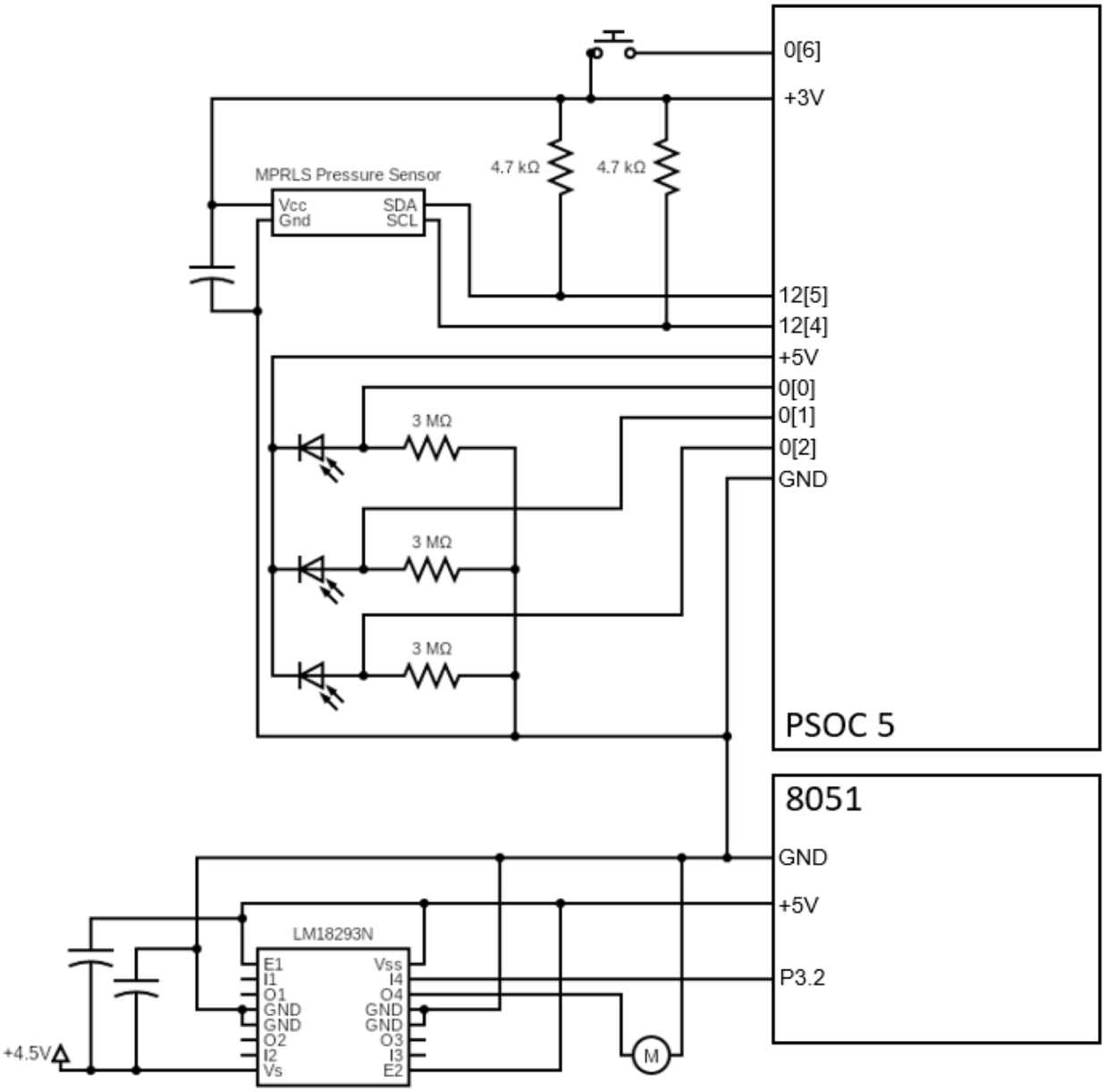
Finally, the communication between the PSoC and 8051 is handled by a serial connection over the DB9 connector. This is sufficient for our purposes since polling the pressure sensor only happens at most ~50 times a second, and updates to the pump voltage are even less frequent. This is well within the bitrate of this communication method. Power to the PSoC is delivered via USB from a laptop, and the 8051/R31JP is powered by the kit power supply.

Hardware Schematics

The detailed hardware schematics are divided into two diagrams. First is the PSoC internal components from the PSoC Creator schematic. Second is the wiring schematic diagram for all the components external of the PSoC.



PSoC Internal Hardware Schematic



Project wiring schematics

We can now proceed component-by-component. Images of the assembled project and its parts are provided later in the report.

- **I2C communications channel**. The MPRLS pressure sensor from Honeywell is a 3-5V compatible pressure sensor that functions as an I2C slave. Because it is wired directly to the PSoC, we power it with 3V. By I2C standards, SDA and SCL high is the default no-communication state. Thus, we have to use pull-up resistors to ensure the value stabilizes at 1 given no input (see wiring schematics). Although the PSoC internal pull-up resistors have resistance ~5kOhms, the I2C protocol is very sensitive to variations in capacitance and signal timing as a result. Explicitly pulling the lines high gave much more consistent performance. Also to ensure more consistent

performance, the I2C master IC was set to fixed function instead of UDB so that the timing of the read and write signals are necessarily less precise. Sharing a power line with the TFT display also occasionally caused resents even when bypassed, so it is running on its own 3V line. The SDA and SCL lines are connected to 12[5] and 12[4], respectively. There are other pins on the pressure sensor IC left unconnected. There is a 3V output pin, since the MPLRS can downconvert 5V power for its internal use and to supply other low-wattage ICs. There is also an active-low reset pin, which is drawn up when not connected. Finally, the EOC pin is left unconnected since the sensor reports a status byte upon data request which contains a bit indicating if conversion is under process since the last write command. Since the SDA and SCL pins are manually pulled high, the PSoC pins are simply open drain, drives low.

- **LCD Panel.** The LCD panel is the one connected to the PSoC since near the beginning of this class. The hardware interface is plug-and-play as long as it is limited to use in only the 3.3V slot.
- **Serial communications.** The universal asynchronous receiver/transmitter integrated IC enables the PSoC serial communications over RS-232 DB9 connector provided. It is running at default settings for the 8051 (9600 baud) with 8 data bits. Interrupts are enabled, but this project configuration does not require receiving data from the 8051, so the hardware connections are not used. To ensure both systems were properly grounded, the VSSD ground on the PSoC was tied to the kit/8051 ground before connection via the serial port. In addition, the RX and TX pins were connected to 6[0] and 6[6] externally for psoc access to the pins.
- **TFT Module.** The TFT communicates with the PSoC using SPI, a different serial interface. The SPI Master IC controls the master/slave communications as well as the clock input. These values were left at default from the provided setup. The pins take up P3[1:7] as can be seen in the PSoC schematic (and were wired as indicated), in addition to two wires for 3.3V power and ground.
- **ADC ICs.** The onboard ADC ICs are used to process the voltage generated by the photodiode circuit. Since the PSoC only has one Delta – Sigma ADC, two of the inputs are processed by SAR ADCs. Although SAR ADCs sacrifice accuracy for sampling rate, the purposes of sensing a brightness threshold require neither high accuracy nor very fast sampling rates. For our purposes, both ADCs are perfectly adequate. They measure the voltage relative to ground and give a 12-bit digital output. We do not measure the EOC pins, and simply check the ADC control register value to confirm when conversion is complete.
- **Photodiode Inputs.** The photodiodes serve as light sensors that enable the PSoC, after measurement with the ADC, to sense proximity and trigger based on shadows of 940nm wavelength IR light. There is an IR motion sensor already included in our kit, but it is far too sensitive to use since it is sensitive to distance at the order of meters and not inches as we require. The photodiodes do not have a transistor to amplify the current, so they only produce < 1 microAmps of reverse current under the conditions they will be used in this project. Thus, to get a measurable voltage sample, they are connected in series (but reverse bias) to three 1 megaOhm resistors (also in series). This puts the maximum voltage drop across the resistors due to the reverse current at around 1V, sufficiently high resolution for the ADCs to sense when they are covered by a hand and when they are not. The input pins are analog.
- **Button switch.** The button switch is connected to the PSoC to indicate mode changing; pressing (and releasing) the button will cycle through the modes of the system. One pin of the button is tied to voltage high, and upon the press of the button the other pin reads high as well which is detected by the 0[6] pin.

- **Grounding.** As noted previously both the 8051 and PSoC share a common ground so that a serial connection will not overvolt any of the components.
- **Pump control.** The 8051 is connected primarily to the LM18293N to help drive the pump. We do not need an analog variable voltage circuit since we can control the effective DC voltage applied to the pump via pulse width modulation. The LM18293N is tied to Vs = 4.5V, the rated voltage of the pump, and logic voltage is tied to the 8051's Vss = 5V. In the circuitry all inputs are enabled but the project only requires one output (output 4). This output is connected to the pump (the pump operates in the same direction regardless of polarity), and the input is driven by the 8051 P3.2. An alternative would have been to use an PWM block on the PSoC or to connect an analog output to an op-amp, but given the serial connection already existed and to minimize the risks of improper grounding the 8051-only method is sufficient. The details of the implemented PWM scheme will be discussed in the software section.

Software Schematics

The high-level software functionality can be divided by mode. Mode 1 measures flow rate and provides graphical feedback, Mode 2 uses the measured flow rate to adjust pump voltage for feedback control, and Mode 3 uses the pressure sensor apparatus as an I/O method with which the user can interact with the system.

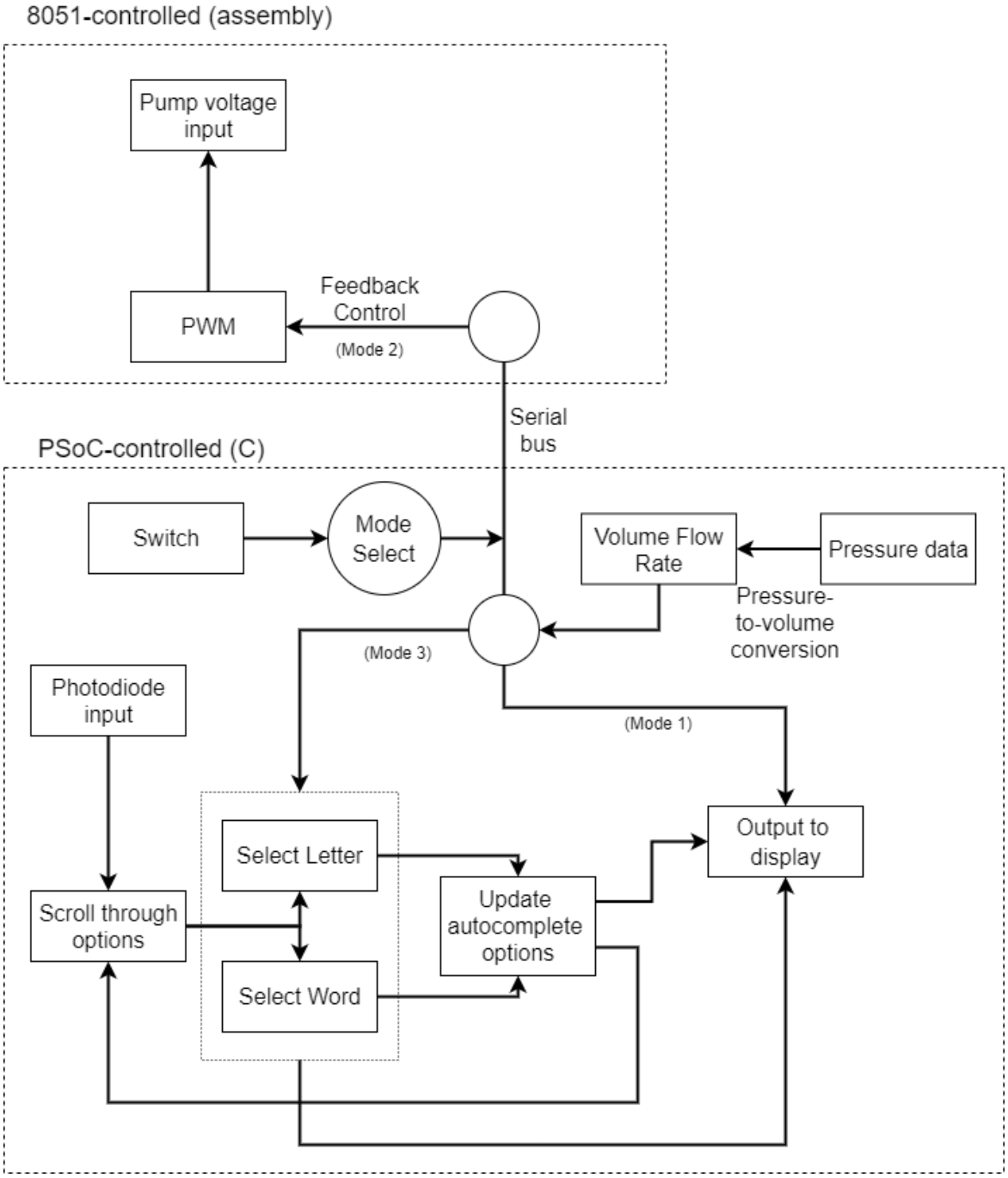
Before going into the software specifics for each mode, the pressure data to volume flow rate pathway is critical for the function of all parts of this project, as is highlighted on the right in the schematic on the next page. This process follows the I2C standards for master – to – slave communication, and the exact code can be found in Appendix A, in functions get_pressure() and get_flow_rate().

The function get_pressure() contains the entirety of the communication with the MPRLS pressure sensor. After initializing the I2C IC, the program initiates and halts a series of writes to a set of addresses counting up from 0. The address of this sensor is at 0x18, which is further confirmed by the documentation, but removing this section of code causes future start attempts to fail. It may be a consequence of buffer sizes/caching on the side of the PSoC, since writing or reading a full buffer fails every time but initiating a read/write and manually reading and acknowledging every byte works fine. This is the approach in the provided code, where after the initialization process the conversion start signal (writing 0xAA, 0x00, 0x00 in sequence) is given. The MPRLS sensor has a 32 bit output for every pressure conversion. The first byte is a status byte, which indicates (among other things) whether the conversion is complete, followed by three bytes to encode the pressure (linearly from 0 to 25 psi). If any of these read or write requests results in an error, the system is set to reset itself (since there are infinite loops that will not break otherwise). After obtaining this 24-bit pressure hex value, we rescale it to obtain the pressure in PSI.

Since the variable we are most interested in is the volume flow rate, we use the function get_flow_rate() to convert from the pressure differential to the flow rate since we know the hole size in the tube. The general formula is

$$\Phi_V = \frac{\Delta P d^2}{P_{\text{avg}}} \sqrt{\frac{\pi RT}{32M}}$$

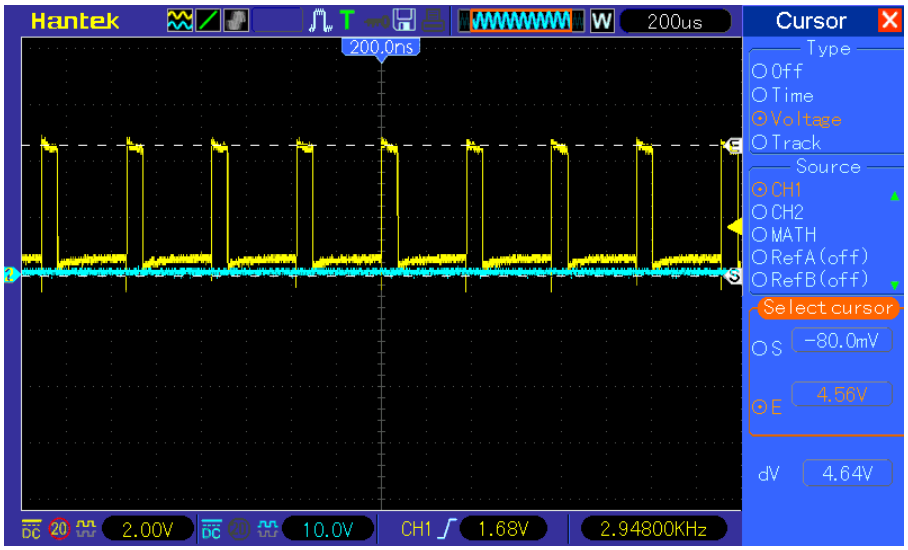
for which we know the diameter *d* of the tube and the remaining are known constants; we can then rescale by the constants in get_flow_rate() to get volume flow rate in liters/min. This quantity and the pressure in psi above are the key software inputs to the other modes.



Software schematics

- **Mode 1:** This is the first part of the for loop. The PSoC measures the pressure every 50 ms, and when it detects a breath (that is, a large enough pressure differential), it saves the pressure data for the next approximately 20 seconds, and then calls graph() to plot it on the PSoC.

- **Mode 2:** This feedback loop is similar to the one constructed in Lab 4. We simply take the distance of the pressure from an arbitrary target and rescale it. Subtracting from 0xFF is to get the input in a proper form for the 8051
    - **8051 PWM:** The 8051 code comes into play here, and can be found in Appendix B. The program has interrupts enabled and is set up to receive from the serial port at 9600 baud. The program produces a square wave on P3.2 that spends a fixed 42 machine cycles high, and a variable number of machine cycles (set by the register R1). Thus, the duty cycle can be varied by the PSoC since every reception will trigger an interrupt that will update R1 given the pressure reading. We note that 01h is a duty cycle close to 1 and 0FFh is a duty cycle close to 0. A representative image of the square wave produced is shown below, for R1 = 0x74h

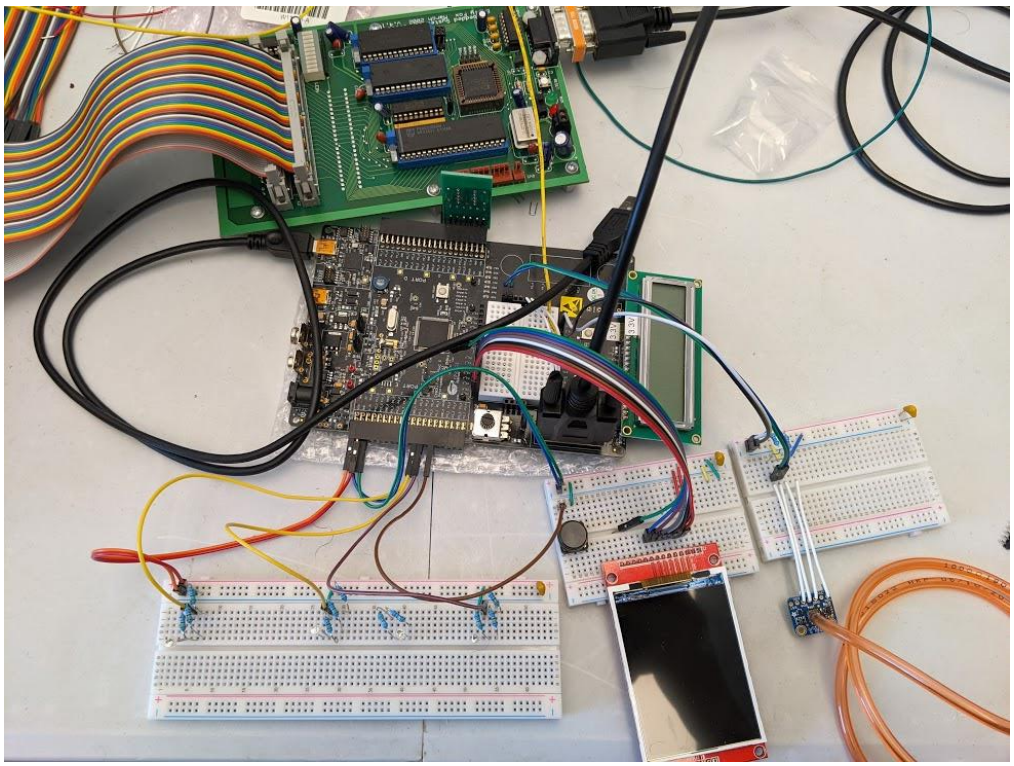


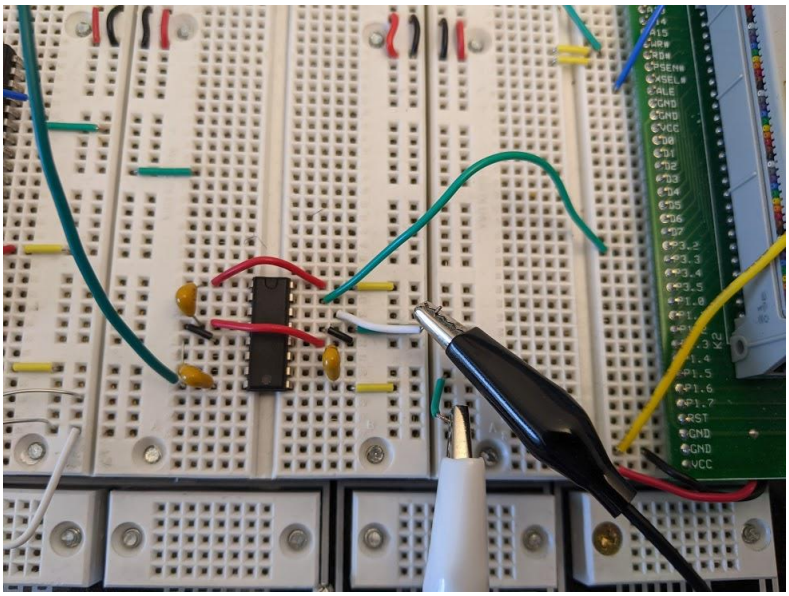
- **Mode 3:** This mode uses the pressure readings to interact with the text writing interface. Each of the three ADCs (each connected to a photodiode) is queried in sequence to determine the subsequent action. One photodiode selects the letter, one switches focus between the letter-by-letter word and the autofill suggestion, and the last submits a word and prepares for the next round of inputs. Each word is capped at 16 characters, and the last word entered is displayed on the screen for reference.

Usage and Expected Output

The assembled product looks like the following images (one of the PSoC + connections, and another of the wiring of the pump/LM18293N). To start, the program must be loaded onto the PSoC (and powered via USB from the laptop) and the pump assembly code has to be transferred to the 8051. After transfer to the 8051 it must be put in run mode and connected to the PSoC DB9 cable.

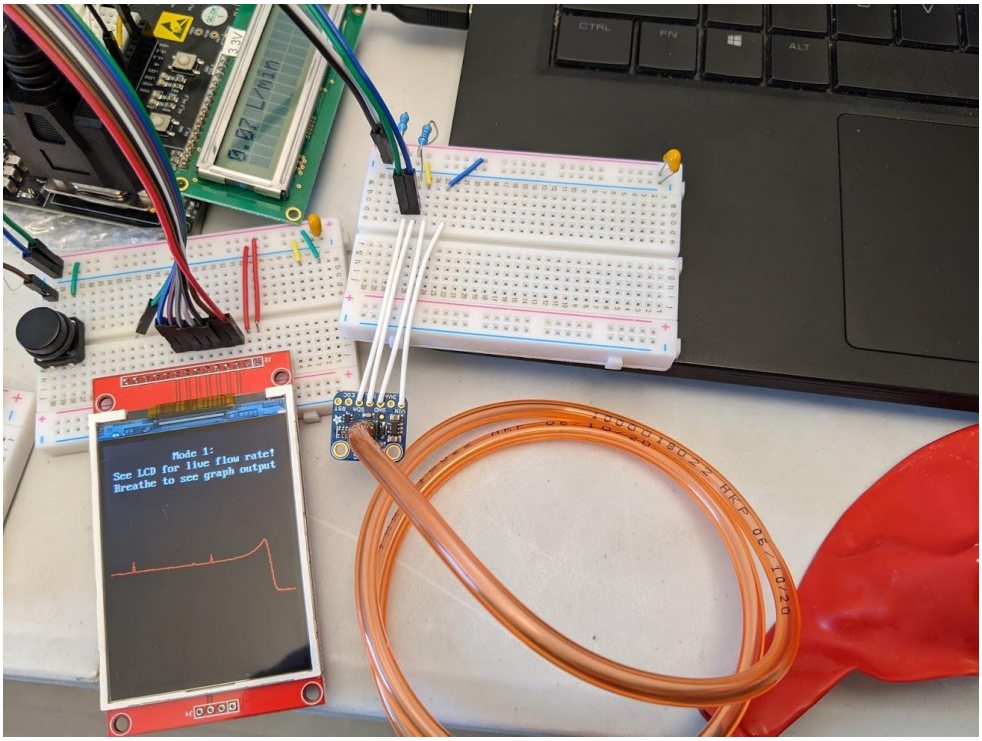
Now, we will step through how to use each mode of the program and the expected results from each. The program starts in Mode 1, the breath-tracking program.





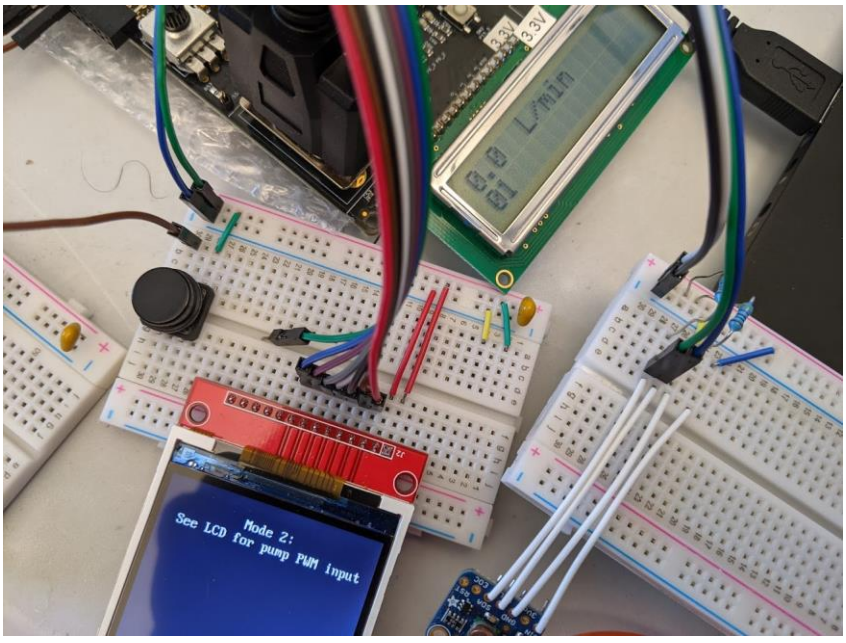
Mode 1:

The only user interaction in Mode 1 is applying air pressure to the tube. The TFT display prompts the user to blow on the tube; live instantaneous flow rates are displayed on the LCD and a graph of the flow rate over time is displayed on the TFT. A representative example output is shown on the following page. We note, for safety, to use a balloon as the model of the human lungs.



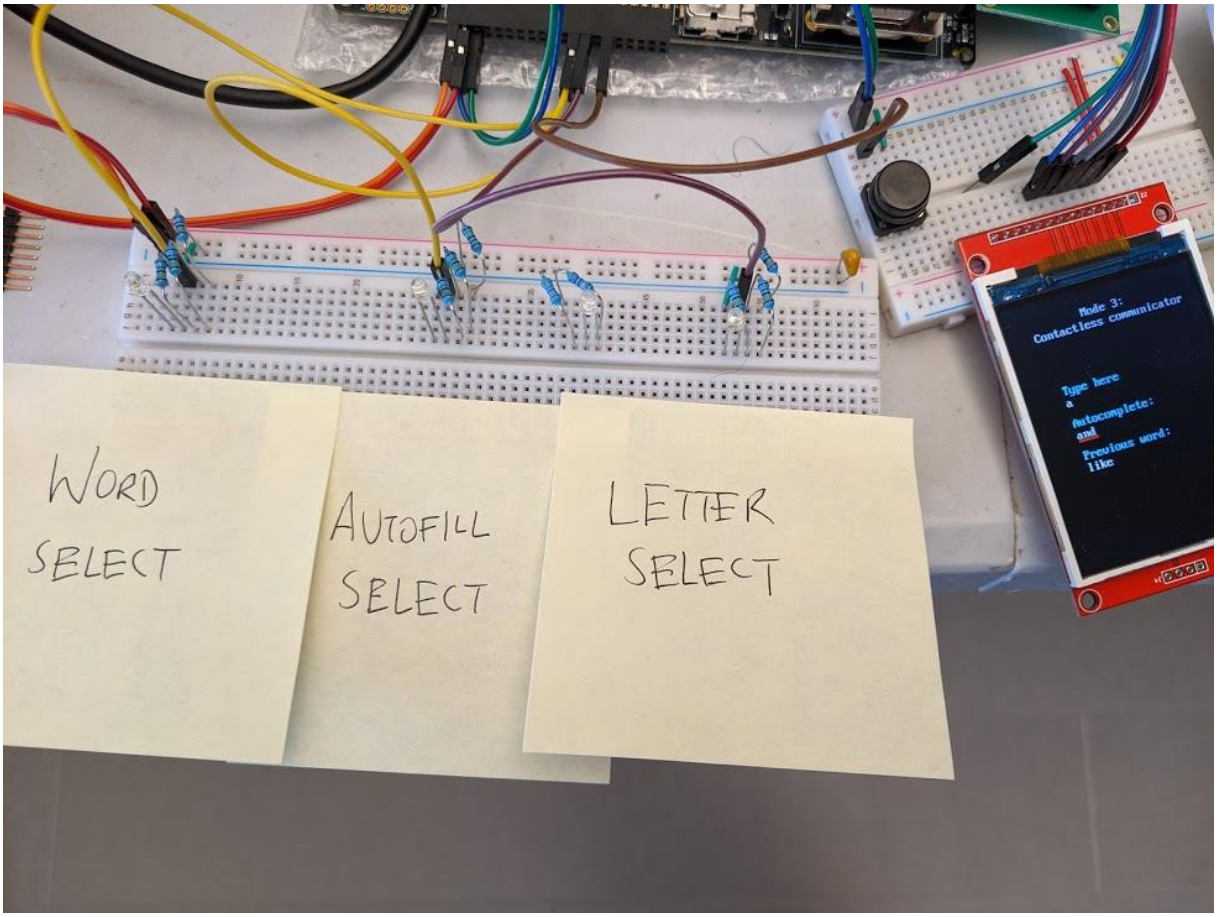
Mode 2:

Again, the only user interaction is applying pressure to the input tube. The reference pressure parameter is key here; in the following image, it is set to ~0.5 psi above ambient pressure. When in mode 2 with no additional pressure input, the pump will be at maximum voltage (corresponding to the hex 0x01 visible in the LCD screen in the image below). As pressure is added, the voltage across the pump will drop in accordance to the feedback loop set up. The pump used in this project is not powerful enough to actually maintain an inflated balloon, but its response to the changing applied voltage can be measured and heard.



Mode 3:

The final mode includes a new way of interacting with the program. At start, there is a letter highlighted and a suggested word based on that letter. Blowing on the pressure sensor causes the single highlighted letter to cycle. By blocking the light to the photodiodes, the user also can activate one of the three commands; word select, to approve the highlighted word (whether typed or suggested) and to reset the typing mechanism, autofill select which changes which word is highlighted, and letter select, which allows the user to build the typed word one letter at a time. A representative output after some time is shown in the image below.



Appendix A
PSoC main.c

```
#include <project.h>
#include <device.h>
#include "GUI.h"
#include "tft.h"

uint32_t pressure = 0x20;
double psi, fractional, integer, flowrate;
double targ_pressure = 15.0;
uint8 parr[4];
uint8 * status;
uint8 wbuf[] = {0xAA,0x00,0x00};
uint8 addr = 0x80;
uint8 addr7;
uint8 err;
I16 aY[236];
int record = 236;
int graph_flag = 0;
int mode = 0;
uint16 adcResult = 0;
char word[16];
int letter_count;
char last_word[16];
int autofill;
char common_words[26][16] = {
    "and",
    "but",
    "come",
    "do",
    "even",
    "for",
    "get",
    "her",
    "it",
    "just",
    "know",
    "like",
    "me",
    "no",
    "other",
    "people",
    "question",
    "reply",
    "some",
    "then",
    "use",
    "very",
    "what",
    "xray",
```

```
    "yes",
    "zebra"
};

double get_pressure(void);
void print_float(double p, int digits, int clear, int x, int y);
double get_flow_rate(double p);
void MainTask0(void);
void MainTask1(void);
void MainTask2(void);
void graph(void);
void draw_line_high(void);
void draw_line_low(void);

int main()
{
    CyGlobalIntEnable; /* Enable global interrupts. */
    LCD_Char_1_Start();                          // initialize lcd
     LCD_Char_1_ClearDisplay();

    SPIM_1_Start(); // initialize TFT display
    GUI_Init();                                  // initilize graphics
library

    ADC_DelSig_1_Start();                       // start the ADC_DelSig_1
     ADC_DelSig_1_StartConvert();               // start the ADC_DelSig_1
conversion
    ADC_SAR_1_Start();
    ADC_SAR_2_Start();

    addr7 = addr >> 1; // 7 bit address
    addr = 0;

    for(;;)
    {
        if (mode == 0){
            MainTask0();
            LCD_Char_1_Position(0,5);
            LCD_Char_1_PrintString("L/min"); // prep output
            while (mode == 0){
                psi = get_pressure();
                flowrate = get_flow_rate(psi); // get flowrate
                if (record >= 236 && flowrate > 0.1){ // if breathing
initiate recording
                    record = 0;
                    graph_flag = 1;
                }
                if (record < 236){ // record inverse because graph
counts down
                    aY[record] = 100-(I16) 100*flowrate;
                    record ++;
```

```
                }
                if (graph_flag == 1 && record == 236){ // graph 236
data points
                    graph_flag = 0;
                    graph();
                }
                print_float(flowrate, 2, 0, 0, 0);
                CyDelay(50);
                if (Toggle_Read() == 1){ // check if button pressed
                    while(Toggle_Read() == 1){}
                    mode = 1;
                }
            }
        }
        if (mode == 1){
            UART_Start();                       // initialize UART
            MainTask1();
            while(mode == 1){
                psi = get_pressure();
                print_float(flowrate, 2, 0, 0, 0);
                double err = targ_pressure-psi; // get pressure
differential
                err = err*0xFF/0.5; // rescale
                int data;
                if(err < 0){
                    data = 0x00;
                }
                else if (err >= 0xFF){
                    data = 0x01;
                }
                else{
                    data = (int) (0xFF-err); // convert to R1 hex
value
                }
                LCD_Char_1_Position(1,0);
                LCD_Char_1_PrintInt8(data);
                LCD_Char_1_Position(0,0);
                UART_WriteTxData(data); // send to 8051
                CyDelay(50);
                if (Toggle_Read() == 1){ // check if button pressed
                        while(Toggle_Read() == 1){}
                        mode = 2;
                        UART_Stop();                        // stop
UART
                        LCD_Char_1_Position(0,0);
                        LCD_Char_1_PrintString("           ");
                        LCD_Char_1_Position(1,0);
                        LCD_Char_1_PrintString("    ");
                    }
            }
        }
```

```
        if (mode == 2){
            letter_count = 0;
            last_word[0] = 'N';
            last_word[1] = '/';
            last_word[2] = 'A';
            last_word[15] = '\0';
            word[0] = 'a';
            word[15] = '\0';
            autofill = 0;
            MainTask2();
            while(mode == 2){
                CyDelay(500);

                // Cycle through letters dependent on pressure
                psi = get_pressure();
                if (psi > 14.6){
                    if (word[letter_count] < 'z'){
                        word[letter_count]++;
                    }
                    else{
                        word[letter_count] = 'a';
                    }
                }

                //update TFT
                GUI_SetColor(0xFFFFFF);
                GUI_DispStringAt(word,40,120);
                GUI_DispCEOL();
                if (letter_count ==0) {
                    GUI_DispStringAt(common_words[word[0]-97],40,160);
                    GUI_DispCEOL();
                }
                if (autofill == 0){
                    draw_line_high();
                }
                else{
                    draw_line_low();
                }
                GUI_SetColor(0xFFFFFF);

                // check first ADC
                ADC_SAR_2_StartConvert();
                while(
!ADC_SAR_2_IsEndConversion(ADC_SAR_2_WAIT_FOR_RESULT) )
                {}
                adcResult = ADC_SAR_2_GetResult16();        // read
the adc and assign the value adcResult
                      if (adcResult & 0x8000)
                      {
                      adcResult = 0;       // ignore negative ADC
results
```

```
                    }
                    else if (adcResult >= 0xfff)
                    {
                    adcResult = 0xfff;      // ignore high ADC
results
                    }
                // If activated, select letter
                if(adcResult < 200){
                    letter_count++;
                    word[letter_count] = 'a';
                    continue;
                }

                // check second ADC
                ADC_SAR_1_StartConvert();
                while(!
ADC_SAR_1_IsEndConversion(ADC_SAR_1_WAIT_FOR_RESULT) )
                {}
               adcResult = ADC_SAR_1_GetResult16();         // read
the adc and assign the value adcResult
                    if (adcResult & 0x8000)
                    {
                    adcResult = 0;       // ignore negative ADC
results
                    }
                    else if (adcResult >= 0xfff)
                    {
                    adcResult = 0xfff;      // ignore high ADC
results
                    }

                // If activated, toggle between autcomplete and
letter-by-letter
                if(adcResult < 200){
                    if (autofill == 0){
                        autofill = 1;
                        draw_line_low();
                    }
                    else{
                        autofill = 0;
                        draw_line_high();
                    }
                    continue;
                }

                // check third ADC
                while(
!ADC_DelSig_1_IsEndConversion(ADC_DelSig_1_WAIT_FOR_RESULT) )
                {}
                adcResult = ADC_DelSig_1_GetResult16();          //
read the adc and assign the value adcResult
```

```
                    if (adcResult & 0x8000)
                    {
                    adcResult = 0;       // ignore negative ADC
results
                    }
                    else if (adcResult >= 0xfff)
                    {
                    adcResult = 0xfff;      // ignore high ADC
results
                    }

                // If activated, select either the typed or auto word
and update
                if(adcResult < 200){
                    if(autofill == 0){
                        int i;
                        for(i = 0; i < 16; i++){
                            last_word[i] = word[i];
                            word[i] = 0;
                        }
                        word[0] = 'a';
                        word[15] = '\0';
                    }
                    if(autofill == 1){
                        int i;
                        for(i = 0; i < 16; i++){
                            last_word[i] = common_words[word[0]-
97][i];
                        }
                        for(i = 0; i < 16; i++){
                            word[i] = 0;
                        }
                        word[0] = 'a';
                        word[15] = '\0';
                    }
                    autofill = 0;
                    letter_count = 0;
                    GUI_DispStringAt(last_word,40,200);
                    GUI_DispCEOL();
                    draw_line_high();
                    continue;
                }


                if (Toggle_Read() == 1){ // check if button pressed
                        while(Toggle_Read() == 1){}
                        mode = 0;
                    }
            }
        }
    }
```

```
}

double get_pressure(void){
    I2C_1_Start(); // start I2C communication
    pressure = 0x20;
    addr = 0x00;   // reset address
    while(addr < 0x19){ // i need this but i dont know why
        I2C_1_MasterSendStart(addr, I2C_1_WRITE_XFER_MODE);
        I2C_1_MasterSendStop();
        addr ++;
    }

    //LCD_Char_1_PrintString("D"); For debugging

    //--
    // Send write signal to pressure sensor, then initiate write
    err = I2C_1_MasterSendStart((uint8) 0x18, I2C_1_WRITE_XFER_MODE);
    if (err != 0x00){
        CySoftwareReset();
    }
    I2C_1_MasterWriteByte((uint8) 0xAA);
    I2C_1_MasterWriteByte((uint8) 0x00);
    I2C_1_MasterWriteByte((uint8) 0x00);
    I2C_1_MasterSendStop();
    //LCD_Char_1_PrintString("D"); For debugging
    I2C_1_MasterClearStatus();
    //--

    // wait until bus is free
    while(!I2C_1_CHECK_BUS_FREE(I2C_1_MCSR_REG)){}

    // check status byte until conversion is over, reset if error
    while (pressure & 0x20){
        err = I2C_1_MasterSendStart(0x18, I2C_1_READ_XFER_MODE);
            if (err == 0x60){
                CySoftwareReset();
            }
        pressure = I2C_1_MasterReadByte(I2C_1_NAK_DATA);
        I2C_1_MasterSendStop();
    }

    //--
    //Once status is ready, read all 4 bytes (1st is status, last 3
data)
    I2C_1_MasterSendStart(0x18, I2C_1_READ_XFER_MODE);
    pressure = I2C_1_MasterReadByte(I2C_1_ACK_DATA);
    pressure <<= 8;
    pressure |= I2C_1_MasterReadByte(I2C_1_ACK_DATA);
    pressure <<= 8;
    pressure |= I2C_1_MasterReadByte(I2C_1_ACK_DATA);
    pressure <<= 8;
```

```
    pressure |= I2C_1_MasterReadByte(I2C_1_NAK_DATA);
    I2C_1_MasterSendStop();
    I2C_1_MasterClearStatus(); //Clear I2C master status

    //LCD_Char_1_Position(1,0);
    //LCD_Char_1_PrintInt32(pressure);

    // convert pressure to psi
    psi = (pressure & (0x00FFFFFF));
    psi = psi * (float) 25;
    psi = psi/0x00FFFFFF;

    I2C_1_Stop();
    return psi;
}

double get_flow_rate(double p){
    // returns volume flow rate in L/min
    double dp = p - 14.55;
    dp = dp*6894.76;
    dp = dp*60;
    dp = dp*0.000005206;
    return dp;
}

void print_float(double p, int digits, int clear, int x, int y){
    if(clear == 1){
        LCD_Char_1_ClearDisplay();}

    // separate integer and fractional parts, and print to LCD
    fractional = p - (int64) p;
    integer = p - fractional;
    int i;
    for (i = 0; i < digits; i++){
        fractional = 10*fractional;
    }
    if ((int) fractional == 0){
        LCD_Char_1_Position(x,y+2);
        LCD_Char_1_PrintString("  ");}
    LCD_Char_1_Position(x,y);
    LCD_Char_1_PrintNumber((uint16) integer);
    LCD_Char_1_PrintString(".");
    LCD_Char_1_PrintNumber((uint16) fractional);
}

void MainTask0()
{
    // Init mode 1
    GUI_Clear();
    GUI_SetFont(&GUI_Font8x16);
```

```
    GUI_DispStringHCenterAt("Mode 1:\nSee LCD for live flow
rate!\nBreathe to see graph output", 120,20);
}

void MainTask1()
{
    // Init mode 2
    GUI_Clear();
    GUI_SetFont(&GUI_Font8x16);
    GUI_DispStringHCenterAt("Mode 2:\n See LCD for pump PWM input",
120,20);
}

void MainTask2()
{
    // Init mode 3
    GUI_Clear();
    GUI_SetFont(&GUI_Font8x16);
    GUI_DispStringHCenterAt("Mode 3:\n Contactless communicator",
120,20);
    GUI_SetColor(0x00FFFF);
    GUI_DispStringAt("Type here", 40, 105);
    GUI_DispStringAt("Autocomplete:", 40, 145);
    GUI_DispStringAt("Previous word:", 40, 185);
}

void graph()
{
    // draw pressure graph
    GUI_SetColor(0x0);
    GUI_FillRect(0,100,240,320);
    GUI_SetColor(0xFF0000);
    GUI_DrawGraph(aY, GUI_COUNTOF(aY), 2, 100);
}

void draw_line_high()
{
    // highlight typed word
    GUI_SetColor(0xFF0000);
    GUI_DrawHLine(133,40,40+(letter_count+1)*8);
    GUI_DrawHLine(134,40,40+(letter_count+1)*8);
    GUI_DrawHLine(135,40,40+(letter_count+1)*8);
    int len;
    int i;
    for (i = 0; i < 16; i++){
        if(common_words[word[0]-97][i] < 97 || common_words[word[0]-
97][i] > 122){
            len = i + 1;
            break;
        }
    }
```

```
    GUI_SetColor(0x000000);
    GUI_DrawHLine(173,40,40+(len)*8);
    GUI_DrawHLine(174,40,40+(len)*8);
    GUI_DrawHLine(175,40,40+(len)*8);
}

void draw_line_low()
{
    // highlight auto word
    GUI_SetColor(0x000000);
    GUI_DrawHLine(133,40,40+(letter_count+1)*8);
    GUI_DrawHLine(134,40,40+(letter_count+1)*8);
    GUI_DrawHLine(135,40,40+(letter_count+1)*8);
    int len;
    int i;
    for (i = 0; i < 16; i++){
        if(common_words[word[0]-97][i] < 97 || common_words[word[0]-
97][i] > 122){
            len = i;
            break;
        }
    }
    GUI_SetColor(0xFF0000);
    GUI_DrawHLine(173,40,40+(len)*8);
    GUI_DrawHLine(174,40,40+(len)*8);
    GUI_DrawHLine(175,40,40+(len)*8);
}
```

Appendix B
8051 assembly code

```
.org 00h
ljmp main

.org 0023h
ljmp RIISR

.org 100h
main:
    lcall init
    mov A, #00h
    mov P1, #00h
    mov R0, #20h        ; time spent high
    mov R1, #01h        ; time spent low
loop:
    mov R2, 00h         ; reload and set high
    setb P3.2
high_loop:              ; countdown high voltage
    djnz R2, high_loop
    mov R3, 01h         ; reload and set low
    clr P3.2
low_loop:
    djnz R3, low_loop   ; countdown low voltage
    sjmp loop

init:
    setb ea             ; enable interrupts
    setb ES             ; enable serial interrupts
   mov   tmod, #20h       ; set timer 1 for auto reload - mode 2
   mov   tcon, #41h       ; run counter 1 and set edge trig ints
   mov   th1,  #0fdh      ; set 9600 baud with xtal=11.059mhz
   mov   scon, #50h       ; set serial control reg for 8 bit data
                          ; and mode 1
   ret

RIISR:
    lcall getchr        ; get byte over serial
    mov R1, A
    mov P1, A
    reti

sndchr:
   clr  scon.1            ; clear the tx  buffer full flag.
   mov  sbuf,a            ; put chr in sbuf
txloop:
   jnb  scon.1, txloop    ; wait till chr is sent
   ret
```

```
getchr:
   jnb  ri, getchr        ; wait till character received
   mov  a,  sbuf          ; get character
   clr  ri                ; clear serial status bit
   ret
```