Final Project -- Breathing Rate Detection System Integration

Objectives

1. Learn how to integrate all the components of the system to implement your team's breathing rate detection system that warns caregivers in the event of a potential problem
2. Test the system against the requirements using multiple test inputs to evaluate and debug operation.

System Integration

In the last lab exercise you designed three filters to determine breathing rates. You designed a low pass filter to detect breathing rates below 12 BPM. A Bandpass filter was designed to detect breathing rates between 12 BPM and 40 BPM. Finally, a high pass filter was designed to detect breathing rates over 40 BPM.

Measurement of the output energy of each of these filters was performed by viewing standard deviation of each filter output signal.

Now it's time to pull all the pieces of the system together. Recall the system block diagram in Figure 1. The block diagram includes a data acquisition subsystem, some signal pre-processing, a breathing rate detection function, warning logic and then finally an output for user signaling.

The signal preprocessing block consists of a lowpass filter to reduce high frequency noise beyond the highest rate expected rate of breathing. We do not expect to encounter breathing rates greater than about 70 BPM. Additionally, a signal equalizer is included to compensate for the high frequency roll-off of the temperature sensor.

After the breathing rate detection subsystem, a warning logic subsystem is included to compare the standard deviation of the energy from each filter and determine whether the breathing rate is too low, within the normal breathing rate range, or at a high rate of breathing indicating possible pneumonia.

Following the warning logic subsystem there is a user signaling block that will warn the clinician of an abnormal condition.

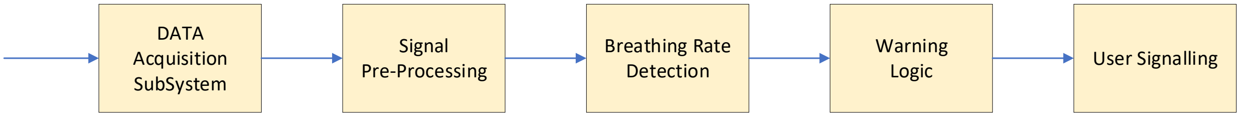


Figure 1 High Level Breathing Rate Detection System

Pre-Processing Subsystem Description

The preprocessing subsystem consists of a lowpass filter to reduce the amount of high frequency noise above the maximum expected breathing rate of 70 BPM. Either an FIR or and IIR filter can be used for this purpose.

In the equalizer lab we demonstrated an equalizer that acted as a differentiator to compensate for the frequency response of the temperature sensor. This was implemented using an FIR filter with just 4 samples in the impulse response [1, 1, -1,-1].

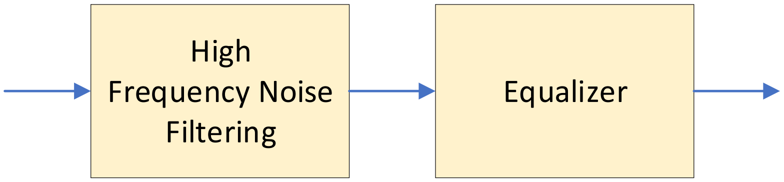


Figure 2 Pre-Processing Subsystem

Warning Logic Subsystem Description

To determine which filter has the most energy output, the standard deviations of each of the signal detector blocks are compared with one another. The filter bank with the greatest amount of output energy will most likely indicate which range the breathing rate is in.

There are several conditions that we wish to determine. We must first determine if the system is operational, that is, are the detectors detecting values above a certain low threshold. If the system is not operational, we need to warn the user.

Once we have determined that the system is operational, we need to decide which filter bank has the most output energy. The filter bank with the most output energy will most likely indicate the breathing rate range of the patient. Hence, a comparison of the energy output of each of the filter banks is used to determine the breathing rate (range).

A possible flow chart of this logic is shown in Figure 3. In the diagram, four operating states are defined:

0 – System operational and normal breathing rate

1 – System operational and low breathing rate

2 – System operational and high breathing rate

3 – System operation but rate is undetermined

4 – System not operational

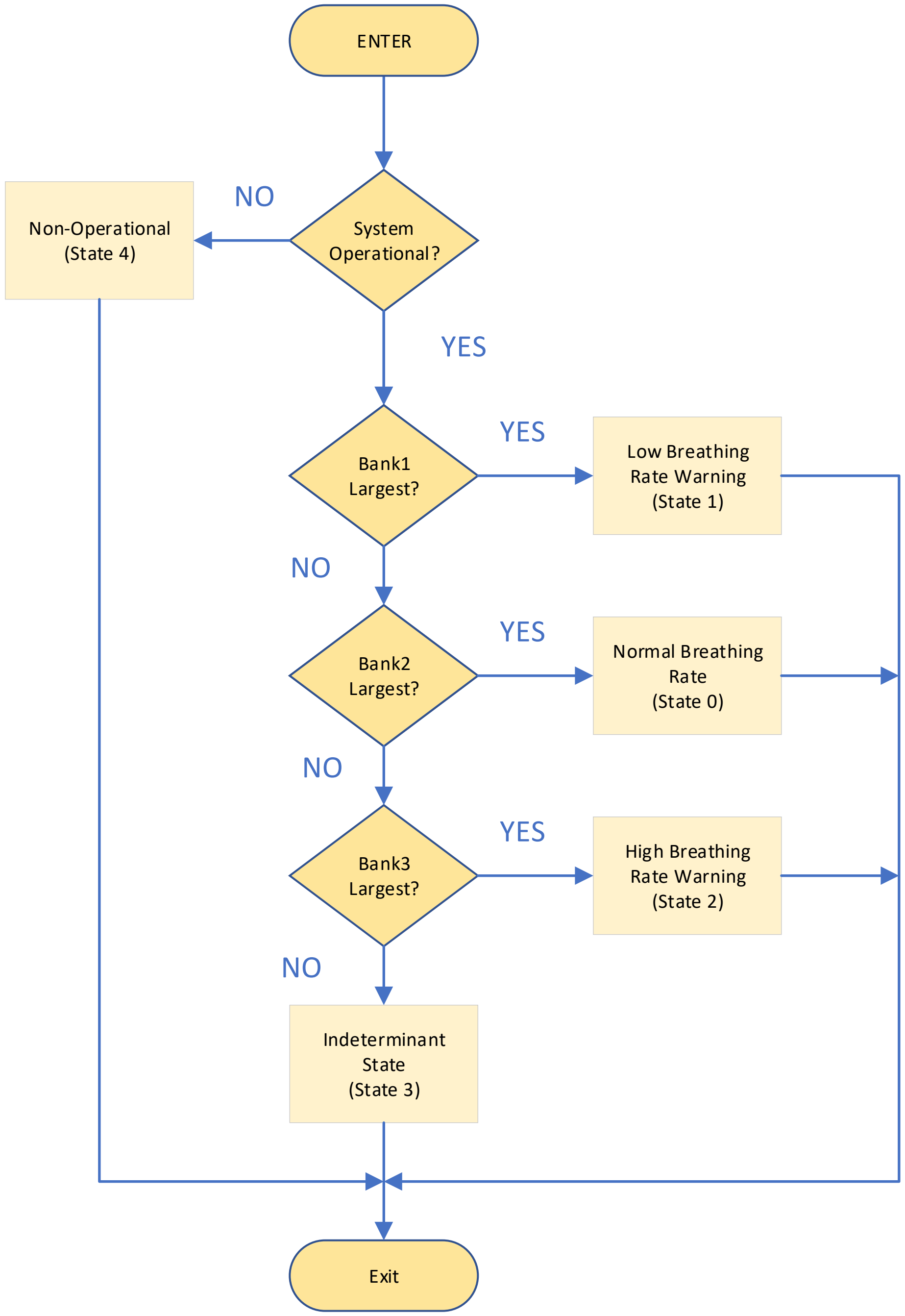

ENTER
System Operational?
NO
Non-Operational (State 4)
YES
Bank1 Largest?
YES
Low Breathing Rate Warning (State 1)
NO
Bank2 Largest?
YES
Normal Breathing Rate (State 0)
NO
Bank3 Largest?
YES
High Breathing Rate Warning (State 2)
NO
Indeterminant State (State 3)
Exit

Figure 3 Possible Warning Logic Flowchart

User Signaling Subsystem Description

The user signaling subsystem will signal an alert for an abnormal condition. The abnormal condition is when the system is not working properly. The alert for this condition is a continuous 200 Hz tone.

An alert will be sounded when the breathing rate is low (less than 12 BPM). The alert for this condition is a continuous 400 Hz alarm tone.

When the breathing rate is above 40 BPM, then the alert that is sounded is an intermittent 1000 Hz tone which is on for 1 second and off for 1 second.

When the breathing rate is in the normal range, no alert is sounded.

System Integration Steps -- Building on the Final Code Base

In the last lab, you started with a code base and added filters that you designed for detecting the various breathing rates. You tested these by checking the impulse response of each filter and with a swept tone in the time domain.

Structure of the C-Code for the Integrated System

In Week 01 we covered a flow chart of the processing system to gather input data and filter the data and then apply statistics to the output samples of each filter.

An updated flow chart is shown in Figure 4 below. Note the additional blocks shown in blue for the equalizer, the low pass filter, the warning logic and signaling subsystems.

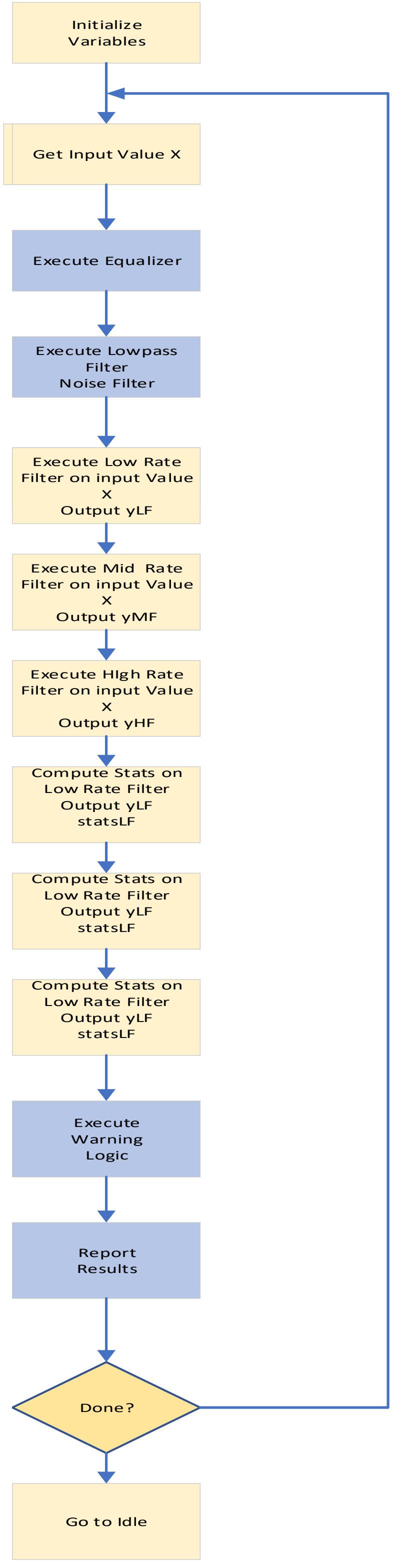

Initialize Variables
Get Input Value X
Execute Equalizer
Execute Lowpass Filter Noise Filter
Execute Low Rate Filter on input Value X Output yLF
Execute Mid Rate Filter on input Value X Output yMF
Execute HIgh Rate Filter on input Value X Output yHF
Compute Stats on Low Rate Filter Output yLF statsLF
Compute Stats on Low Rate Filter Output yLF statsLF
Compute Stats on Low Rate Filter Output yLF statsLF
Execute Warning Logic
Report Results
Done?
Go to Idle

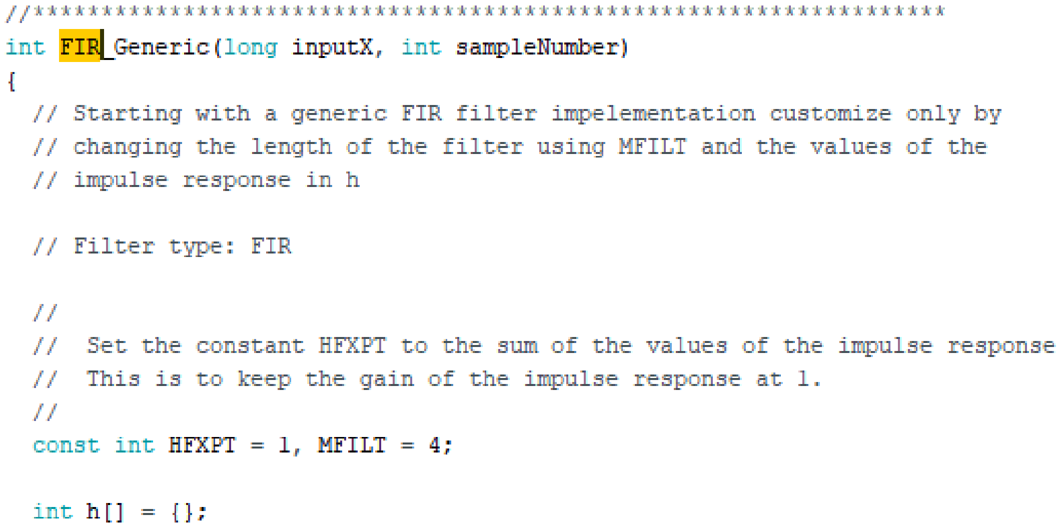
Figure 4 Breathing Rate System Flowchart

Step 1 – Add the Equalizer

The equalizer was designed and tested in the equalizer lab. It consisted of an FIR filter with only 4-taps: 1, 1, -1 -1.

In the code base there is a function called FIR_Generic. It is the same code that was used in earlier labs and implements an FIR filter. The impulse response, the length of the impulse response and the fixed point scaling factor need to be specified as shown in Figure 5. The impulse response is an array h, the length of the filter is the variable MFILT and the fixed point scalar value is HFXPT.

Copy or rename the function to create the equalizer and modify the impulse response, filter length and HFXPT scale value. Recall that the FIR filter operates using fixed point values for the input, output and the impulse response. HFXPT is normally the sum of the value of the impulse response for a low pass filter. In the case of the equalizer it is just 1.



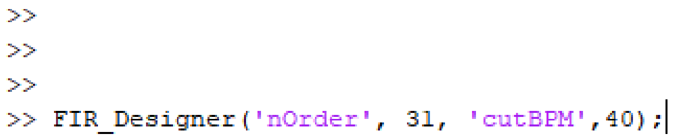
```
//*******************************************************************
int FIR_Generic(long inputX, int sampleNumber)
{
  // Starting with a generic FIR filter impelementation customize only by
  // changing the length of the filter using MFILT and the values of the
  // impulse response in h

  // Filter type: FIR

  //
  //  Set the constant HFXPT to the sum of the values of the impulse response
  //  This is to keep the gain of the impulse response at 1.
  //
  const int HFXPT = 1, MFILT = 4;

  int h[] = {};
```

Figure 5 C-Code to specify the FIR filter impulse response, length and scalar

Step 2 – Add a Low Pass (Noise) Filter

A breathing rate above about 70 BPM is very difficult to maintain, therefore, looking at values above this level is impractical. In order to minimize the possibility of high frequency noise entering the detection system at frequencies above practical levels, a low pass filter is needed. This helps reduce noise from the USB port and other activities that can corrupt the ADC input and reduces the possibility that the high rate breathing detector will detect and alarm on such noise.

The low pass filter can be implemented using either an FIR or an IIR filter. The IIR_Designer.mlx Live Script file or the FIR_Designer.m function can be used to create this filter. Recall that the FIR_Designer function is called from the MATLAB command window using the syntax shown in Figure 6. Note the example shown below creates a 31st order LPF with fc=40BPM, NOT the desired LPF for high frequency noise suppression).

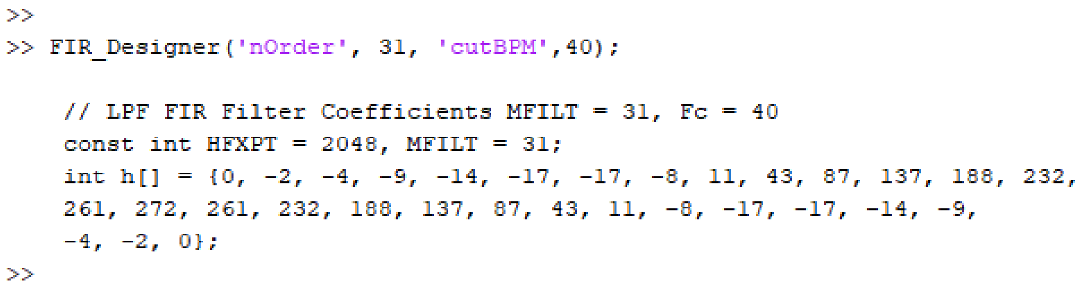


```
>>
>>
>>
>> FIR_Designer('nOrder', 31, 'cutBPM',40);
```

Figure 6 Syntax for Calling MATLAB function FIR_Designer

The two input parameters are entered using Name/Value pairs. The parameter nOrder is the length of the impulse response in samples and the parameter cutBPM is the filter ½ voltage value. It's not quite the same as the traditional filter corner frequency, where the corner frequency output value is 0.707 (or 70.7%) of the input value.

The filter generated by the call above is a windowed SINC LPF that was used in an earlier lab. The coefficients by default are fixed point coefficients. The filter length MFILT and the fixed- point scalar HFXPT are also computed. The output of the MATLAB function is C-code that can copied into the FIR_Generic C function. Recall that the function also plots the frequency response on a linear scale. This plot can be used to adjust the filter parameters until you achieve the desired response. The command window output looks like Figure 7:



```
>>
>> FIR_Designer('nOrder', 31, 'cutBPM',40);

    // LPF FIR Filter Coefficients MFILT = 31, Fc = 40
    const int HFXPT = 2048, MFILT = 31;
    int h[] = {0, -2, -4, -9, -14, -17, -17, -8, 11, 43, 87, 137, 188, 232,
    261, 272, 261, 232, 188, 137, 87, 43, 11, -8, -17, -17, -14, -9,
    -4, -2, 0};
>>
```

Figure 7 Command Window Output of the FIR_Designer function

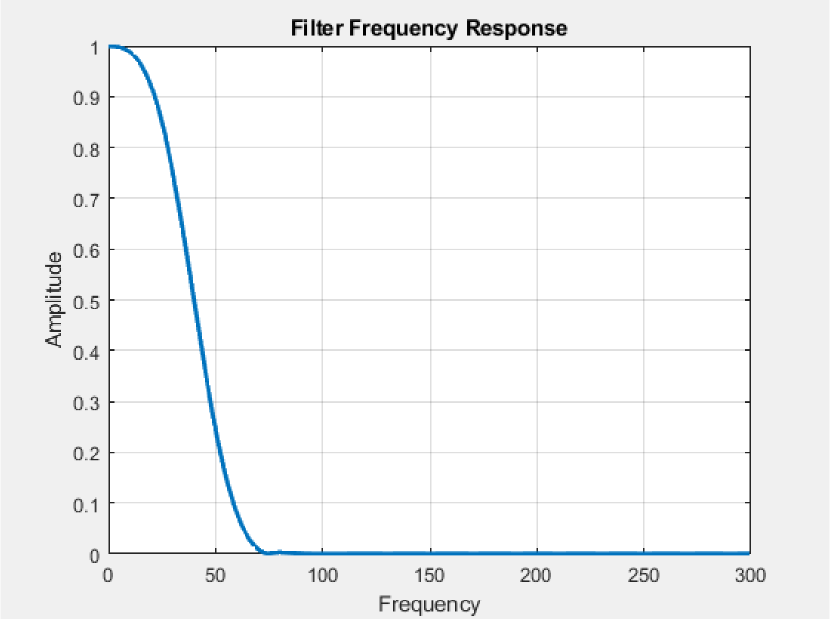


Figure 8 Example Frequency Response Graph Output from FIR_Designer

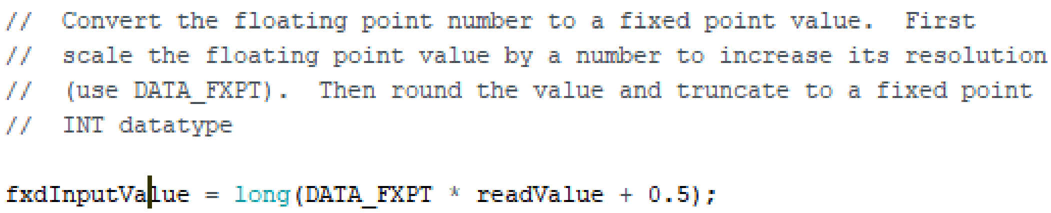
After deciding which filter to use (IIR or FIR) and determining the coefficients, copy and rename the selected filtering function in your C-code and set the correct coefficients in the code. Hint – Think about the cutoff frequency and filter order in terms of your overall system design and operation.

Step 3 -- Test the Equalizer and the Low Pass Filter

Test the Equalizer and the Low Pass Filter using the impulse response input and/or the Test Vector Generator input.

IMPORTANT – Recall that the FIR filter runs using fixed point variables, but the input to the function is a LONG. The output of the test vector generator and the impulse response are both FLOATs. So, you must convert the floating point value from the input to a LONG before calling FIR filters. Also, in the DSP Number Systems lab, we scaled the floating point number to retain precision by the variable DATA_FXPT. You must do the same here.

An example of converting the floating point number to a fixed point value is shown in Figure 9. Remember, when using the test vector generator or impulse test input, xv is the signal to be converted and not readValue.



```
//  Convert the floating point number to a fixed point value.  First
//  scale the floating point value by a number to increase its resolution
//  (use DATA_FXPT).  Then round the value and truncate to a fixed point
//  INT datatype

fxdInputValue = long(DATA_FXPT * readValue + 0.5);
```

Figure 9 Converting a Float to a Long Example

Make sure that you update the printArray variable and the numValues variable to send the correct data to the serial port.

Step 4 -- Write the Warning Logic Code

In this step you will write a function in C to determine in which state the system is operating. You can compare the outputs of the energy detector from each filter bank and make decisions on the state of the system. Have the function return the system state to the calling routine in the Loop function of the code. See one approach to the warning logic code in Figure 3. Your implementation may be different.

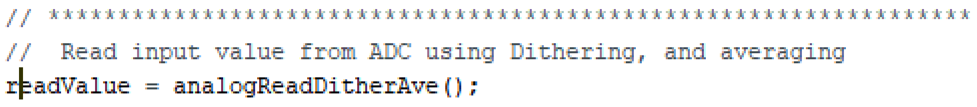
To determine if the system is operational you will compare the outputs of the filter banks to a threshold noise value that you will determine in the calibration step. Plan for using that threshold value in your warning logic code.

Step 5 -- Calibrating the Noise Level

In order to determine what standard deviation value indicates a valid signal, you will have to get a measurement of the amount of noise coming into the system and being detected at the output of the filter banks when there is no breathing. You can do this by capturing temperature data from the sensor when it is just sitting in free air.

To calibrate the noise level, you will need to integrate all the previous sections of code. That is, getting an input value, and then running it through the equalizer, the low pass filter, all three filter banks and the three running standard deviation calculations. The outputs of the standard deviation calculations are used to drive the warning logic. Make sure that you are using the correct data types to call functions, casting to a new data type when necessary.

Change the input to the system to read data from the sensor after dithering and averaging. Then, setup your output to gather data from the output of the filter banks and the standard deviation outputs.

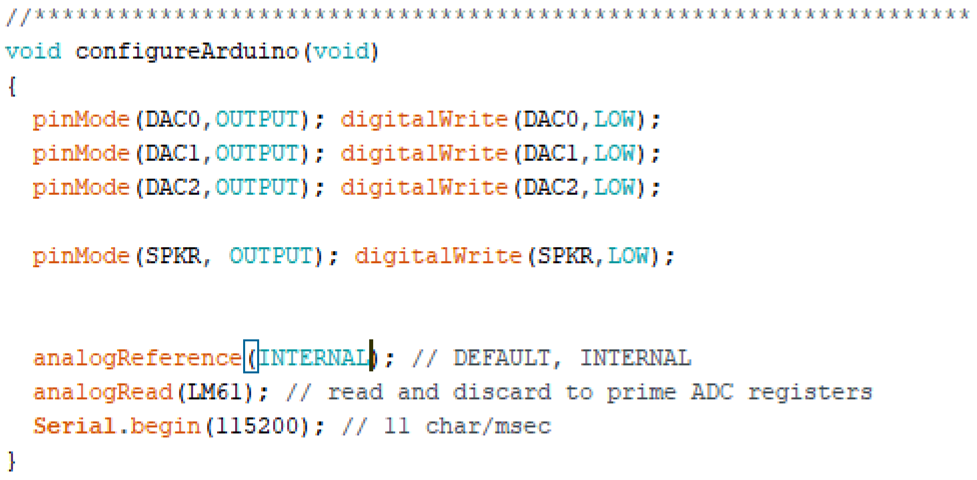


```
// ***************************************************************
//  Read input value from ADC using Dithering, and averaging
readValue = analogReadDitherAve();
```

Figure 10 Getting Input from the Temperature Sensor

VERY IMPORTANT – Make sure that the ADC is configured to use the INTERNAL voltage reference. This is done by calling the analogReference function with the argument INTERNAL. This is done in the configureArduino function.

This will greatly reduce the susceptibility of the ADC to power supply noise. It will also increase the relative level of the sample values out of the ADC compared to using the EXTERNAL or DEFAULT voltage reference.



```
//****************************************************************
void configureArduino(void)
{
  pinMode(DAC0,OUTPUT); digitalWrite(DAC0,LOW);
  pinMode(DAC1,OUTPUT); digitalWrite(DAC1,LOW);
  pinMode(DAC2,OUTPUT); digitalWrite(DAC2,LOW);

  pinMode(SPKR, OUTPUT); digitalWrite(SPKR,LOW);


  analogReference(INTERNAL); // DEFAULT, INTERNAL
  analogRead(LM61); // read and discard to prime ADC registers
  Serial.begin(115200); // 11 char/msec
}
```

Figure 11 Set the reference voltage to the ADC to INTERNAL

Capture enough data in MATLAB to determine a standard deviation value (threshold) that is considered noise. Signal levels above this value will be considered true breathing. Give yourself some margin to account for different systems and noise levels.

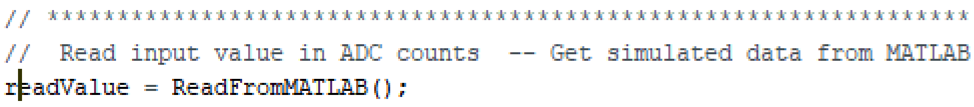
Step 6 -- Test the Warning Logic Code

To test the warning logic code, you will need to integrate all the previous sections of code. That is, getting an input value, running it through the equalizer, the low pass filter, all three filter banks and the three running standard deviation calculations. The outputs of the standard deviation calculations are used to drive the warning logic.

To test the integrated DSP system, it is often helpful to use a set of known test signals as the input, so that the system can be modified and the output compared to previous values achieved the same input. Then, testing can proceed with the real input signal from the temperature sensor.

There is a function in the base code that allows you to integrate with MATLAB to provide data to the Arduino system and retrieve data back from the system over the serial port. The C function in the Arduino is ReadFromMATLAB.

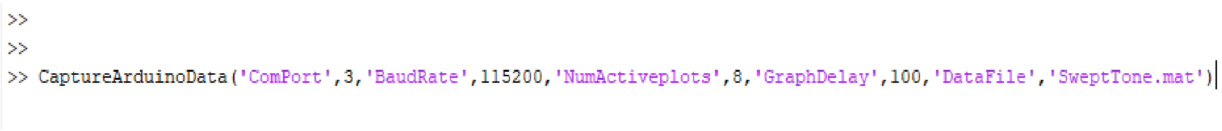
To integrate with MATLAB, comment in the call to the ReadFromMATLAB to retrieve a data value on the Serial Port. Comment out the call to analogReadDitherAve.



```
// ******************************************************************
//  Read input value in ADC counts  -- Get simulated data from MATLAB
readValue = ReadFromMATLAB();
```

Figure 12 Call to ReadFromMATLAB function

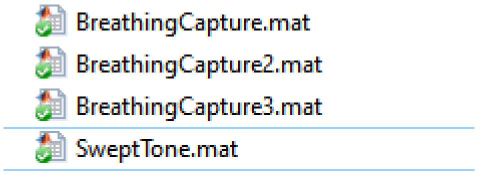
On the MATLAB side, in the command window, use the CaptureArduinoData.m function and add the Name/Value parameter ‘DataFile’ as shown below.



```
>>
>>
>> CaptureArduinoData('ComPort',3,'BaudRate',115200,'NumActiveplots',8,'GraphDelay',100,'DataFile','SweptTone.mat')
```

Figure 13 MATLAB Call to the CaptureArduinoData function with 'DataFile' Parameter

The parameter following ‘DataFile’ is the file name where the data will be drawn from. There are four files with test data available on myCourses. These are shown in Figure 14.



BreathingCapture.mat
BreathingCapture2.mat
BreathingCapture3.mat
SweptTone.mat

Figure 14 MAT Files with Test Breathing Data

Start with testing with the “SweptTone.mat” file. This is a tone that is swept from 1 BPM to 60 BPM. The data has been integrated so that it has the same frequency response as the temperature sensor but contains no noise. This is a good first step to get things working and check your system.

Use this file and make modifications to your algorithm until you are satisfied that your system is working correctly. Once you are satisfied, try the other breathing data files for verification.

* NOTE – The Final System Integration Steps shown below (steps 7 through 10) are DRAFT only. I will clarify as needed and post a finalized document before the start of week 14 (Project Week 3) *

Step 7 – Add the Piezo Speaker to the hardware

Your class kit came with a small piezo electric speaker. The speaker requires a series 100 ohm resistor to limit the current drawn from the Arduino output pin.

Connect the output of Pin 12 on the Arduino to one side of the 100 ohm resistor. Connect the other side of the resistor to the either terminal of the speaker. Connect the other side of the speaker to ground.

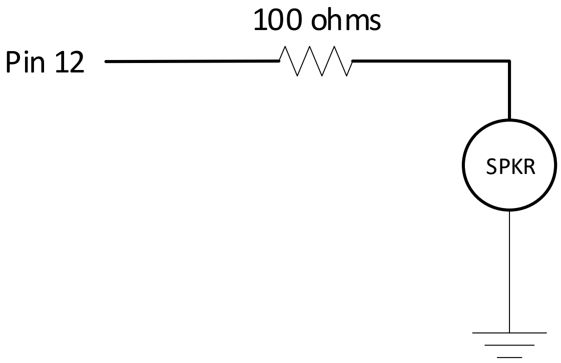


Figure 15 Piezo Speaker Schematic

Step 8 – Write the User Signaling Code

In this step you will write a function in C to signal the user.

The user signaling subsystem will signal an alert for an abnormal condition. One abnormal condition is if the system is not working properly, that is getting nothing out of the system above the noise threshold. The alert for this condition is a continuous 200 Hz tone.

An alert will be sounded when the system is working, and the breathing rate is low or less than 12 BPM. The alert for this condition is a continuous 400 Hz alarm tone.

When the breathing rate is above 40 BPM, the alert that is sounded is an intermittent 1000 Hz tone which is on for 1 second and off for 1 second.

When the breathing rate is in the normal range then no alert is sounded.

There is a built-in function to generate a tone in the Arduino, that is tone(). However, this function uses Timer2 in the processor to generate the tone. This conflicts with our use of Timer2 as the interrupt timer and will not work.

To generate a tone that is compatible with our use of Timer2 for the interrupt, a modified tone library function will be used. This tone library is based on the function found here

https://github.com/bhagman/Tone

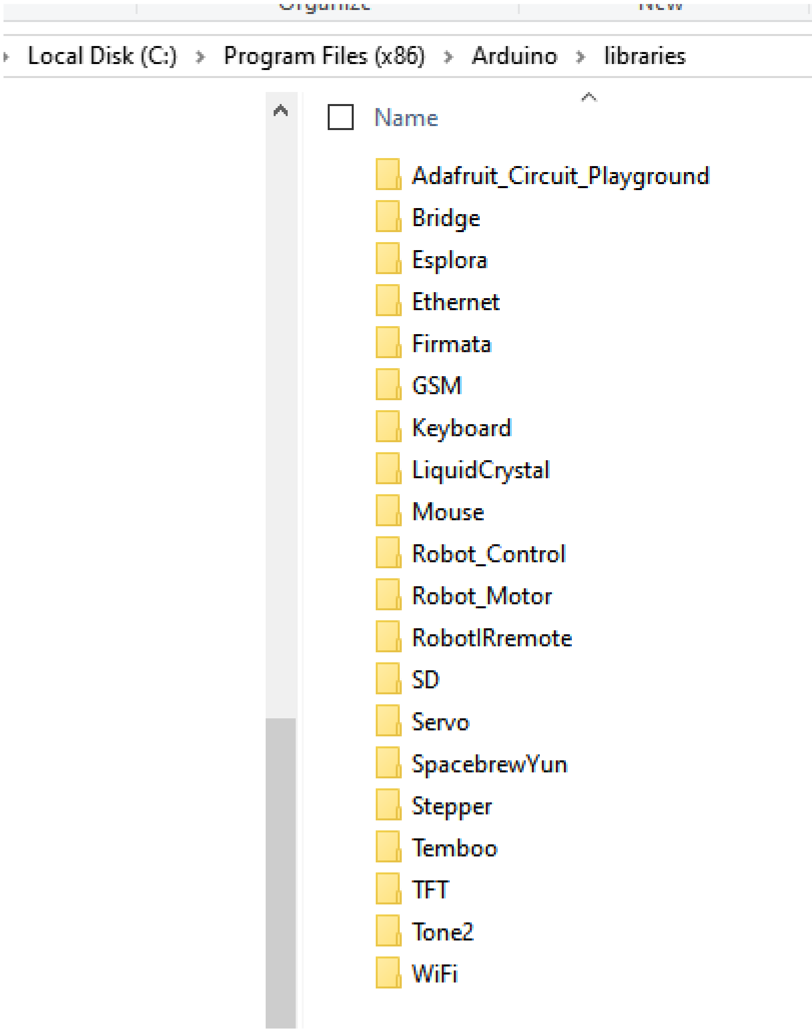
This library can use timers other than just Timer2. We want to use Timer1 so as not to interfere with the interrupt.

The library has been modified to be called Tone2 so not to conflict with the built in Arduino tone() function. Several functions have been modified to work with the system. Use the files from myCourses. Do not download and use the files from the github repository shown above. This for documentation reference only.

Create a directory called Tone2 in the libraries directory for you Arduino. This is typically in a directory such as:

C://Program Files (x86)/Arduino/libraries

Download the files for the library from myCourses. Place them in the libraries directory for your Arduino.



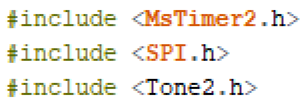
Local Disk (C:) › Program Files (x86) › Arduino › libraries

□ Name

- Adafruit_Circuit_Playground
- Bridge
- Esplora
- Ethernet
- Firmata
- GSM
- Keyboard
- LiquidCrystal
- Mouse
- Robot_Control
- Robot_Motor
- RobotIRremote
- SD
- Servo
- SpacebrewYun
- Stepper
- Temboo
- TFT
- Tone2
- WiFi

Figure 16 Location of Tone2 Library

In your C-code you will need to include the library



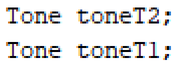
```
#include <MsTimer2.h>
#include <SPI.h>
#include <Tone2.h>
```

Figure 17 Include statements to include the Tone2 Library

The library creates a Tone object and assigns them to timers in the order of Timer2, Timer1 and Timer0 for this Arduino.

We must create a Tone object for Timer2 first then for Timer1. The Tone object for Timer2 will not be used, just created. In this way it will not interfere with the interrupt on Timer2.

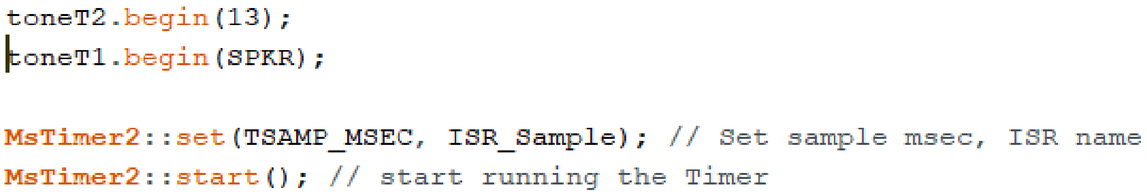
In the beginning variable declaration portion of the code above “void Setup”, include these two lines to create two Tone objects.



```
Tone toneT2;
Tone toneT1;
```

Figure 18 Creating two Tone objects

In “void Setup” use the begin method to setup the two tone objects. The object toneT2 has to be “begun”, but will not be used. Only the object toneT1 will be “begun” and used. MAKE SURE THAT THESE TWO METHOD CALLS OCCUR <u>BEFORE</u> SETTING UP MSTimer2. Otherwise the interrupt will not be set correctly.



```
toneT2.begin(13);
toneT1.begin(SPKR);

MsTimer2::set(TSAMP_MSEC, ISR_Sample); // Set sample msec, ISR name
MsTimer2::start(); // start running the Timer
```

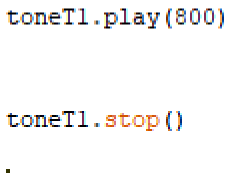
Figure 19 Starting the Tone Objects toneT2 and toneT1

The argument to the “begin” method is the pin used to output the tone. Previously in the code, the value SPKR has been set to pin 12. Pin 13 which is the output for toneT2 will be left unused.

DO NOT create a third timer object as this will create one using Timer0 which will interfere with other Arduino operations.

To play a tone use the “play” method with the frequency as an argument. To stop the tone use the “stop” method as shown in Figure 20.

DO NOT use the toneT2 object as this will interfere with the interrupt.



```
toneT1.play(800)


toneT1.stop()

.
```

Figure 20 Using the play and stop methods of the toneT1 object

Other methods that can be used are shown in the documentation section in the github repository shown earlier.

Once you have your signaling system written, integrate it with the rest of the code and test the system. You can use the MATLAB files such as SweptTone.mat and the three BreathingCapture files as test inputs.

Step 9 – Test the Entire System with Real Inputs

Finally test the system with real breathing data. Set the input of the system to read from the temperature sensor after dithering and averaging. Verify that your system works and is reliable. Test that it can detect a broken system (no breathing), low rate breathing, high rate breathing and normal rate breathing. Verify that the output alarms are as specified.

Use all the tools that you have developed and worked with to verify the correct operation of the system.

Step 10 – Continuous Operation

The system is intended to run without interruption and without intervention. Modify the portions of the code that require a handshake to get started and cause the code to stop after a specified number of samples have been taken.

Report Requirements

Be sure to review the report requirements for the final project. They are shown in the Project specification document. The report must be in the correct IEEE format to be accepted.