Liam Mulshine and Shaan Desai

ES 52: Final Project Report

The Fast Reacting Heart Rate Monitor

Introduction

We started our project planning to design a pair of wearable monitoring/display devices, one of which would sense useful information about the condition of a child, such as heart rate, motion, and room temperature and transmit this information using an Xbee, while the other would display this information, presumably to a parent/guardian, through a digital display or haptic feedback. Although ultimately we needed to narrow our scope – implementing just the heart rate sensor and display - our project still encompassed a wide range of interesting and relatively complex analog and digital circuitry.

Since we utilized the TCRT1000 infrared sensor to detect heart beats, the input signal to our circuit had a DC offset that was subject to sudden changes if the sensor lost contact with the skin, sensed sudden peaks of light from the outside world or experienced other disturbances, such as general motion. It is also a consequence of using the TCRT1000 that the voltage of our HR signal was at best approximately 20mV peak to peak. Given this, our goal was to get rid of the DC offset to ensure we could add appreciable gain to our HR signal without exceeding the input common mode range of our operational amplifiers. The common strategy employed to address the DC offset issue is to pass the signal through a high pass filter before adding gain (just as we did in Lab 4). Yet, since the critical frequency for such a filter is so low, large resistor and capacitor values are needed, creating a rather large RC time constant (4-5 seconds due to the fact that the critical frequency is ~6Hz). The circuit cannot output valid data until the DC offset has diminished to zero, so each time that the DC offset rapidly changes (for example, if someone removes the sensor from his or her skin to adjust it, hiking the DC offset up to approximately 9V when removed and then lowering it to approximately 1.5V when reapplied) the capacitor would take time (as long as 6 seconds) to charge/discharge before fully attenuating the DC offset to zero. Thus, passing the signal through a low pass filter has the significant disadvantage of adding a large time delay to the circuit, which is not practical in real life situations where disturbances to the sensor are commonplace. We set out to fix this problem using modulation.

Characterizing TCRT1000

We had few options when it came to choosing a component to sense heart rate. Light sensors, such as the TCRT1000, are very commonly used in this application. The TCRT1000 consists of an LED, which emits infrared light, and a phototransistor BJT, that allows base current to flow in proportion to the amount of light that enters the phototransistor. When one applies the TCRT1000 to his or her finger, ear, etc., outside light is blocked out, so essentially the only light that the phototransistor picks up corresponds to infrared light that is reflected off of the skin. Every heartbeat causes blood to rush into the finger, ear, etc., which causes more light to be reflected back into the sensor, spiking the base current and, thus, causing the collector voltage to drop. We can sense these spikes/drops, and use them to calculate a heart rate in beats per minute.

The LED within the TCRT has an absolute maximum current of 50 mA. We thus need a current limiting resistor between power and the LED to avoid overcurrent. Since 50mA was the maximum, we characterized our design so that 45mA could run through it.

According to the datasheet, if 45mA runs through the TCRT LED, there is a 1.2V drop across the LED.

Since we were supplying our circuit with +/- 9V we chose resistor, R1, such that

$$9-1.2 = 0.045*R$$

$$R1 = 173\ \Omega$$

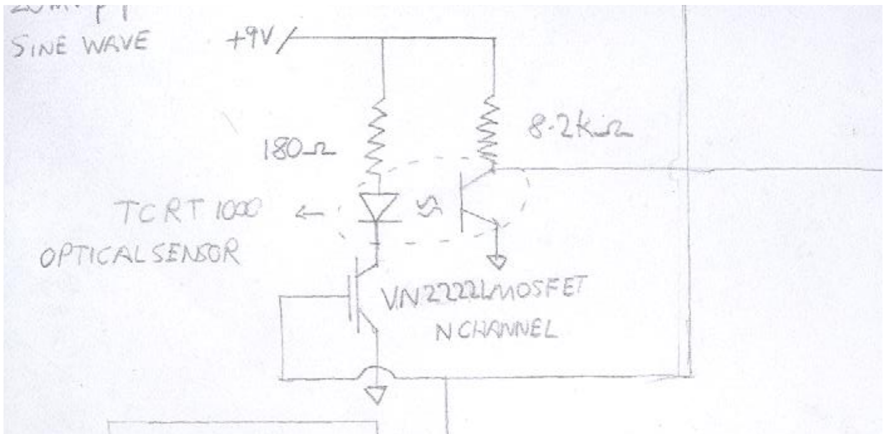
We rounded up R1 to a 180 Ω standard resistor value; because a higher R meant that we didn't risk overcurrent even with possible resistor tolerances.
The voltage at the transistor is dependent upon the base current. When the sensor is not in contact with the skin, no current flows through the transistor, because not enough IR light is reflected into the phototransistor for base current to flow. Therefore, it is pulled up to 9V. When IR light is reflected into the phototransistor base current flows; collector/emitter current is proportional to base current. We chose R2 to be big enough to cause a noticeable voltage drop when small HR pulses spike the base and collector current, while also ensuring that the voltage dropped across the resistor does not cause the collector voltage to be < 0.1V greater than the emitter voltage, which would cause the BJT to saturate, causing us to lose the AC component of our signal corresponding to the heart beat. The Datasheet states that when there is a current of 50mA through the LED, and there is a 1V drop across the transistor then 1mA flows through through the transistor. Given these values we chose R2 to be such that:

$$9 - 1 = 0.01 * R2$$

$$R2 = 8\ k\Omega$$

Once again we need to ensure we don't face any overcurrent issues, therefore we set R2 to a standard resistor value that was higher, so R2 = 8.2KΩ.

Below is a schematic of what was built:



Modulating the Signal

The purpose of modulating the heart rate signal generated by the TCRT to a higher frequency was to ensure that we could remove the DC offset in the signal, without using an RC circuit with a long time constant. To modulate the signal we needed an oscillating circuit. We chose our frequency of oscillation to be 1000Hz but we needed to generate this signal, a detailed explanation of how this was done follows.

Oscillating Circuit

We used a chopper circuit to modulate our low frequency heart rate signal up to a high frequency signal, which varied between 9V and the output voltage of the TCRT (1.5V DC with 20mV peak to peak AC component when sensor was applied to the skin). We did so by placing a fast switching N-Channel MOSFET (the VN2222L) between the cathode of the TCRT's IR LED and ground.

Initially, we approached our modulator circuit without thinking about how we would demodulate the signal later on. We simply generated a 1 kHz clock signal, that oscillated between 0V and 5V with a 50% duty cycle using the astable mode of the LMC555 and connected this to the gate of the MOSFET in our modulator circuit. However, we found while we were attempting to demodulate our signal that we needed another clock signal that was exactly in sync with the modulator clock, but had a 25% duty cycle instead of a 50% duty cycle (we go into more depth on why this is the case later in the report).

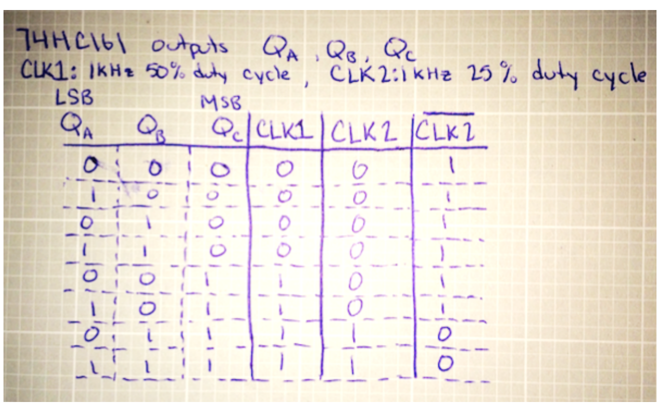
To generate these clock signals, we utilized the 74HC161 counter with asynchronous clear (we did not have access to a '163 in the lab, so we used *LD to reset the counter, instead of CLR.), the LMC555 to generate a 8kHz clock signal for the counter, and combinatorial logic to create two 1kHz clock signals, one used for modulation with a 50% duty cycle and the other used for demodulation with a 25% duty cycle.

We needed to first generate a clock signal with which to feed the '161, and we did so using an LMC555. We determined that the frequency of this signal needed to be 8 kHz, so that we could use just the first three output pins of the '161 counter and combinatorial logic to generate both the 50% duty cycle and 25% duty cycle 1 kHz square waves.
To generate the 8kHz square wave that fed into our counter, we used the astable mode of the LMC555 with R = 91kΩ and C = 1nF, following the topology depicted in the circuit

diagram below. We chose such a small capacitor value so that we could use a larger resistor and increase the power efficiency of the circuit, by causing it to sink less current.

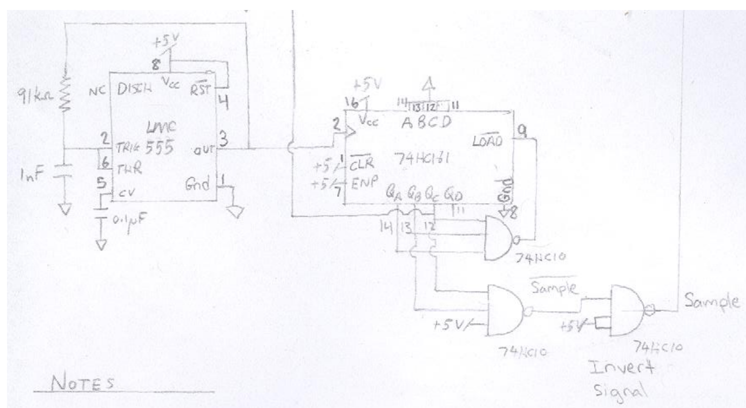
Output C on the ‘161 divides the CLK signal by 8, so we used that output as our 1kHz, 50% duty cycle modulator signal. We used a 3-input 74HC10 NAND gate with one of its inputs pulled high, another input connected to output B and the last input connected to output C to generate our 1kHz, 25% duty cycle active low demodulator signal (the signal was active low because we used a NAND gate). Since this signal was active low when the modulating signal was active high, we had to invert this signal using another NAND gate. See truth table, timing diagram, and schematic of this circuit below:

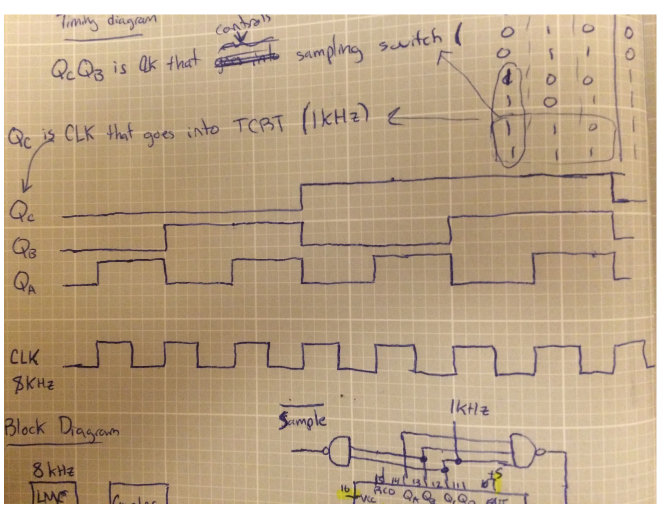
Truth Table


74HC161 outputs Q_A, Q_B, Q_C
CLK1: 1kHz 50% duty cycle, CLK2: 1kHz 25% duty cycle

Q_A (LSB)	Q_B	Q_C (MSB)	CLK1	CLK2	$\overline{CLK2}$
0	0	0	0	0	1
1	0	0	0	0	1
0	1	0	0	0	1
1	1	0	0	0	1
0	0	1	1	0	1
1	0	1	1	0	1
0	1	1	1	1	0
1	1	1	1	1	0

Schematic

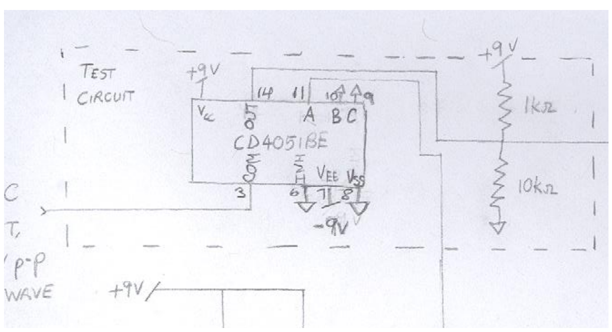




Timing Diagram

Test Signal

It turns out that for testing purposes, having someone place their finger on the TCRT every time we wanted to test our signal was not practical, since it would involve someone sitting still for a very long time, and the TCRT signal was not particularly consistent or distinguishable from noise prior to filtering and adding gain to the signal. Therefore, we designed a test circuit to replicate our heart rate signal, which later helped us with debugging. Just as you will see later on in our demodulating circuit, we can use a multiplexor to switch between our signal and 9V since the multiplexor can act as a switching circuit. We used this device because it was the only part we had in the lab that could act as switch that operated at ±9V, but was also able to interpret logic levels of +5V and 0V. The schematic diagram will help to better inform us of how the test circuit works, see next page.



By feeding our input (a 1V DC offset sine wave of 20mV peak-to-peak which replicates our heart rate from the TCRT1000) into the COM line and holding INH grounded so that the switch at INH was always connected, we were able to regain our signal at Ch 0. We then controlled the switching by feeding our 1kHz square wave into A. The reason behind this is that A controls the first bit and therefore, if A were to be high, the angular switch would move to Ch 1 and if it were 0 it would move to Ch 0. This switching allows us to control the output pin (Ch 1) value. This output is then connected to the middle of a voltage divider so that when the switch is open, we get approximately 9V between the 1kΩ and the 10kΩ resistors. When the switch is closed, due to Thevenin (in this case working in our favor!) the output in between the two resistors will be the value of our signal. Therefore, our test signal oscillates between 9V and our HR signal, which is exactly the type of signal our modulated TCRT circuit generates. See the figure below for what this looks like on the scope:

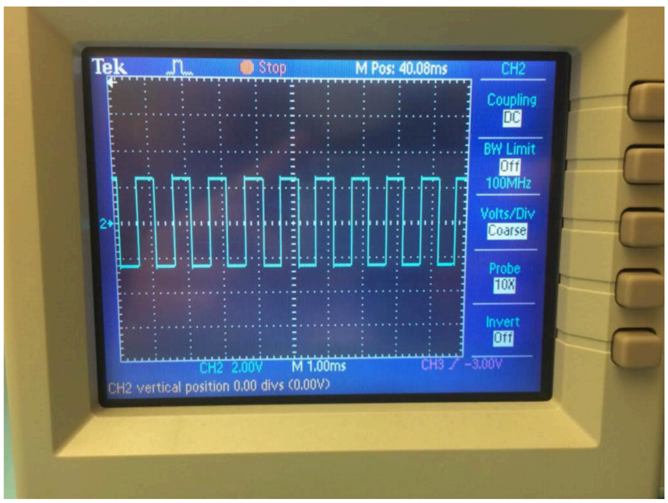
High Pass Filter

Given that we had a signal modulated up to 1000Hz that oscillated between 9V and ~1V we wanted to filter this signal to get rid of its DC offset. We quickly jumped to $f_c = \frac{1}{2\pi RC}$,

and set f_c to 450Hz so that, not only could we get rid of the DC offset, but also could we ensure that we were strongly attenuating frequencies below 900 Hz.

Using this we obtained the value of C to be 1nF and R to be 350kΩ. To see whether this was correct we tested our circuit and saw significant "drooping" on our output signal. The reason behind this drooping was that the RC value was too small which led to rapid discharging between peaks, but a small RC is exactly why we wanted to modulate the signal. Since the RC time constant was more important to our design than actually filtering out lower frequencies, we characterized our high pass filter so that RC was small enough that the circuit was still fast but that it also ensured we had little or no drooping on the output signal. It was a bit of a balancing act. Eventually, after trial and error we came to the conclusion that with R = 160kΩ and C = 220nF we had an RC value of 35ms, which was still much faster than the 5-6 seconds we obtained in lab 4, but also did not cause us to notice much drooping within our signal. The f_c value obtained using these values was 4.5Hz which was $1/100th$ our initial f_c. We paused here to ask ourselves to what extent we needed high pass filtering to get rid of noise at low frequencies, and came to the conclusion that most of the noise we would expect would be at high frequencies due to factors such as power supply fluctuations and motion of the TCRT1000, and, therefore, we didn't focus on high pass filtering much after this step.

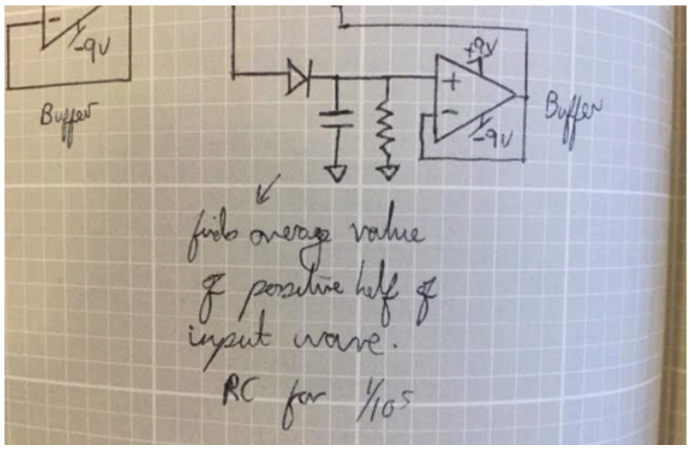
Given this new signal which had no DC offset, we put this signal through a unity gain buffer to ensure we wouldn't have Thévenin issues later in our design process. The scope shows our signal after passing through the high pass filter and unity gain buffer:



Centering our Modulated HR Signal

Once we had this signal we realized that the actual heart rate signal was at the bottom of the square wave, which was offset by approximately -2.1V DC. Therefore, we wanted to move this signal up to 0 V so that we could demodulate the signal later on. However, in order to do this, we couldn't simply add 2.1 V to our input signal because the -2.1V was only characterized for one of our fingers. Different skins reflect light to a different degree, and, thus, different fingers/ears would change the initial DC offset of the input signal, which would change the peak-to-peak value of the buffered heart rate signal. Given these fluctuations we needed a circuit to detect the value we needed to add to our signal. David suggested we use an envelope detector circuit, which would chop off the negative half of the input signal using a schottky diode, and find the average value of the top half of the signal using an RC circuit with a time constant somewhere between 0.001 seconds and 0.2 seconds, corresponding to a frequency range of 5Hz to 1kHz.

The following diagram displays this envelope detector circuit:

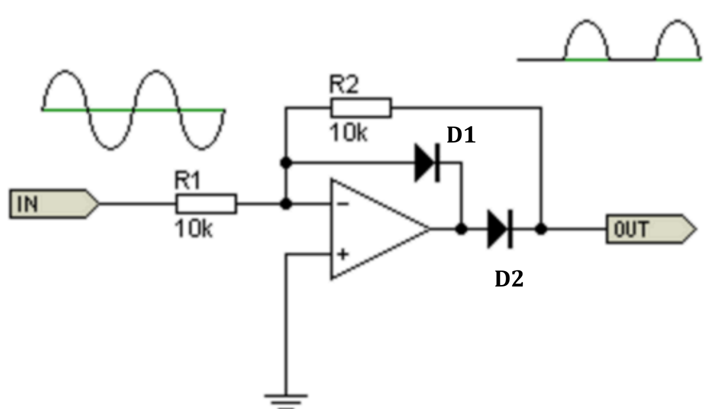


As Avi explained to us, the point of adding an RC circuit (a resistor and capacitor in parallel to ground) after the schottky diode is to essentially trace the signal. When current flows through the diode, the capacitor begins to charge at a rate governed by the value of R and C. When no current flows through the diode (when the signal is < 0V), the capacitor begins to discharge. The rate of discharge needs to be set such that the capacitor does not discharge too much in between closely spaced positive voltage peaks in the signal. Therefore, we chose RC for 10Hz (a 0.10 second time constant), which is much slower than the 1000Hz rate at which the signal oscillates, which ensured that we would get the average value of the positive portion of the signal without much drooping or discharging between the times when current flows through the diode. With RC set to 0.10s, we chose the value of R to be 430kΩ and C to be 220nF.

In testing our design the output signal was very good at detecting the envelope, but it had a 0.2V offset from the top because of the 0.2 V drop across the schottky diode. This meant that once we added this value back to our original signal and demodulated it, we would have a DC offset that would make it difficult to add gain to our signal without adding gain to the DC offset which would cause our signal to hit the positive rail of our amplifying op amp. Therefore, we needed a circuit that could perform the same task without creating a voltage drop of 0.2V. We thus chose to use a precision rectifier diode circuit.

Precision Rectifier Diode

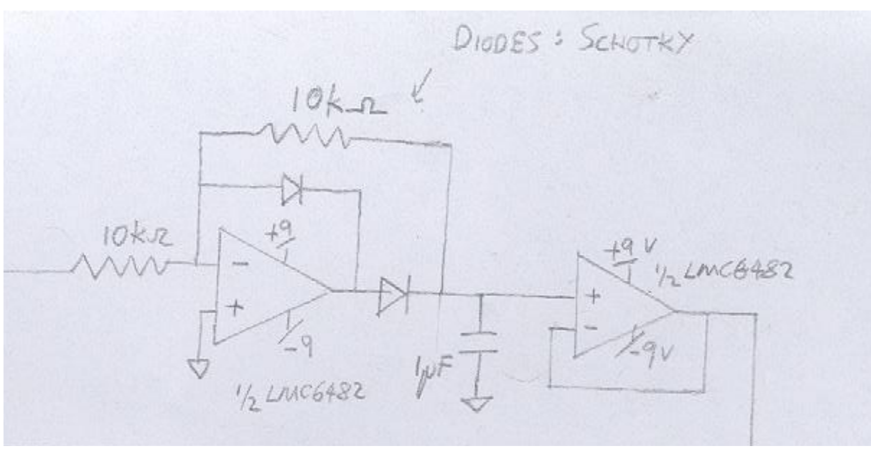
The schematic for a simple half-wave precision rectifier diode is shown below:



Essentially, when the input signal to the opamp is negative, the circuit acts much like an inverting op amp with a gain of 1. The op amp output goes positive, so diode 1 (D1) shuts off, and diode 2 (D2) turns on. Due to the negative feedback loop, the output of the opamp is set such that the voltage at OUT (at the cathode of D2) is equal to the magnitude of the input signal voltage.

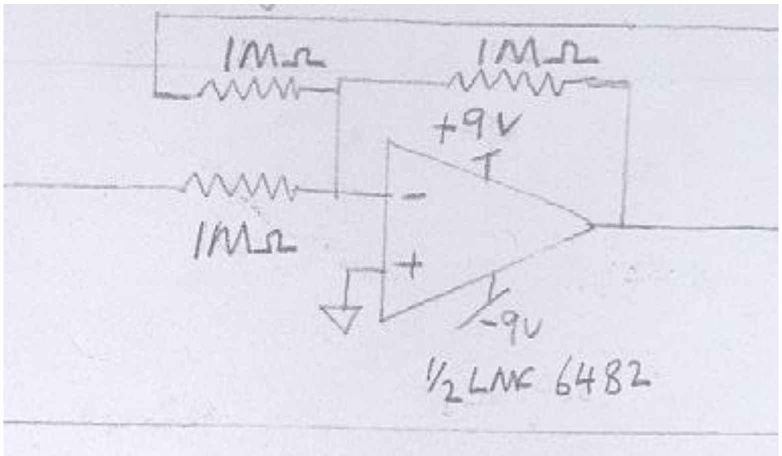
When the input signal goes positive, the output of the opamp goes negative, shutting D2 off and turning on D1. Since the diode is essentially a short circuit when on, current will flow through D1 and bypass R2 entirely. This means that no voltage is dropped across R2 and the OUT voltage equals 0V.

Therefore, we see that when the input signal is negative V_out = - V_in and when the input signal is positive V_out = 0V. However, we want the positive portion of the modulated HR signal. To address this, we simply inverted our signal before passing it through the rectifier diode. See the schematic below:



Adder Circuit

Once we had an average value of the top half of our input signal from the precision diode, our goal was to take our original DC buffered signal and add this value to it so that we could shift our ‘meaningful’ signal so that it was centered around 0V. We decided to use an inverting adder as seen in the schematic below.

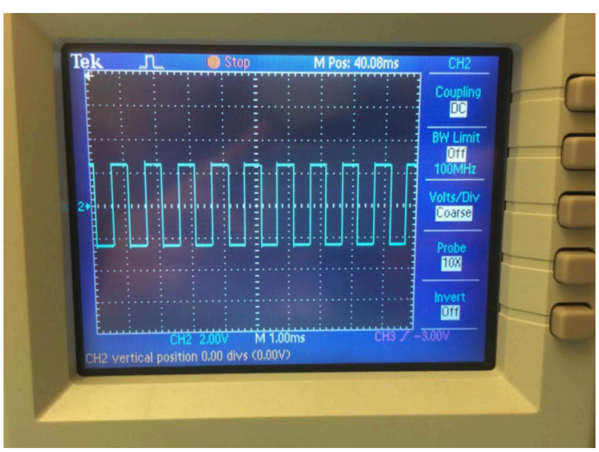


Notes:

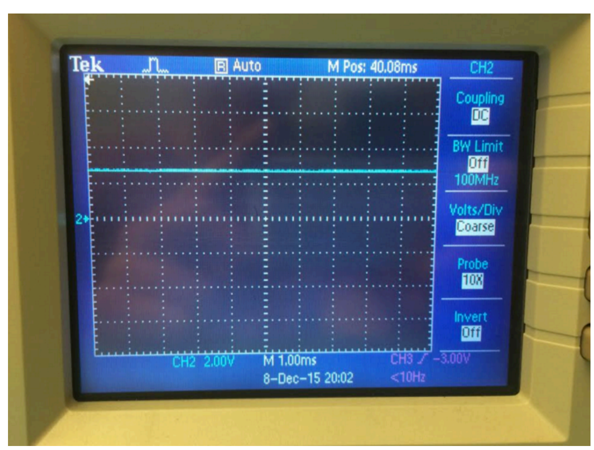
1. Inverting the signal doesn’t affect our ability to detect the HR since it takes the form of a sine wave.
2. The use of 1MΩ resistors was to reduce current drawn, and thus improve on power consumption.

As a reminder one of the inputs to the adder is our original buffered signal after the high pass filter while the other input is the average value of the top half of this input, namely it is the peak voltage of this input. See the figures below for what these inputs look like:

Modulated HR signal without DC offset

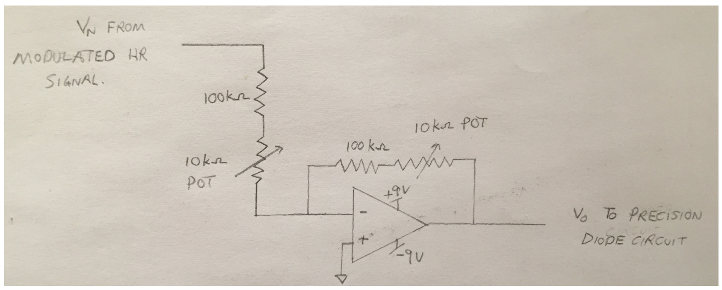


Average peak voltage of HR signal



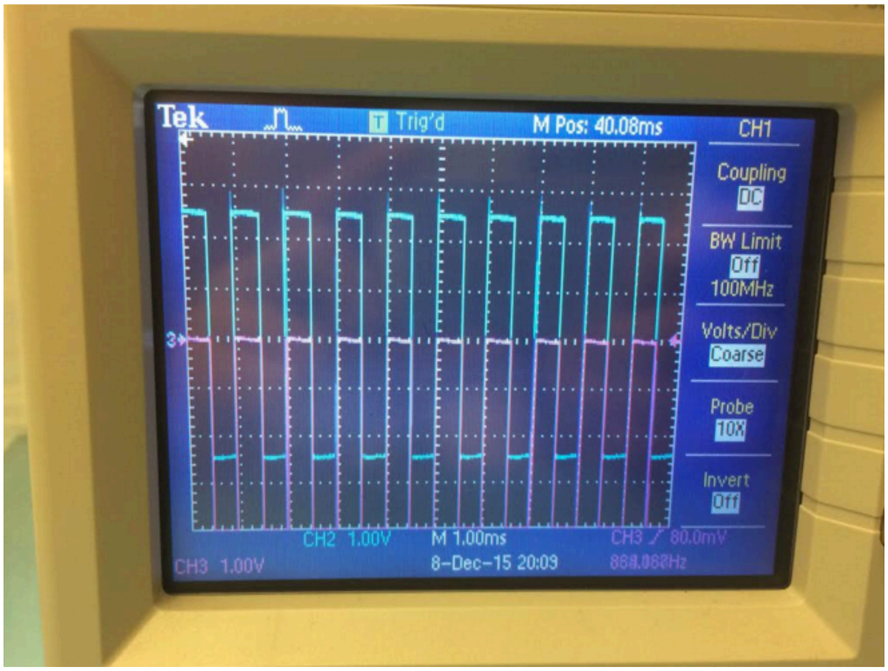
Reducing DC Offset to Zero

We noticed that even with the precision diode, our signal out of the adder had a slight DC offset, which we attributed to component tolerances. To account for this, we added variable resistors and resistors to the inverting amplifier that fed into the precision diode, as shown in the schematic below:



Adjusting the potentiometers within this circuit allowed us to modulate the gain of the amplifier between 0.91 and 1.1. We could thus adjust the DC level of the signal that the envelope detector circuit generated. We could adjust the signal until the upper portion of the signal generated by the adder circuit described above was centered on 0V. Note: The final schematic on the last page of this report (the same schematic attached in our lab books) has a mistake. The inverting pin of the op amp should be connected to the bottom of the variable resistor just as the schematic above illustrates not in between the 100k and 10k potentiometer.

The output waveform of the inverting unity gain adder after adjusting the potentiometers appropriately can be seen below, where the purple (channel 3) illustrates the output of our adder circuit and channel 2 shows us the input:

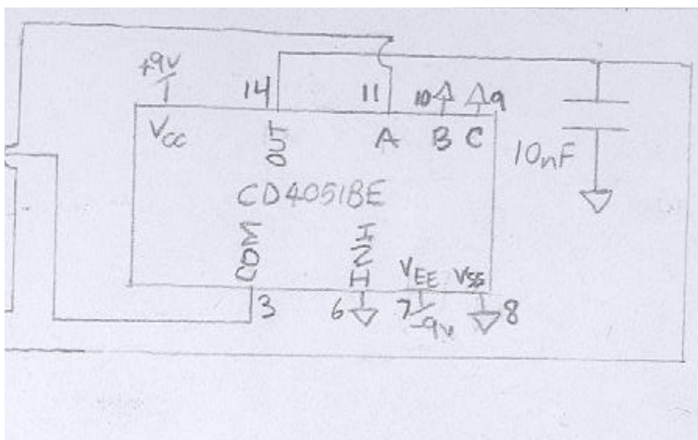


Demodulation Circuit

Once we had reduced the DC offset in our signal to 0V, we needed to regain our original signal so that we could filter out noise with low pass filters, and use a comparator to give us a digital heart beat signal. Yet, before all of this, we needed to demodulate our modulated signal. To do so we fed the output of our adder circuit into a switching circuit that switched at the same rate and phase as the 1000Hz rate of our modulated signal. We wanted to switch at the same rate and phase so that we could obtain our signal centered at 0V on the output as opposed to the -4.2V we would get if we switched our circuit when the -4.2V was occurring. We decided to choose a switching duty cycle of 25% so we would obtain half of our original signal on every cycle. The reason behind this was that, upon demodulation, we were getting bad data due to RC time constants earlier in our circuit in the region of the first half of our cycle. We were able to achieve this duty cycle using the oscillatory circuit we built earlier.

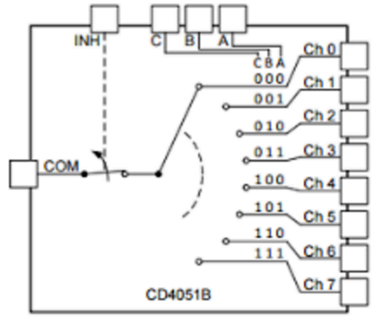
We then added a capacitor to the output pin with C = 10nF so that whenever the switch was on, the capacitor would assume the value of the input and whenever the switch was 'off' the capacitor would hold that value and so the cycle would continue. In essence, this capacitor was used to trace the path of our signal.

The schematic looks as follows:



The use of a multiplexor (CD4051BE) in place of an actual digital switch was because we only had a ±9V power supply, and +5V/0V logic levels, which the multiplexor could handle, but which the digital switches we had in class couldn't. Another major reason behind using the multiplexor was that even though there was no DC logic level pin to set what 'high' and 'low' were, the input pin A assumed 5V was high and could switch using this value. This whole exercise served as reminder that we always need to check the datasheets of the parts we choose to use, since they may not operate on the supplies we have access to. Speaking of datasheets the reason behind why we wired the pins the way we had can be better understood by looking at the following schematic from the datasheet of the CD4051BE.

Functional Diagrams of CD405xB



By feeding our input into the COM line and grounding INH so that the switch at INH was always connected we were able to regain our signal at Ch 0. We then controlled the switching by feeding our 1kHz square wave into A. The reason behind this is that A controls the first bit and therefore, if A was high, the angular switch would move to Ch 1 and if it were 0 it would move to Ch 0. Therefore, as long as our input signal to A and to COM were in phase, Ch 1's output would be our Heart Rate signal. In essence, as A went high, the switch would move to Ch1 and it would be at exactly the same time that the input from COM would be centered around 0V and would be the 20mV peak to peak of our original heart rate. As a reminder, we connected this output to a capacitor, which was grounded on the other end, which would ensure that we could 'store' the last value of Ch1 as the switch moved back to Ch0.

Once we had this working, our next goal as mentioned earlier was to filter the signal.

2nd Order Low Pass Filtering with Gain

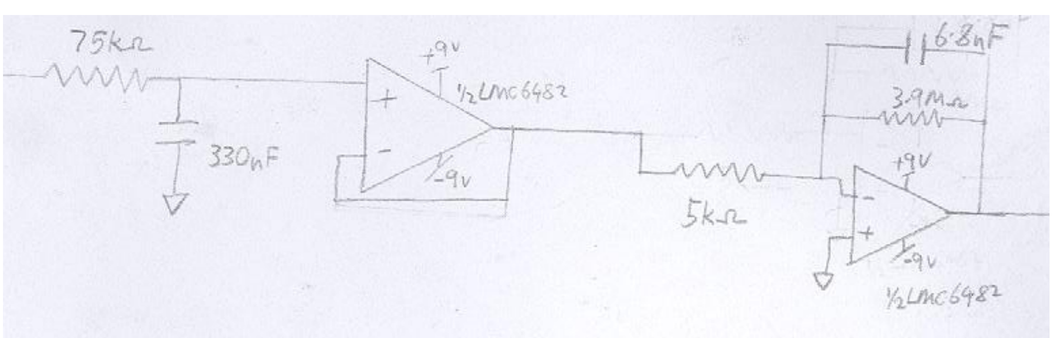
Initially, once we had our heart rate signal back, we wanted to filter out the noise. We decided that the maximum heart rate we would expect a little child to have to be around

190 BPM which translates to $\frac{190}{60} = 3.2Hz$. Therefore, we wanted to pass all frequencies below this value. This then set our fpass at 6.4Hz by the rule of 2.

Given $f_c = 6.4Hz = \frac{1}{2\pi RC}$ implies RC is approximately 0.025s. Given these values, we chose C = 330nF and R = 75kΩ which gave an fpass of 6.43Hz very close to what we originally wanted.

Our output signal was much better once we had filtered it, but there was still noticeable noise in the signal. Therefore, we decided to add a second stage of low pass filtering with equivalent critical frequency and gain as the first stage, so that we could feed a comparator without as much of a risk of experiencing as many multiple transitions due to noise.

Given that our signal was 20mV peak to peak at best, we decided to add a gain of 400 initially, but this was not sufficient as we realized that our circuit was attenuating the input signal slightly. Therefore, we decided to use a gain of 800, which would mean our signal would at best oscillate 16V peak-to-peak, which corresponds to an 8 V amplitude. The design for this took the form of the following:



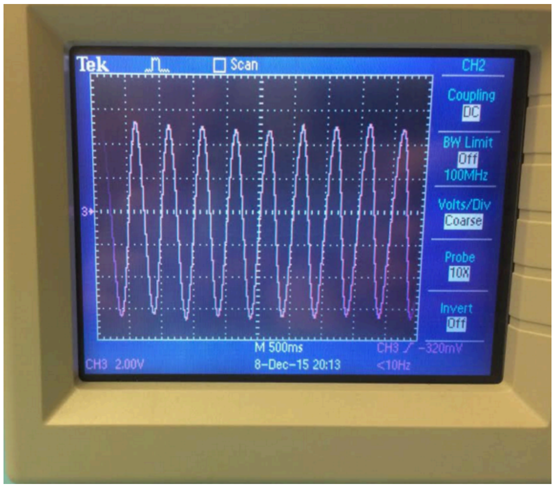
Note the presence of a buffer between the first filter and second is necessary because Thévenin issues would have occurred otherwise. As you can see, we use an inverting gain circuit. Since there wasn't a 4MΩ resistor, we chose the closest value to it. Given this, we were limited to a specific capacitor since we wanted our f_{pass} to be around 6 Hz as we calculated before.

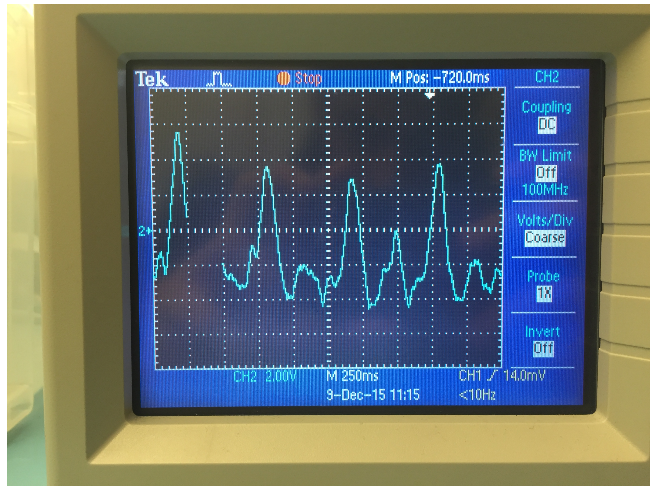
Using C = 6.8nF and R = 3.9MΩ, the fc we obtain was 6Hz, making our fpass at 3Hz. Remember that because we cascaded two filters, the 3Hz fpass now occurs at the -2dB point and the attenuation is 40dB/decade.

In testing our circuit, we were able to achieve very strong results for our heart rate signal, using our sine wave test signal as our input. While we didn't get an 8V amplitude as theoretically expected we were still able to reach 6V amplitudes using our test signal. The reason behind why we obtained some attenuation is due to filtering as our attenuation depended on the frequency of our test signal but it is also due to our adder circuit which used 1kΩ resistors with 5% tolerance leading to a gain that was not exactly 1. We accounted for this attenuation by increasing our gain, initially we had used a 10kΩ resistor with a 3.9MΩ resistor for a gain of approximately 400 but we changed our 10kΩ resistor to a 5kΩ resistor thus increasing our gain to approximately 800. A scope of our test signal and HR signal are illustrated below:

Test Signal

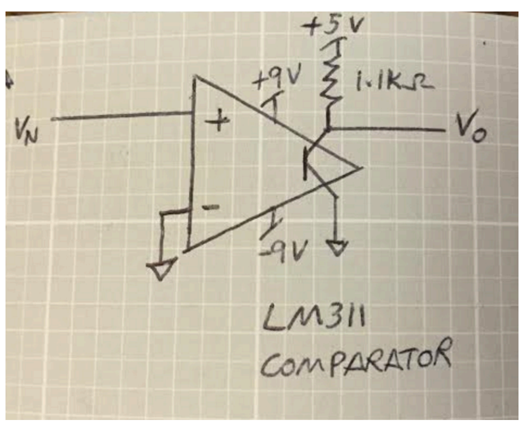
Actual HR Signal





Comparator with Hysteresis

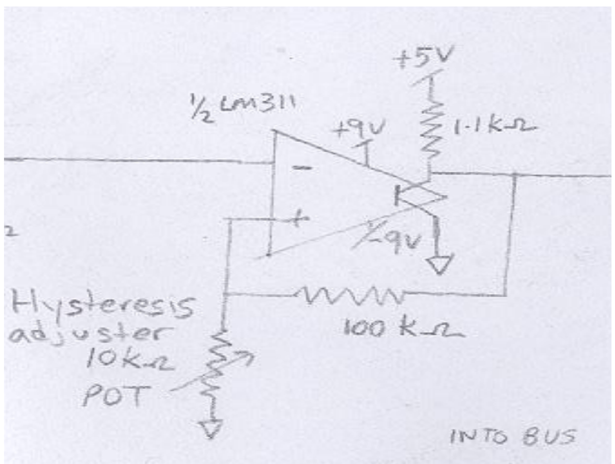
Once we had a working heart rate signal, it was time to convert this to a digital signal. We initially decided that a normal comparator without hysteresis would do and therefore used the following set up:



This comparator worked exactly as we expected it to (Note: the use of the 1.1k resistor was arbitrary). Later in our design process, however, while using Vo to interrupt an

Arduino interrupt pin we noticed that multiple transitions around 0V were taking place, meaning our input signal was causing Vo to oscillate and generate more interrupts than we would have expected given an actual heart rate. To account for these multiple transitions, we added hysteresis to the comparator.

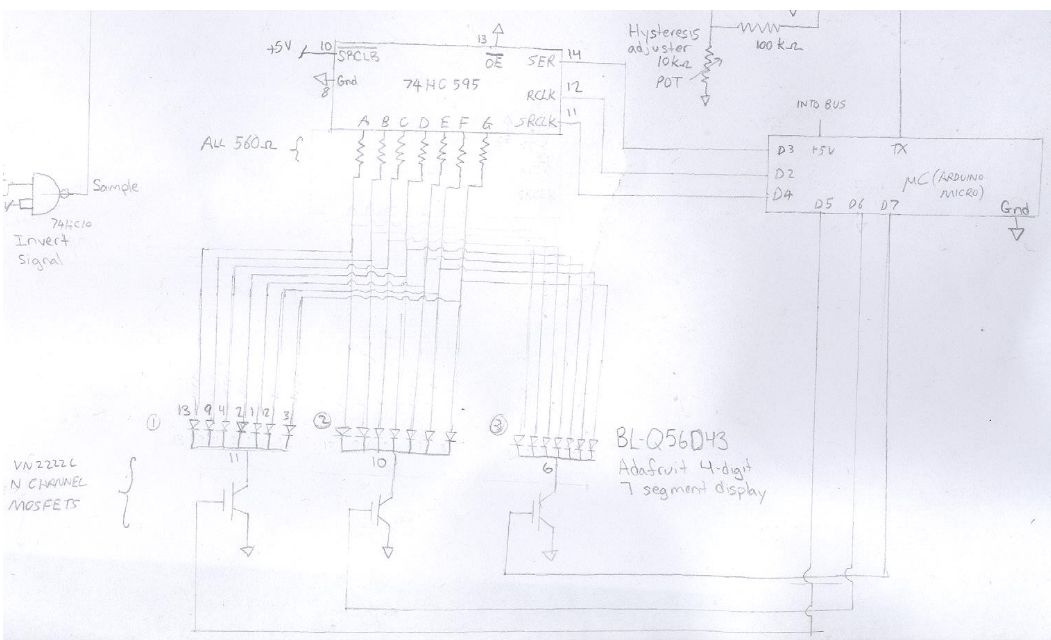
The following part of our circuit illustrates our comparator with hysteresis.



The 10k potentiometer is placed in series with the 100k resistor to adjust the hysteresis value. This value can range between 0V and 0.5V since the maximum voltage that we can divide is 5V between a 100kΩ resistor and a 10kΩ resistor. Our output should transition from 0V to 5V when the heart rate signal goes from negative to positive. Using this fact we can program our Arduino to listen for heartbeats (0V->5V transitions in the comparator output) on the arduino asynchronous interrupt line. From this we can calculate bpm on a microcontroller. Note we do not worry about Thevenin issues here since the output line is going into the microcontroller in which we assume the input pin has high input impedance.

Digital Display

To display the heart rate value that we calculated we utilized the Adafruit 4-digit 7-segment display. The display was the BL-Q56D43, so the LEDs within each digit shared a common cathode (not a common anode as the LEDs within the digital display we used in lab 6 shared). Therefore, we were unable to use the TLC5916 8-Channel Constant-Current LED Sink Driver chip to drive the LEDs, since the common cathode connection required us to source current to the LEDs, not sink current. Therefore, we used a 74HC595 8-bit serial-in-parallel-out shift register to source current to the appropriate display pins. The schematic of this portion of circuit corresponding to the digital display is shown below:



We wired 7 of the output pins of the shift register to 7 of the display pins, taking note of which display pins corresponded to which segments on the digits. Since our microcontroller, which we used to shift values into the 8-Bit register, functioned sequentially, and since we were only using one register as our driver we determined that we were able to source current to one digit in the display at a time. However, our heart rate value could be as long as 3 digits. To fix this issue, we decided to alternate through sourcing current to each of our digits very rapidly (at a frequency > 20Hz, so that a human could not perceive that the digits are flickering on and off). We accomplished this by attaching a VN2222 N-Channel small signal fast switching MOSFET to the common cathode pin corresponding to each of the first three digits of the display, and wiring the gate of each MOSFET to an Arduino pin, which was set high when we wanted to display the digit that the Arduino pin was wired to.

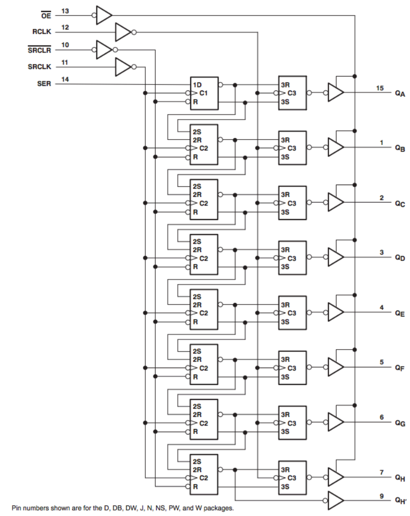
After calculating the beats per minute (bpm) heart rate value, our Arduino code shifted the first sequence of values that corresponded to the number that we wanted to display in the 100s place of the 7-segment display into the shift register (see Appendix B for code on how this was implemented), set the Arduino pin that was wired to the gate of the MOSFET connected to the common cathode of the appropriate digit high to source current to that digit, delayed briefly (~2ms), then set the gate of the MOSFET low to turn off that digit. It did the same for the digit in the 10s place and the 1s place, and then repeated this entire process each time a bpm value was calculated.

We also had to consider the current ratings of the 8-bit shift register and the 7-segment display. According to the datasheet for the '595, the shift register can safely source a maximum of 70mA. Thus, we determined we needed to place current limiting resistors at the output pins of the register, before connecting them to the display. Since at any one time we will be sourcing current to a maximum of 7 of the display pins (when displaying an 8 on one digit of the 7 segment display), we decided that we would need to limit the current to approximately 9mA per pin. Since the absolute maximum forward current for each segment in the display was 25mA, we did not risk providing too much current to the display LEDs. Also, when testing our display with 9mA flowing through each LED, the LEDs appeared to be bright enough, so 9mA was an acceptable choice. Since the source voltage of the ‘595 was 5V, we determined that the value of the current limiting resistors needed to be $\frac{5V}{9mA}$ = 560 Ohms (standard value).

The 74HC595 8-Bit Shift Register

The 74HC595 consists of a preliminary shift register into which we shift values that correspond to the digit we wish to display, and a storage register, which is connected to the output pins of the package. The storage register performs a similar function as the latch in the TLC5916, in that it doesn’t shift in the values from the preliminary shift register until it is signaled to do so. Below is an image showing the components that make up the shift register. The preliminary shift register is the column of flip flops on the left and the storage register is the column of flip flops on the right:

Logic Diagram (Positive Logic)



The 74HC595 has 5 input pins:

1.) SER: serial data pin
2.) SRCLK: clock signal that dictates when the value at SER is shifted into the preliminary shift register
3.) RCLK: clock signal that dictates when values from the preliminary shift register are shifted into the storage register (we set this signal high when SRCLK goes low, so that the values most recently shifted into the preliminary shift register are stored in the storage register making them readable at the output pins)
4.) SRCLR*: active low signal that clears the preliminary shift register
5.) OE*: active low signal that allows the values stored in the storage register to be readable at the output pins of the package

To shift a value into the shift register, we set SER to the value that we want to shift in (+/- 5V), change the value of SRCLK from low to high to shift the value into the preliminary shift register, then change the value of RCLK from low to high to shift the values stored in the preliminary register to the storage register, so that they are readable at the output pins of the package. SRCLR* is set high so that we do not incidentally clear the shift register, and OE* is grounded so that output is always enabled (we shift values into the shift register quickly enough such that we cannot see that each segment is illuminated one by one). See timing diagram for the 74HC595 below:

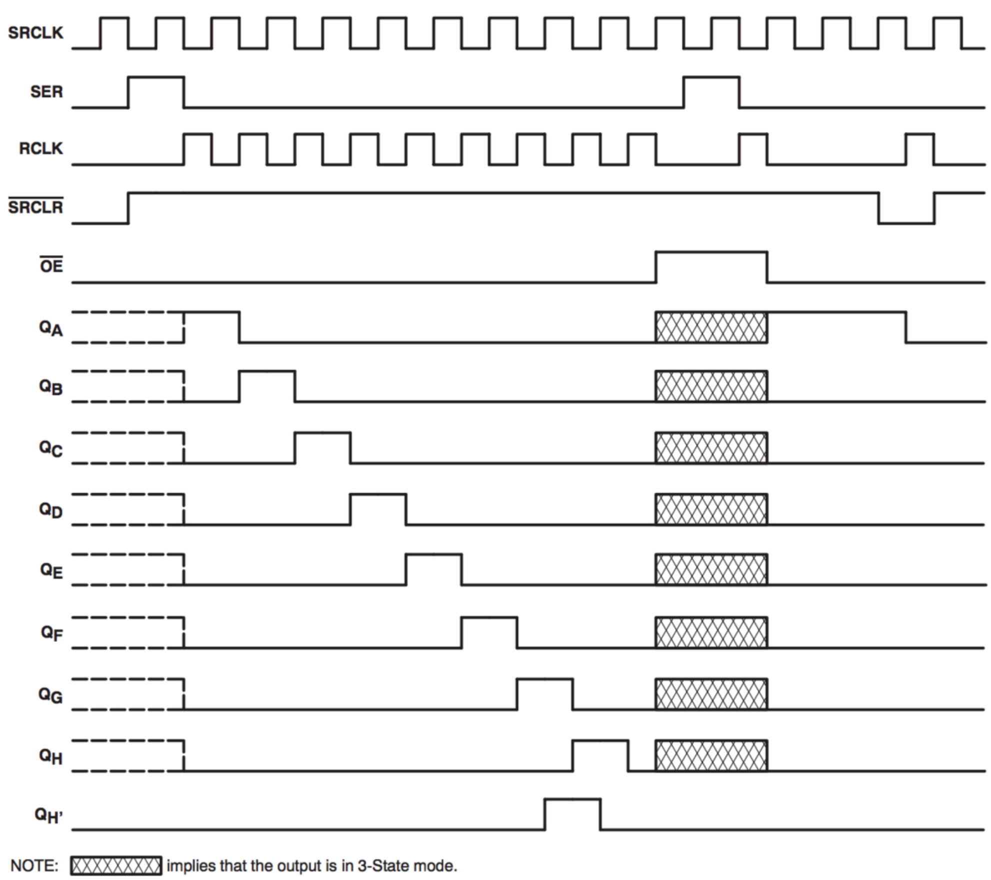


Figure 1. Timing Diagram

http://www.ti.com/lit/ds/symlink/sn74hc595.p

Conclusion

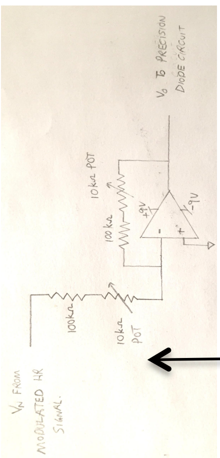
In the end we were able to sense a heart rate signal, modulate the signal to get rid of the DC offset, demodulate the signal, and then filter and compare it so that we could convert it into a digital signal that could interrupt a micronctroller pin. We used the microcontroller to calculate the average heart rate (in beats per minute) using a constantly updating queue. We then used a shift register and an Adafruit 4-digit 7-segment display to visually represent our average heart rate value. While our initial goal was to use serial data so that we could implement our design with Xbee's to transmit heart rate data wirelessly between the sensor and the display, working with modulation and analog circuitry took longer than expected, so we were unable to incorporate this wireless data transfer into our project.

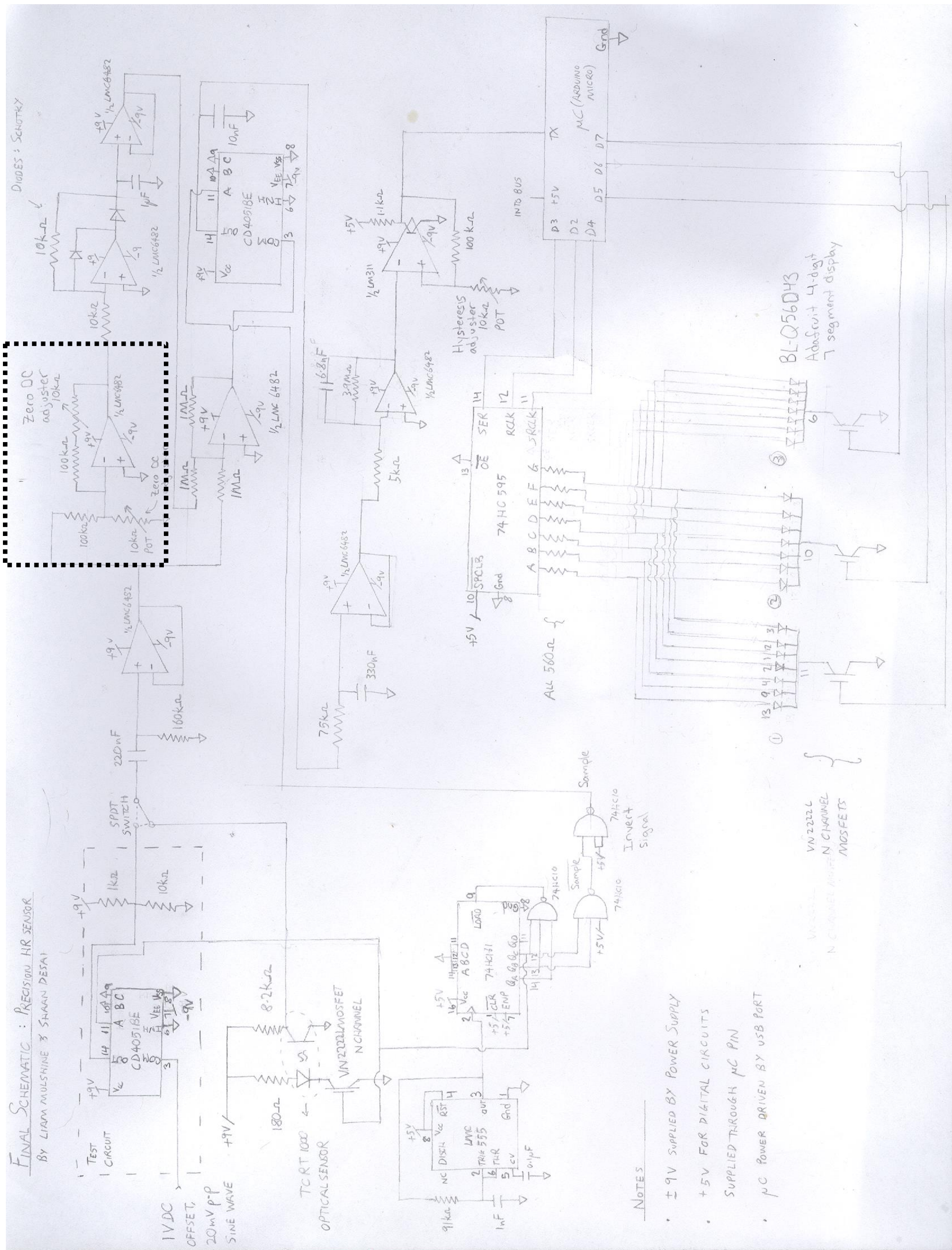
Had we had more time to work, we would have incorporated Xbees to allow data transmission between the sensing circuit and the display circuit to occur wirelessly, we would have tried to resolve the issues we noticed with the frequency dependent attenuation of our signal within our circuit, and we would have worked to fix the issue of multiple transitions on the Arduino interrupt line, which was causing some inaccurate spikes in the calculated heart rate value. Additionally, we would have explored an alternative approach to modulation, which makes use of a multiplier chip to modulate the heart rate signal, and then uses peak detection to demodulate it later on.

Overall, however, we are very pleased with how our project turned out, and how much we were able to learn about modulation, the design process, and analog and digital circuitry in general from it. We would like to thank David, Avi, Sandra and Jon for

helping us gain the skills necessary to realize our project and for providing us with both positive and negative feedback where it was needed along the way.

Final Schematic





Schematic Mistake: Replace what is in the dotted rectangle with circuit above

Appendix A:

Initial BPM Counter Algorithm which was later updated, see Appendix B

```
/****************************************

Liam Mulshine and Shaan Desai

Code to average heart rate and output this data

date: November 16, 2015

Update:

5 December: added num_beats = 0 in the loop

****************************************/


// Variables set by the user before compiling to control program behavior

const boolean debug = true;  // when set to true, debugging statements sent to the serial
port

// Program Constants

const int inputpin = 1;     // arduino pin with yellow LED (active high)

const int outputpin = 16;

int cur_avg = 0;

int num_beats = 0;

const int SIZE = 60;

int bins[SIZE];

int count = 0;

boolean flag = 0;


// Program Variables
```

```
void setup()
{

  pinMode(inputpin, INPUT);
  pinMode(outputpin, OUTPUT);
  attachInterrupt(digitalPinToInterrupt(inputpin),isr,FALLING);
  /* Set up serial port for debugging if enabled */
  if (debug)
  {
    Serial.begin(9600);              // This pipes serial print data to the serial monitor
    Serial.println("Initialization complete.");
  }
}

void loop()
{
  // first do anything we need to do before every state cycle
  if (debug)
  {
    Serial.print("Current State = ");
    Serial.println(cur_avg);
  }
```

```
  // create a time delay to measure number of beats per second
  delay(100);


  // fills an array by shifting each unit of data one unit left in the array
  for(int i = 0; i < SIZE-1; i++)
  {
    bins[i] = bins[i+1];
  }
  // adds the new value to the end of the bin array
  bins[SIZE-1] = count;


  // counts the total number of beats and averages over the last five beats measured per
second
  for(int j = 0; j < SIZE; j++)
  {
    num_beats = num_beats + bins[j];
  }


  cur_avg = 600*num_beats/SIZE;
  num_beats = 0;
  count = 0;
  flag = 0;
}


void isr() {
```

```
  if (flag == 0) {
    count = count + 1;
    flag = 1;
    //Serial.println("Got an ISR!");
  }
}
```

Appendix B:

Final BPM Counter Algorithm with Shift Register for Display

```
/*****************************************
Liam Mulshine and Shaan Desai
Code to calculate average heart rate (bpm)
Algorithm: Create array of size SIZE. Each element in the array contains a time
in milliseconds corresponding to the time that a heart beat was sensed via an
interrupt. A pointer, ptr, points to the oldest element of the array. When ptr
exceeds 9, set ptr back to 0 and overwrite old data. Each time
through the foreground loop, the bpm is calculated using the difference in time
between SIZE - 1 heartbeats.
bpm = 60000 x (SIZE - 1) / (newest - oldest)
date: December 6, 2015
*****************************************/
// Variables set by the user before compiling to control program behavior
const boolean debug = true;  // when set to true, debugging statements sent to the serial port
// Program Constants
const int inputpin = 1;     // arduino asynchronous interrupt pin
const int SIZE = 10;
const int RCLK = 2;
const int SER = 3;
const int SRCLK = 4;
const int D2 = 5;
const int D3 = 6;
```

```
const int D4 = 7;



// Program Variables
int ptr;
int tempPtr;
int tempPtr2;
unsigned long timeDiff;
int count;
int digits[3];
int reg[7];
int bpm; // changed this to int from float
unsigned long dt;
unsigned long oldest;
unsigned long newest;
unsigned long bins[SIZE]; // array that holds heartrate times
volatile unsigned long flagoff; // checks millis on every interrupt
// run setup once
void setup()
{

  pinMode(inputpin, INPUT);
  pinMode(RCLK,OUTPUT);
  pinMode(SER,OUTPUT);
  pinMode(SRCLK,OUTPUT);
```

```
pinMode(D2,OUTPUT);
pinMode(D3,OUTPUT);
pinMode(D4,OUTPUT);
attachInterrupt(digitalPinToInterrupt(inputpin),isr,RISING);
// initialize program variables
count = 0;
ptr = 0;
tempPtr = 0;
tempPtr2 = 0;
timeDiff = 0;
newest = millis();
oldest = millis();
// initialize digits to be filled with zeros
int digits[] = {0,0,0};
clearReg(); // initalize elements of register to hold 0s

// initialize bins to hold values with nonzero dt to
// avoid divide by zero error during first 10 seconds
for(long i = 0; i < SIZE; i++)
{
  bins[i] = i;
}

/* Set up serial port for debugging if enabled */
if (debug)
```

```
  {
    Serial.begin(9600);           // This pipes serial print data to the serial monitor
    Serial.println("Initialization complete.");
  }

}

/* Foreground loop
 *
 *  Sets a temporary ptr for each iteration which points to the oldest
 *  element of the array. Temporary pointer is set to avoid errors that could
 *  arise if an interrupt occurs and increments the pointer between fetching
 *  oldest and the newest element in the array.
 *
 */
void loop()
{

  // set temporary pointer
  tempPtr = ptr;

  // oldest value is at index ptr
  oldest = bins[tempPtr];

  // if ptr is 0 the newest value can be found at index SIZE - 1
```

```
if (tempPtr == 0)
  newest = bins[SIZE - 1];
else
  newest = bins[tempPtr - 1];

// difference in time between 10 heart beats
dt = newest - oldest;

// calculate bpm
bpm = 60000.0*(SIZE - 1)/dt;

// if the last interrupt occured 10 seconds ago, set HR so ERR appears on display
if(millis() - flagoff > 10000.0)
{
  bpm = 400; // to give error if no interrupt in 10 seconds
}
numToThreeDigits(bpm); // populate digits with the digits that make up bpm

clearReg();
digitsToReg(0);
writeRegister();
digitalWrite(D2,HIGH);
delay(2); // turns on an off faster than the human-eye can notice
digitalWrite(D2,LOW);
```

```
  clearReg();
  digitsToReg(1);
  writeRegister();
  digitalWrite(D3,HIGH);
  delay(2); // turns on an off faster than the human-eye can notice
  digitalWrite(D3,LOW);

  clearReg();
  digitsToReg(2);
  writeRegister();
  digitalWrite(D4,HIGH);
  delay(2); // turns on an off faster than the human-eye can notice
  digitalWrite(D4,LOW);



}

/*
 * Interrupt service routine
 *
 * When called, set the bin at index ptr to hold the value millis() return.
 * This value overwrites that value that was previously located at this index.
 * The function millis() returns the number of milliseconds that have passed
 * since the program started. Note that this program must be reset after
 * approximately 60 days since millis() will overflow at that point. Then
```

```
 * increment the ptr, and if ptr > 9, set ptr to 0 so that we do not point to
 * an index outside our array.
 *
 */
void isr() {

  flagoff = millis();
  if (ptr == 0)
  {
    tempPtr2 = 9;
  }
  else
  {
    tempPtr2 = ptr - 1;
  }
  timeDiff = millis() - bins[tempPtr2];
  if (timeDiff < 250)
  {
     return;
  }
  else
  {
    bins[ptr] = millis(); // set newest value
    ptr++;
    if (ptr > 9) // if ptr > 9, move ptr back to 0
```

```
    ptr = 0;
  }
}
/*
 * given an integer, separates it into its digits
 *
 *  e.g. given val = 145
 *
 *  145 / 100 % 10 = 1
 *  145 / 10 % 10 = 4
 *  145 / 1 % 10 = 5
 *
 *  digits = [1,4,5]
 *
 *  if val > 300 returns bad data
 *
 */
void clearReg()
{
  for(long i = 0; i < 7; i++)
  {
    reg[i] = i;
  }

}
```

```
/*
 * Breaks number up into its digits
 * Error values
 * 10: leading 0... signals to writeRegister to shift in all 0s to register
 * 11,12: humanly impossible heartrate... 11 signals writeRegister to shift
 *       in values to shift register that makes it output E... 12 signals
 *       writeRegister to shift in values to shift register that makes it
 *       output r... therefore display will output Err (error)
 *
 */
void numToThreeDigits(int val){

  // heart rate above value that is humanly possibly, assumes bad data and
  // populates digits with 0
  if (val >= 300 || val < 0)
  {
    digits[0] = 11;
    digits[1] = 12;
    digits[2] = 12;
  }
  else if (val < 100)
  {
     digits[0] = 10;
     digits[1] = val / 10 % 10;
```

```
    digits[2] = val / 1 % 10;
  }
  else
  {
    digits[0] = val / 100 % 10;
    digits[1] = val / 10 % 10;
    digits[2] = val / 1 % 10;
  }
}
/*
 * Given index of digits, writes 1s and 0s (corresponding to
 * high and low respectively) into the array, reg, which
 * indicates what value needs to be shifted into the shift register.
 * The following table represents the values that will be stored in registers
 * A through G for given digits from 0 to 9.
 *
 *    A  B  C  D  E  F  G
 * 0: 1  1  1  1  1  1  0
 * 1: 0  1  1  0  0  0  0
 * 2: 1  1  0  1  1  0  1
 * 3: 1  1  1  1  0  0  1
 * 4: 0  1  1  0  0  1  1
 * 5: 1  0  1  1  0  1  1
 * 6: 1  0  1  1  1  1  1
 * 7: 1  1  1  0  0  0  0
```

```
* 8: 1   1   1   1   1   1   1
* 9: 1   1   1   0   0   1   1
*
* e.g.
* if digits = [1,3,5];
*
* digitsToReg(0);
*
* will populate reg such that reg = [0,1,1,0,0,0,0]
*
* http://www.adafruit.com/datasheets/BL-Q56C-43.pdf
* this datasheet contains the information which tell us which
* segments correspond to which registers
*
*/

void digitsToReg(int place){

  switch (digits[place]){
    case 0:
      reg[6] = 1; // segment A
      reg[5] = 1; // segment B
      reg[4] = 1; // segment C
      reg[3] = 1; // segment D
      reg[2] = 1; // segment E
```

```
  reg[1] = 1; // segment F
  reg[0] = 0; // segment G
break;
case 1:
  reg[6] = 0; // segment A
  reg[5] = 1; // segment B
  reg[4] = 1; // segment C
  reg[3] = 0; // segment D
  reg[2] = 0; // segment E
  reg[1] = 0; // segment F
  reg[0] = 0; // segment G
break;
case 2:
  reg[6] = 1; // segment A
  reg[5] = 1; // segment B
  reg[4] = 0; // segment C
  reg[3] = 1; // segment D
  reg[2] = 1; // segment E
  reg[1] = 0; // segment F
  reg[0] = 1; // segment G
break;
case 3:
  reg[6] = 1; // segment A
  reg[5] = 1; // segment B
  reg[4] = 1; // segment C
```

```
  reg[3] = 1; // segment D
  reg[2] = 0; // segment E
  reg[1] = 0; // segment F
  reg[0] = 1; // segment G
break;
case 4:
  reg[6] = 0; // segment A
  reg[5] = 1; // segment B
  reg[4] = 1; // segment C
  reg[3] = 0; // segment D
  reg[2] = 0; // segment E
  reg[1] = 1; // segment F
  reg[0] = 1; // segment G
break;
case 5:
  reg[6] = 1; // segment A
  reg[5] = 0; // segment B
  reg[4] = 1; // segment C
  reg[3] = 1; // segment D
  reg[2] = 0; // segment E
  reg[1] = 1; // segment F
  reg[0] = 1; // segment G
break;
case 6:
  reg[6] = 1; // segment A
```

```
  reg[5] = 0; // segment B
  reg[4] = 1; // segment C
  reg[3] = 1; // segment D
  reg[2] = 1; // segment E
  reg[1] = 1; // segment F
  reg[0] = 1; // segment G
break;
case 7:
  reg[6] = 1; // segment A
  reg[5] = 1; // segment B
  reg[4] = 1; // segment C
  reg[3] = 0; // segment D
  reg[2] = 0; // segment E
  reg[1] = 0; // segment F
  reg[0] = 0; // segment G
break;
case 8:
  reg[6] = 1; // segment A
  reg[5] = 1; // segment B
  reg[4] = 1; // segment C
  reg[3] = 1; // segment D
  reg[2] = 1; // segment E
  reg[1] = 1; // segment F
  reg[0] = 1; // segment G
break;
```

```
case 9:
  reg[6] = 1; // segment A
  reg[5] = 1; // segment B
  reg[4] = 1; // segment C
  reg[3] = 0; // segment D
  reg[2] = 0; // segment E
  reg[1] = 1; // segment F
  reg[0] = 1; // segment G
break;
case 10:
  reg[6] = 0; // segment A
  reg[5] = 0; // segment B
  reg[4] = 0; // segment C
  reg[3] = 0; // segment D
  reg[2] = 0; // segment E
  reg[1] = 0; // segment F
  reg[0] = 0; // segment G
break;
case 11: // E
  reg[0] = 1; // segment A
  reg[1] = 1; // segment B
  reg[2] = 1; // segment C
  reg[3] = 1; // segment D
  reg[4] = 0; // segment E
  reg[5] = 0; // segment F
```

```
    reg[6] = 1; // segment G
   break;
   case 12: //r
    reg[0] = 0; // segment A
    reg[1] = 1; // segment B
    reg[2] = 1; // segment C
    reg[3] = 0; // segment D
    reg[4] = 0; // segment E
    reg[5] = 0; // segment F
    reg[6] = 1; // segment G
   break;
  }

}
/*
 * passes the data contained in the array, reg, serially to
 * the serial-to-parallel shift register.
 *
 */
void writeRegister(){
  for (int i = 0; i < 7;i++)
  {
   digitalWrite(SER,reg[i]); // write serial data pin
   digitalWrite(SRCLK,LOW);
   digitalWrite(SRCLK,HIGH); // rising edge of SRCLK
```

```
    digitalWrite(SRCLK,LOW);
    digitalWrite(RCLK,LOW);
    digitalWrite(RCLK,HIGH);  // rising edge of CLK shifts data to 2nd register in
                              // series so new value can be outputted
  }
}
```