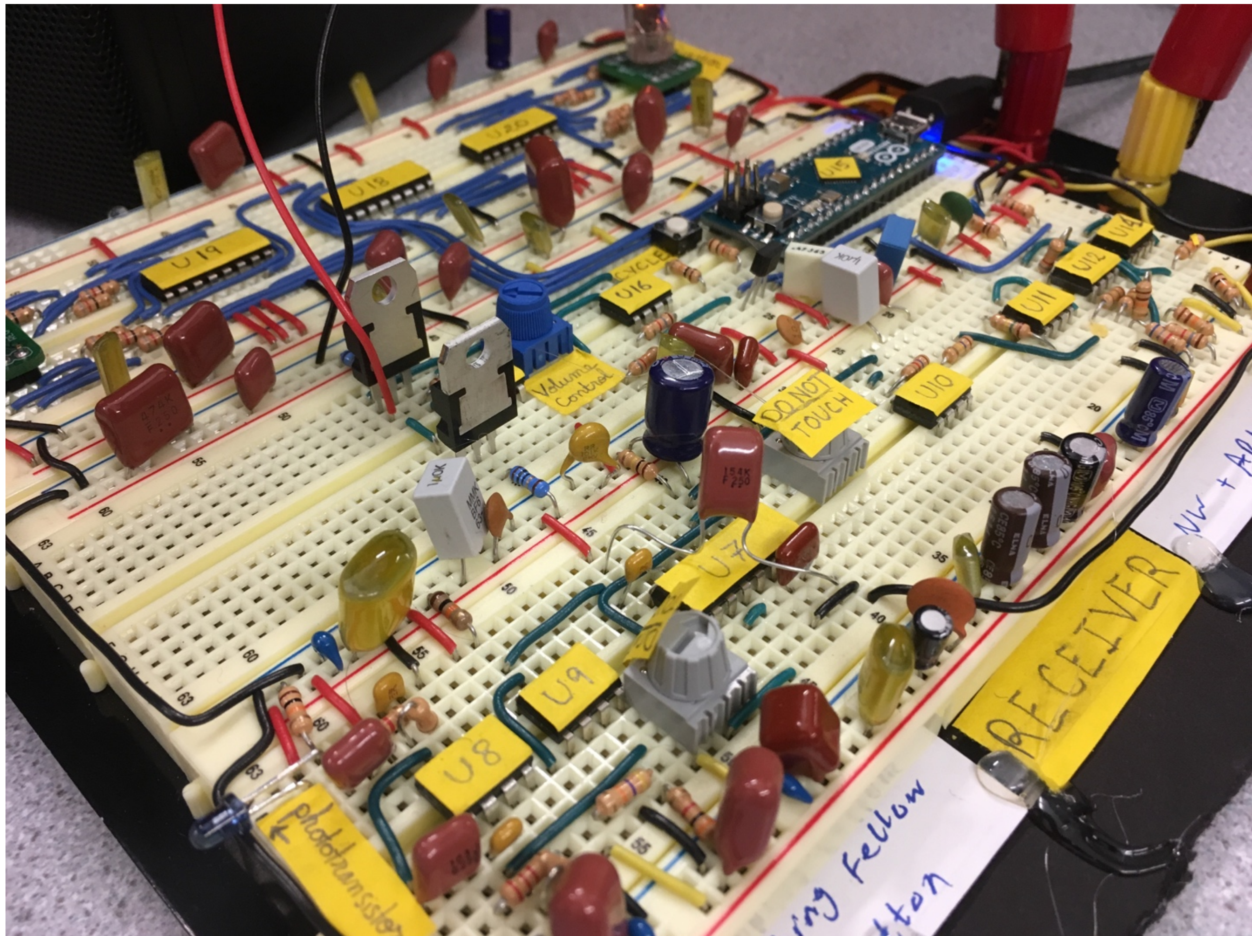
# ES 52 - Teaching Fellow Button

Andrea Rodriguez-Marin Freudmann, Nicolas Weninger

Spring 2017




## Abstract

The purpose of this project was to design and build a wireless communications system capable of sending data over frequency modulated Infrared light in order to create a method by which students in office hours can request help from a Teaching Fellow. The transmitter uses an Arduino and a LM555 astable circuit to encode and transmit the data. The receiver uses analog circuitry to condition the IR signal, which is passed through a phase locked loop to demodulate the signal. After further processing, this is read by a microcontroller, which provides visual and audio indication of a received signal. Although the final version met expected goals and minimum specifications, potential improvements are discussed.

# I. Introduction and Problem Statement

In ES 52 office hours, the late night and the pressure often can get to both Teaching Fellows (TFs) and students alike, especially for those in dire need of assistance in designing a challenging monostable or grappling with the woes of potentiometers in the earlier days of the class. To alleviate this, and to provide some nice electronic context to the Lowell D-Hall office hours, we designed and built a "Teaching Fellow Button". The TF button allows a student sitting at a table to tell the TFs which problem they need help on and they are located so that the TF knows which student it is. The TF unit would indicate when a transmission came in so that the TFs would know that someone needed their help. This could make office hours fairer, livelier and less chaotic, since TFs could ensure they were helping people evenly, and it would make more efficient use of time, as the TFs can see everyone's requests.

To achieve this, we decided to try our hand at wireless communication. After discovering that radio frequencies were too high for the equipment we had in lab, along with the associated challenges of HF design and communication on breadboards, we decided to use Infrared light at comparatively lower frequencies. This was further motivated by the availability of several IR components already in the lab. We knew that there were two predominant methods of wireless communication: amplitude modulation and frequency modulation. We decided to try FM, as this would be a novel topic that neither of us had explored before and seemed appropriately challenging.

In brainstorming ideas for the project, we decided that transmitting digital data rather than analog data made more sense for our application. We only needed to transmit small amounts of data and analog transmission would have undoubtedly complicated the design and the precision required to an unworkable extent to us with the given time.

## Responsibilities

Both of us worked on several aspects of project, but we were each assigned a part to take charge of. At large, this was the breakdown

| Nicolas | Andrea |
|---|---|
| Transmitter Circuitry | Phase-Locked Loop |
| IR reception and filtration | Filtered PLL signal processing |
| FSM Code | Address and Question displays |
| Speaker circuitry | |

*Figure 1: Breakdown of responsibilities*

# II. Systems

## Overview

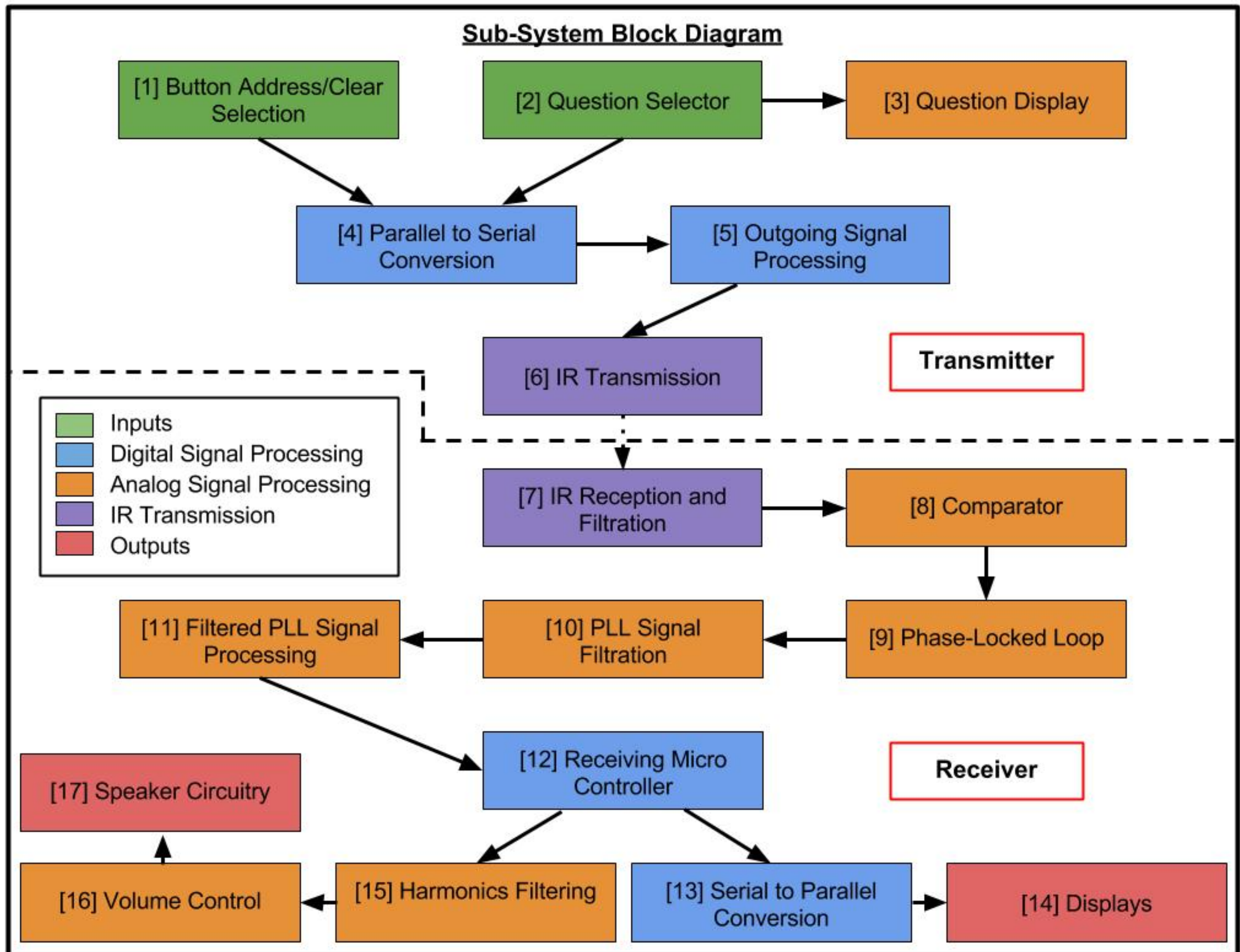



*Figure 2: Block Diagram*

The project consists of two units: the transmitter - or the student button - and the receiver - the TF unit. The transmitter allows the student to select a question that they are having trouble with by cycling through numbers one to nine on a button press. In order to simulate multiple transmission units for multiple students, three switches create an ‘address’ for the transmitter. When the student wishes to make a request, the request must be set to ‘active’ status by turning the ‘clear request’ switch off and upon pressing the ‘send’ button, a request is transmitted using a 555 astable circuit.

The receiver picks up the signal with a photo-transistor, whose emitter is connected to filtration and amplification circuitry that creates a clean square wave. This is fed into a phase-locked loop that gives a DC voltage dependant on frequency, which demodulates the frequency modulated signal. This signal is passed through further processing circuitry before received by another Arduino. This Outputs to two seven-segment displays, displaying the address from where the request came from and the associated question. A speaker also gives an audio indication.

# Power and Colour Scheme

## Transmitter Breadboard

We are using a single supply from 0v to 5v. The Arduino microcontroller PCB provides a regulated 5v single supply from USB power, which is either obtained from a computer or from a wall. As this breadboard is entirely digital, there was no need for a dual supply system and by sticking to single supply, we reached our expected goal for the transmitter.

## Receiver Breadboard

As the receiver breadboard combines both analog and digital circuitry, it was imperative that we had a dual supply scheme for the analog signals. We chose ±5v as this was a voltage that we are both familiar designing for and several of our components in lab are rated for approximately this voltage range. To avoid issues of connecting power supplies when we had the computer connected to the Arduino, we added a diode to isolate the two.

## Wire Colour Scheme

| Red | +5v |
|---|---|
| Black | GND |
| Yellow | -5v |
| Blue | Digital circuitry |
| Green | Analog circuitry |

# 1. Button Address and Clear Selection

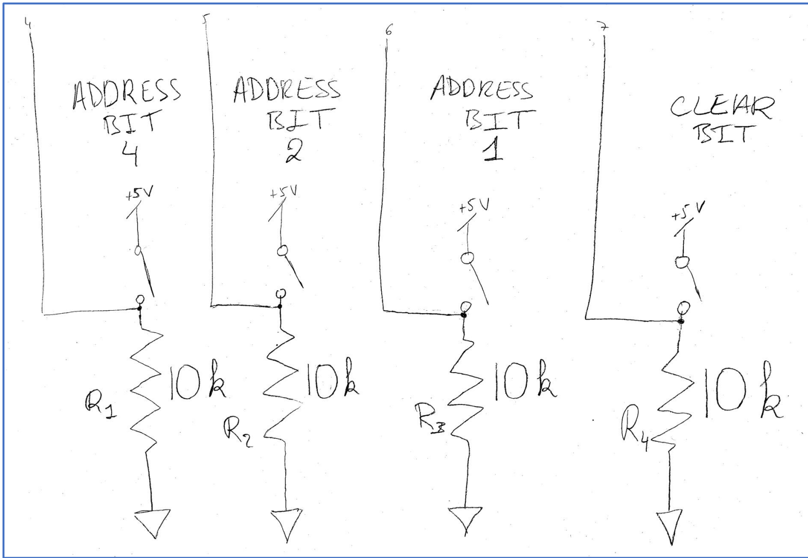



*Figure 3*

## Sub-Circuit Functionality and Explanation

In order to transmit address and other request data, we implemented a series of four SPST slide-switches that would create a three-bit binary address number and the 'clear' bit.

## Sub-Circuit Design Justification

These were configured as voltage dividers as we needed a logic LOW or HIGH for the parallel-to-serial converter the switches are connected to. Initially we intended them to be connected to the Arduino microcontroller directly, which would not have needed the external resistor by using the internal pull-up resistors, but after we decided to use the parallel-to-digital converter, they were required. The resistors are configured as pull-downs to make the signal active-HIGH, simplifying the conceptual design later on.

# 2. Question Selection

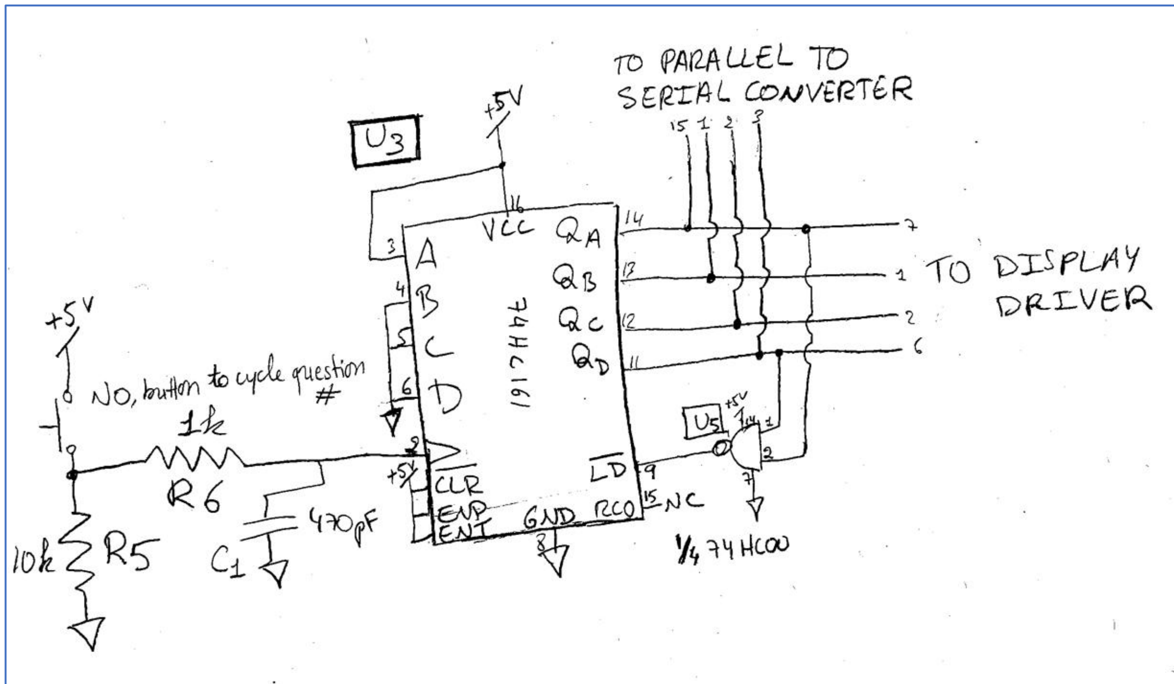



*Figure 4*

## Sub-Circuit Functionality and Explanation

This block uses the rising edge of the filtered push-button to clock a 4-bit counter. The outputs are connected to the 7-segment display (Numitron) driver to display the number. The NAND gate connected to Qa and Qd is connected to the active low synchronous Load (LD) input that resets the counter to its pre-loaded value of 1 when the counter has reached the count of 9. This allows the user to cycle between question numbers when making a request.

## Sub-Circuit Design Justification

When designing this block, we wanted a way for the user to be able to specify a question that they were having trouble with. To add variety to the design, we decided to use a 74HC161 counter (asynchronous as we are not using the clear functionality) and a pushbutton that would allow the user to cycle through questions 1 - 9 on the display. To achieve the resetting, the LD input is asserted when the counter outputs 1001 by the NAND gate pulling LD low. On the next button press, the rising clock edge caused the 74HC161 to reset to the loaded count of 0001. When testing, it became clear that the button needed to be debounced to avoid inaccurate button presses. We designed for fc = 100Hz by using R6 = 160Ω and C1 = 10uF but after further experimentation the values of 1kΩ and 470pF were chosen. Upon reflection, it appears that there may be Thevenin loading issues with this circuit, but it appears to work well nevertheless.

# 3. Question Display

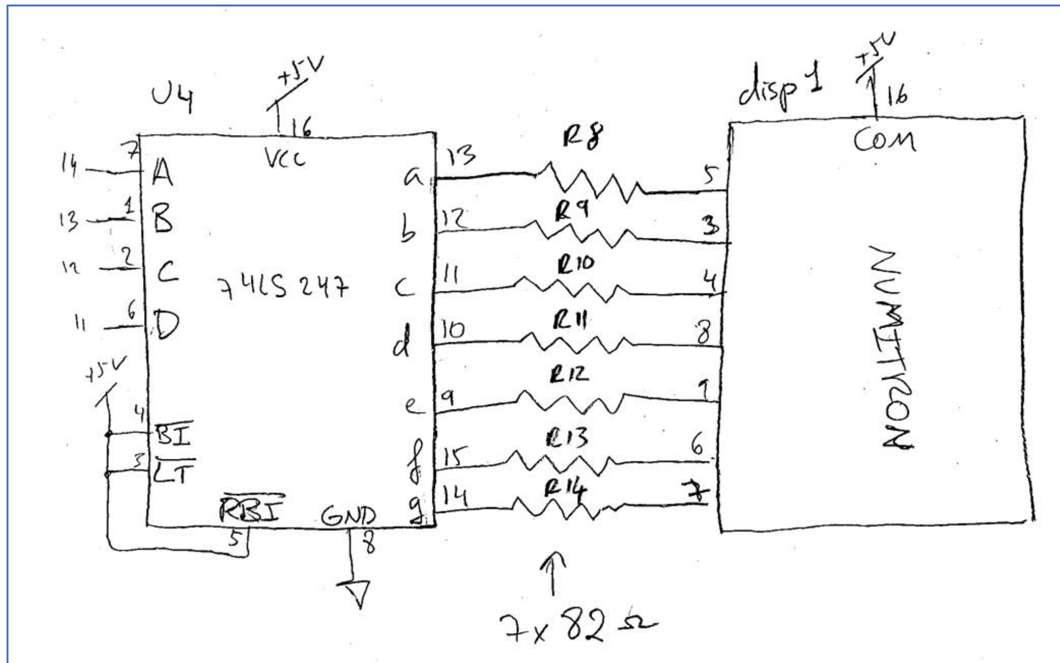



*Figure 5*

## Sub-Circuit Functionality and Explanation

The binary output of the 74HC161 counter is connected to the input of the 7-segment decoder and driver U4, which is configured to display the binary value on the input on the Numitron display.

## Sub-Circuit Design Justification

We designed this circuit with the lab notes to help us. Through the testing procedure described in the lab notes, we chose the current limiting resistor values of 82Ω for R8 - R14. In the lab 5 we characterized the numitrons to find their VI curve and then, based on those measurements, calculated that to get 20mA of current we the resistors to be 82 ohms (since the numitron can sink a maximum of 25mA it was a safe choice).

# 4. Parallel to Serial Conversion

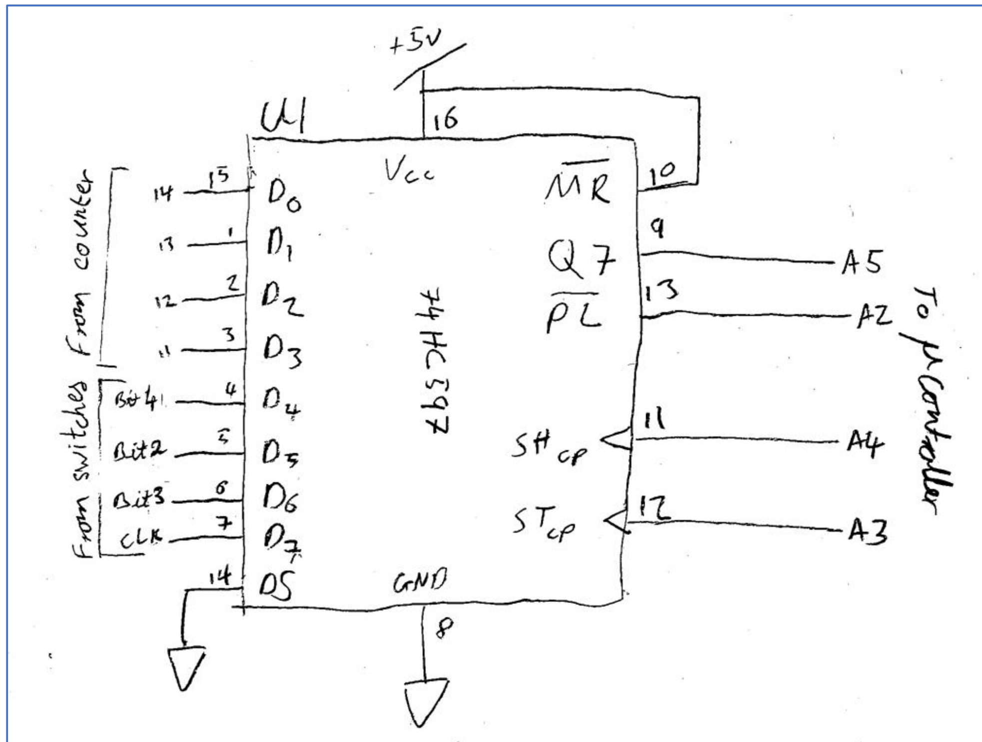



*Figure 6*

## Sub-Circuit Functionality and Explanation

The parallel-to-serial shift-register U1 receives inputs from the three address bit switches, the clear bit switch and the outputs of the question selection counter U3 and converts this to a serial output that, using the shiftIn() command on the Arduino, is read into the microcontroller to be transmitted via IR.

## Sub-Circuit Design Justification

We added this block to increase the complexity of the digital circuitry and to reduce the number of pins that we used on the Arduino. The 74HC597 is double-buffered, as opposed to the 74HC165, which does not have the storage buffer and thus does not require the additional clock pulse on STcp[1]. Thus, our design requires an extra pin on the Arduino, while the double buffering does not help hugely in its current implementation as we are not designing for fast changing inputs. The reason we designed for the '597 is because at the time of construction, the '165 was not available to us and we were under time pressure to complete the transmitter circuit. PL is asserted to clear current data and load new data. De-asserting PL allows the Arduino to read in the serial data from Q7 on each rising edge on SHcp. As the '597 reads-out the 0th bit before the first rising edge on SHcp, the FSM code on the Arduino compensates for this by reading Q7, performing a bit-shift, read in the last 7 bits then performs another bit shift operation. Please refer to the readData() function comments.

# 5. Outgoing Signal Processing

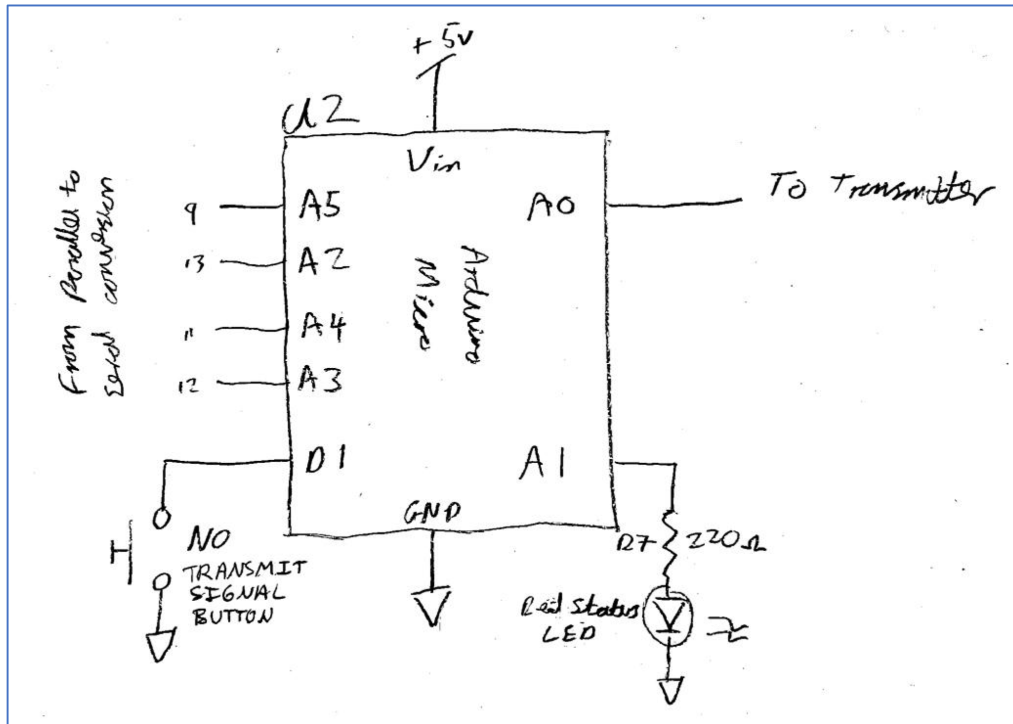



*Figure 7*

## Sub-Circuit Functionality and Explanation

The microcontroller is running an FSM loop that checks the status of the clear bit, read from the parallel-to-serial shift register with the address bits and question selection nibble (half-byte). Based on this, it pulses a high or low signal on A0 to control the transmission circuit to generate the FM signal. The user makes a request by pressing the push button and if the request is to ask for assistance, the status LED will illuminate. When the request is to clear a previous request, the LED will extinguish. The user cannot make multiple active requests.

## Sub-Circuit Design Justification

The user interface was designed so that we did not have to handle multiple requests at once from one address, which would have complicated the receiving circuitry and code. We used an Arduino in this situation rather than digital hardware to add digital complexity to the project in the form of an FSM and to speed up construction, as we anticipated (correctly) that the receiving analog circuitry would take up the majority of our time. The push button is configured as an interrupt on D1 with its internal pull-up resistor activated to simplify external circuitry. The interrupt allows the FSM to run at a very slow frequency 4Hz as we do not need to poll it frequently. This makes the code easier for the transmission states, as there is no need for several delay states and counters, and allows us to easily set the data rate as the frequency of the FSM.

# 6. IR Transmission

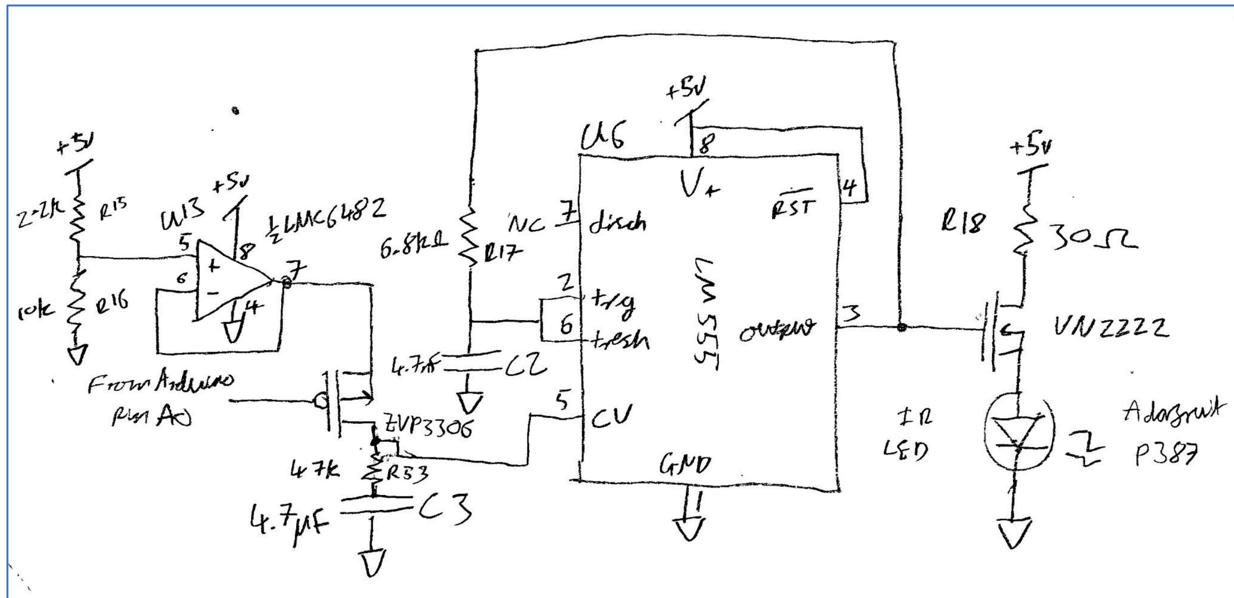



*Figure 8*

## Sub-Circuit Functionality and Explanation

The LM555 IC is configured as a 50% duty cycle astable oscillator with a p-channel MOSFET connected to the Arduino, able to switch in a buffered voltage to the CV pin that changes the frequency of oscillation to create the frequency-modulated signal. The output is connected to an ultra-bright 940nm IR LED.

## Sub-Circuit Design Justification

The resistor for the IR LED was chosen by finding the forward voltage and current for the LED at Vf = 1.6v and I = 100mA. The VN2222 has a resistance of 7.5Ω. So, R = [(5v – 1.6v ) / 100mA] – 7.5Ω = 27Ω. 30Ω is used due to resistor tolerance. The output of the 555 is not able to source 100mA as discovered in testing (despite what the datasheet says), as the waveform went flat when the LED was connected directly. Adding the n-channel MOSFET resolved this issue. We chose the 50% duty cycle topology to generate a square wave that would work best with the PLL [5].

Initially, we used a n-channel MOSFET to switch in a capacitor in parallel with C2 in order to change the oscillation frequency. While this worked well, the PLL would unlock from the incoming signal on the receiver and not provide a valid output. It was discovered in testing that the PLL did not unlock when the frequency was changed slowly rather than instantaneously. The final design thus uses an RC circuit to slowly change the voltage at CV to change the frequency, where the time constant = 0.2s. This constant was determined experimentally by observing how the PLL behaved. Unfortunately, this time constant limits our transmission data rate to 4Hz. When the p-channel MOSFET gate is at 5v, CV acts as though it is unconnected and the 555 oscillates at 22kHz. When the MOSFET gate is at 0v, CV is held at 4.10v and the

555 oscillates at 18kHz. This voltage was determined experimentally using the power supply directly at CV and measuring the voltage with a scope. After creating this voltage with the potential divider with R15 and R16, the 555 was not oscillating at the frequency expected. Suspecting that we were facing thevenin loading issues, we placed a buffer U13 after the voltage divider and before the MOSFET. This resolved the issue.

# 7. IR Reception and Filtration

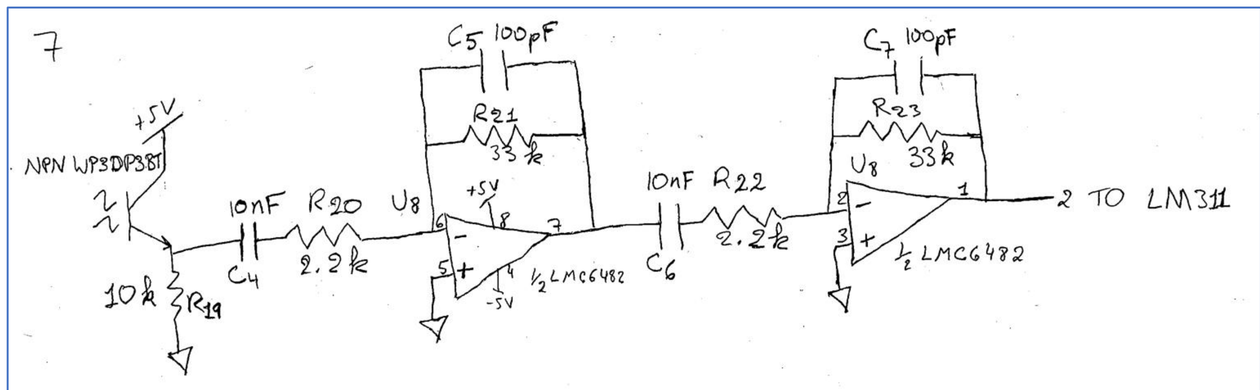



*Figure 9*

## Sub-Circuit Functionality and Explanation

The NPN W3DP38T phototransistor receives the IR signal emitted by the IR LED and generates a voltage that pulses at the same frequency as the one emitted by the LED. This voltage is then filtered using a band pass filter (a high pass followed by a low pass) to isolate the range that included 18 kHz - 22 kHz. The circuit also amplifies the signal by a gain of approximately 225, clipping at ±5v to generate a nice signal (fig.9).

## Sub-Circuit Design Justification

The phototransistor was chosen because it was designed to capture the same wavelength that the IR LED emits (940nm). We would get an optimal range and a reliable connection between the emitter and the receiver. The choice of 10nF and 2.2k for the high-pass gives us an Fcritical of around 7 kHz, and an Fpass of about 14 kHz. The choice of those two values for the resistor and capacitor in particular was based on our desire to keep our resistor values between 1k and 1M, the availability of components, and their RC which allowed us the passing frequency that we wanted. The low pass used 100pF and 33k to give us an Fcritical of about 48 kHz and an Fpass of about 24 kHz. The choice of those two values for the components was based on the same range of resistor values as for the high pass, the availability of components and their RC. The use of op amps allowed us to repeat the same values twice (in order to filter twice) without thevenin issues, at the same time as amplifying the circuit with a gain of R21/R20 (and R23/R22), so of about 15 for each filter. This circuit filtered around the range of 14 kHz and 24 kHz, which was the band of frequencies that we wanted (since it is near to the 18-22 kHz range).

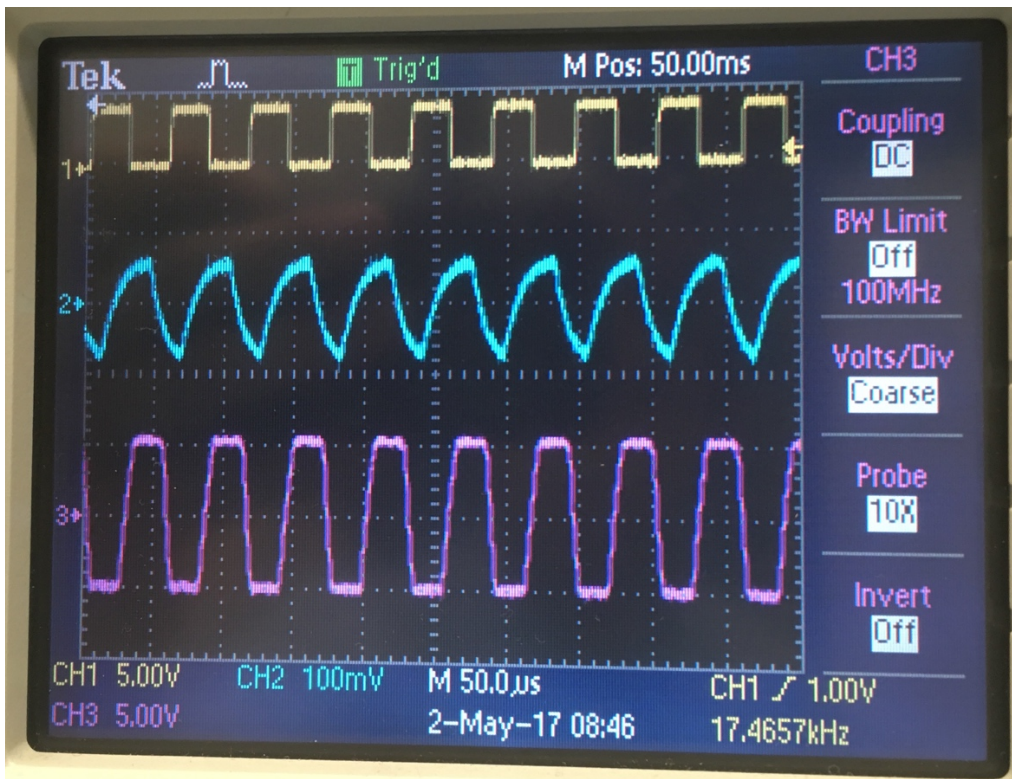



*Figure 10: Yellow: processed signal. Blue: first amplification. Purple: second amplification.*

# 8. Comparator

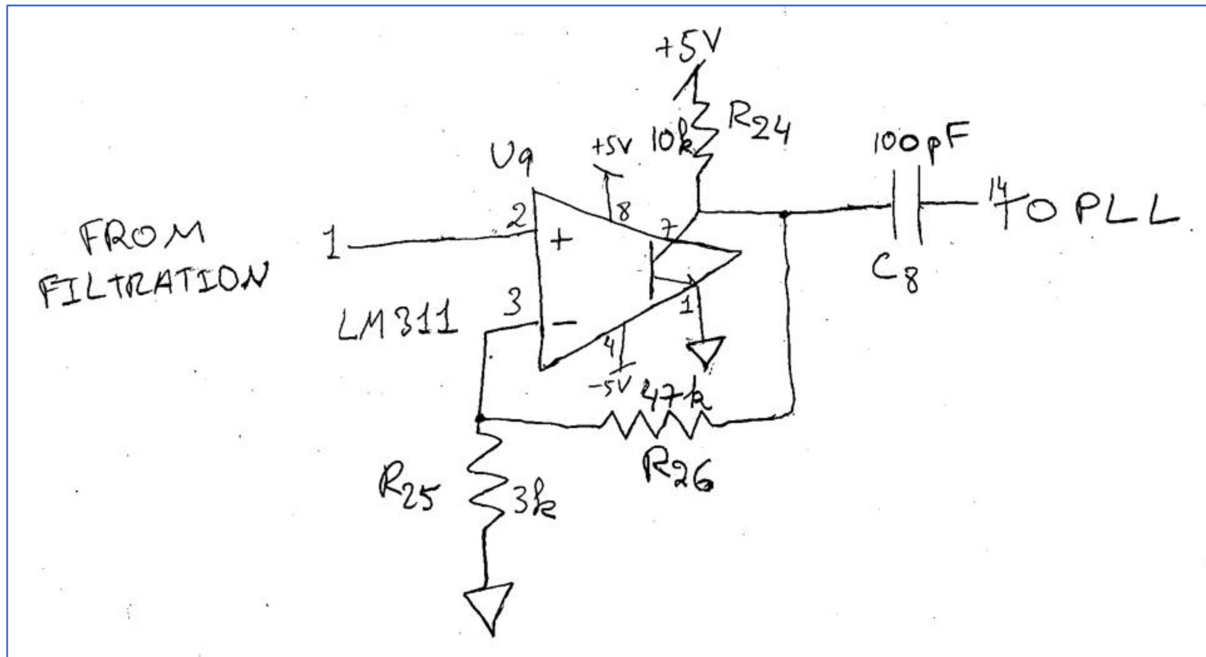



*Figure 11*

## Sub-Circuit Functionality and Explanation

The sine wave output that comes from the band pass filter is connected to the positive terminal of the LM311 comparator which allows us to generate a square wave. The comparator has 0.3V of hysteresis to ensure that the square wave is not overly affected by noise. The square wave then goes through a 100 pF capacitor to remove any DC offsets.

## Sub-Circuit Design Justification

We chose to use the LM311 comparator because it is faster than an op amp, we weren't using a large number of comparators (so we didn't need the two op amps in the package), and it allowed us to use hysteresis. We wanted the sine wave going into the positive terminal to come out of the LM311 as a "high" if above 0V and a "low" if below 0V (to get a square wave), but in order to make sure that the transitions were clean around the 0V cut-off (since noise could cause jumping) we added in 0.3 V of hysteresis. We thought 0.3V was a good value since it is very near 0V but also large enough that it cuts out the noise around ground, and because it was more than the noise level that we were observing on that line. We used 47k and 3k resistors because they are in the 1k-1M range and because 3/(47+3) = 3/50 = 0.06, and 0.06*5 = 0.3 V. We chose to use 10k for R24 since the value had to be less than the input impedance on pin 14 (PLL) divided by 10 to avoid thevenin loading issues, and the datasheet for the PLL listed its input resistance as 150kΩ. The 100 pF capacitor was chosen as the '4046 application notes suggests adding this small capacitor. Upon reflection, this capacitor is likely only recommended to remove any DC offset that the FM signal may have had. As our design does not have a DC offset, it is likely that we could have removed this capacitor.

# 9. Phase Locked Loop

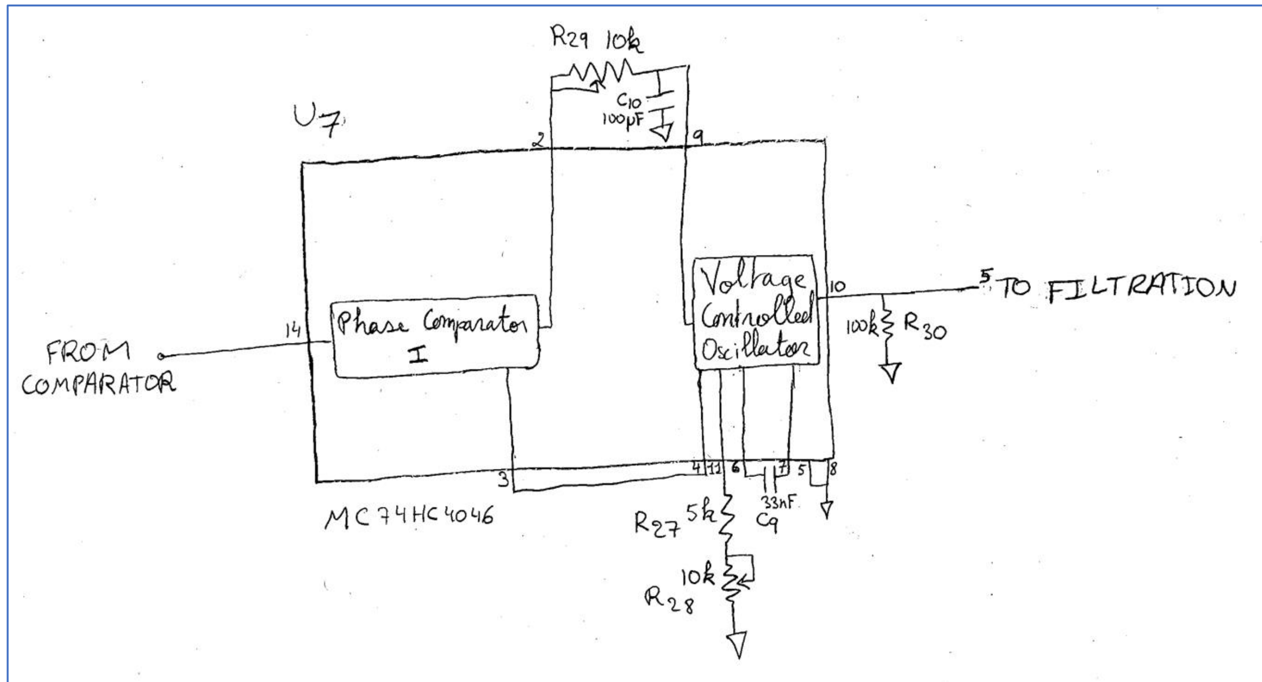



*Figure 12*

## Sub-Circuit Functionality and Explanation

The phase-locked loop (PLL) takes the square wave from the comparator as an input. The phase comparator compares the phase to the one generated by the voltage controlled oscillator (VCO), and then generates a signal which is filtered using a low pass becoming the “error signal”. The difference in frequencies when we change between 22 kHz and 18 kHz makes a different error voltage, which we can then compare so that we have a “high” and a “low” signal depending on the incoming frequencies.

## Sub-Circuit Design Justification

Using the application notes [3] we determined the rough outline for how to use the chip based on a sample FM modulation design that was provided. The notes also said how to calculate the values of the various resistors and capacitors. R29 and C10 had to allow us to stay within the frequency capture range (and also determined what that range was). By using the equation given and choosing the capture range to be 4 kHz (centred around 20 kHz) we calculated an RC of 0.789s and we chose C10 to be 100uF and R29 to be about 8k; however, we used a 10k potentiometer in order to modify the resistance while we were testing the phase locked loop with the oscilloscope, since a very small change resulted in a large difference in locking and so the 5% tolerances were not precise enough.  This was why we decided on 100uF and 8k: it made it possible to use the 10k potentiometer which was a fairly precise way of modifying the PLL so that it would work. According to the graphs on the application notes for a centre frequency of 20 kHz we needed a capacitor of 33nF (C9) and a resistor value of 10k; however, again for the

sake of flexibility with our circuit we decided to make the resistor into one 5k (R27) and one 10k potentiometer (R28) so that it could be easily modified. The datasheet showed that the output needed a pull-down resistor, and it used 100k in its example, so we decided to do the same.

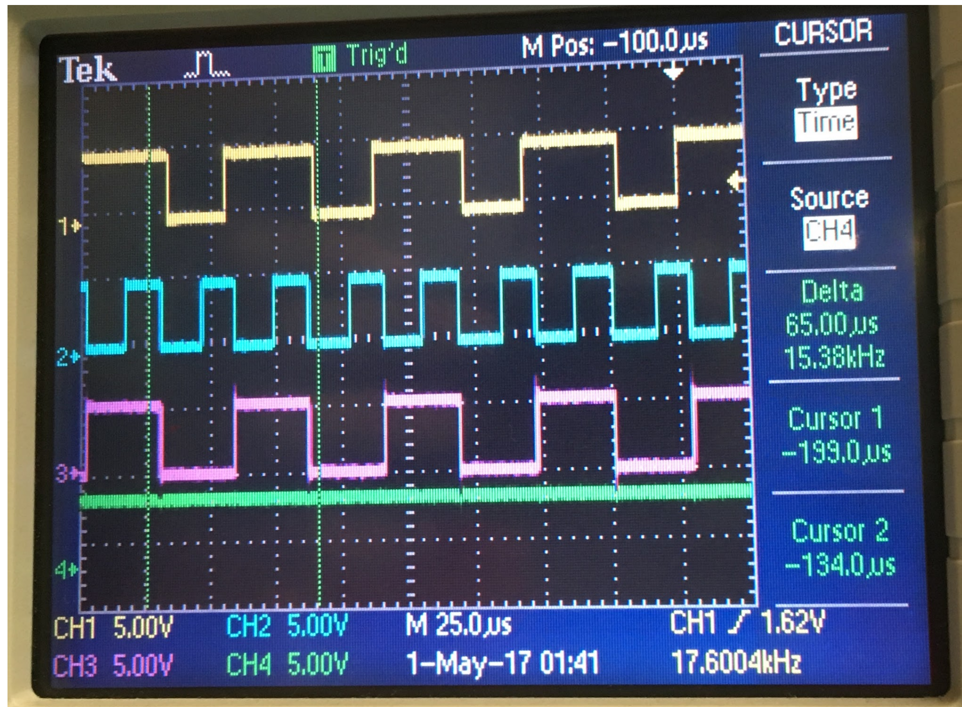



*Figure 13: PLL once it locked: the yellow is the input signal, the blue is the output of the phase comparator, the pink is the output of the VCO and the green is the DC output of the PLL.*

# 10. PLL Signal Filtration

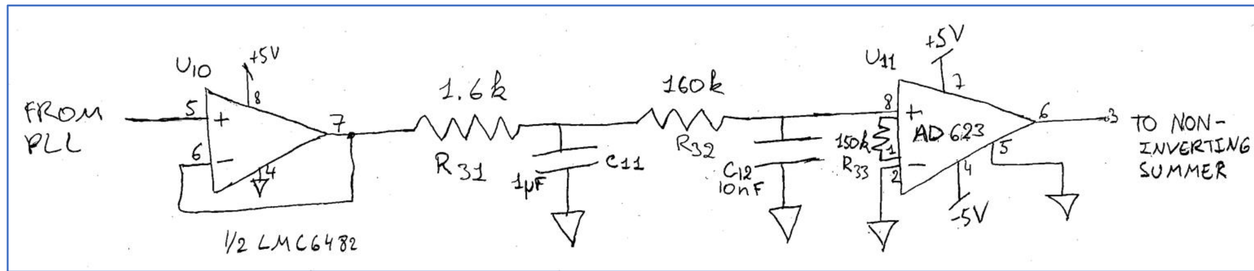



*Figure 14*

## Sub-Circuit Functionality and Explanation

The output of the PLL is buffered and then passed through two low pass filters and then a difference amplifier (AD623). The low pass filters are cascaded and serve to filter out some of the noise that affects the PLL's output voltage, and the difference amplifier does the same. The reason for this circuit was to attempt to get a cleaner voltage output from the PLL, because we had a lot of noise in our DC signal which was causing us problems.

## Sub-Circuit Design Justification

We decided to buffer the output in order to minimize the amount of noise in our signal. However, that did not work very well so we added the low pass filters to remove the harmonics in our supposedly DC signal, so we made Fpass 50 Hz and Fcritical 100 Hz. We chose 1.6k and 1uF at for the first low pass because the components were in the right range (the resistor is between 1k and 1M and the capacitor is not polarised which makes things easier), their RC is what we wanted and also because it is possible to cascade them with another low-pass fairly easily (the resistor in the next one will still be in the right range while respecting thevenin rules). The second filter has the same Fpass and Fcritical but due to thevenin rules we had to make the resistor bigger and the capacitor smaller (though we end up with the same RC). The difference amplifier is configured so that the difference between the filtered signal and ground is amplified. Using the formula $Rg = 100\ k\Omega/(G - 1)$ from the AD623 datasheet, R33 is 150kΩ as this gave us a difference amplification of 5/3, which we experimentally determined to reduce noise enough to do signal processing on the output without exceeding the 0v - 5v range the subsequent signal processing performs [4].

# 11. Filtered PLL Signal Processing

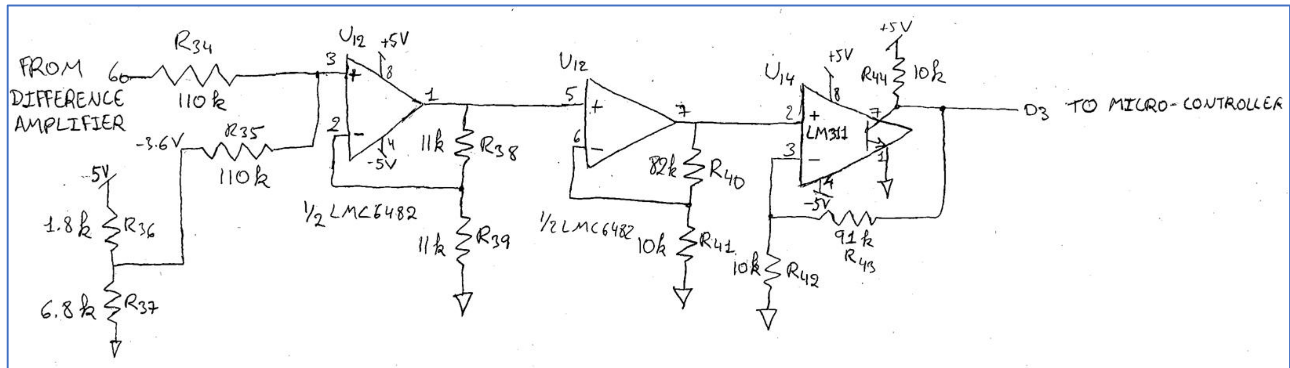



Figure 15

## Sub-Circuit Functionality and Explanation

The non-inverting summer subtracts 3.6 V from the output of the difference amplifier, and the new signal is then multiplied by 4. Finally, it passes through a comparator which gives us a high or a low signal depending on its input (and so we have a high or a low depending on the different frequencies that are sent by the emitter).

## Sub-Circuit Design Justification

We used an inverting summer to subtract 3.6 V from the difference amplifier signal. The reason behind this was that the difference between the high frequency voltage and the low frequency voltage was not large enough to constitute a HIGH or a LOW and still had errors, but in order to multiply it and get a bigger difference between the voltages we needed to make them smaller (otherwise they would go past the bounds of Vcc and Vee). 1.8kΩ and 6.8kΩ were not originally chosen because the voltage divider equation would suggest that it should be 6.8kΩ and 2.7kΩ to get about -3.6V. However, our ground was so noisy that none of the calculations were completely right, and so by checking on the oscilloscope and experimentation we managed to get to -3.6V by lowering the 2.7kΩ resistor value to 1.8kΩ. For the inverting summer to give us just a subtraction we needed R38 and R39 to be the same, so we chose 11kΩ on the basis that it was not a particularly commonly used value, though it really could have been anything in the 1kΩ to 1MΩ range (the same reasoning went into R34 and R35, since they just need to be the same value). The non-inverting amplifier is supposed to multiply the voltages by 4 (to get the largest difference between them without going outside the bounds of our components) and originally, we used a 27kΩ and an 82kΩ resistor; but again, due to noise we changed the 27kΩ to a 10kΩ and verified using the oscilloscope that we were indeed multiplying by four. The LM311 is actually not using resistor values that are mathematically correct because the reference should ideally be between 1V and 3.2V, whereas we currently have hysteresis at 0.5V; but due to the noise on ground and the continuing error in the signals the comparator works to give a high of 5V when the incoming frequency is 18kHz and a low of 0V when the incoming frequency is 22kHz. 0.5V of hysteresis made our signal high for half a duty cycle rather than a whole one, but we later dealt with this problem in software.

# 12. Receiving Micro-Controller

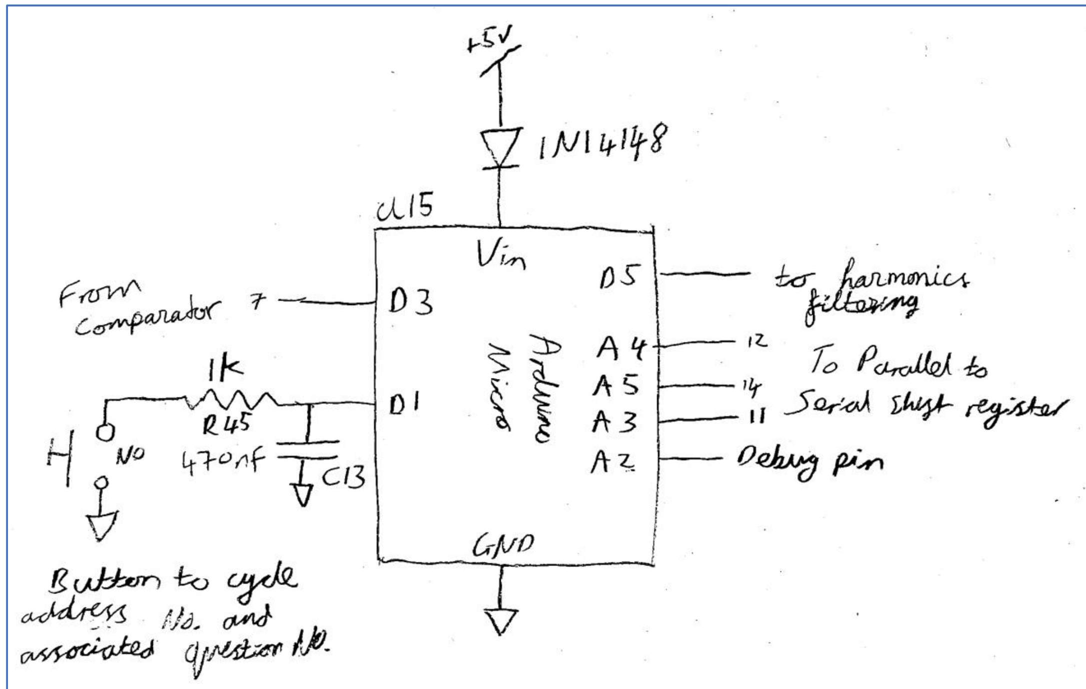



Figure 16

## Sub-Circuit Functionality and Explanation

The microcontroller on the receiver receives a digital signal from the comparator. The software polls this line for a rising edge and when a rising edge is detected, it undergoes a signal acquisition routine that, when completes, checks that the received signal is indeed valid by checking the bits. If a valid signal is received, it outputs a square wave on D5 to the speaker circuitry and shifts out the address and question number of the most recent request to the serial to parallel shift register. The push button is connected to D1 with the internal pullup activated and with a low-pass filter to debounce the button. An interrupt triggered by the button allows TFs to cycle through button addresses and see the associated question number.

## Sub-Circuit Design Justification

Given time constraints we did not have time to design digital hardware to read the digital IR signal. The Arduino additionally allows us to perform more involved signal verification steps to prevent false positives if the PLL became unlocked. We used the same low-pass filter design as for the transmitter Arduino as this was experimentally determined as a good solution. The diode on the power supply is to isolate the bench power supply from the computer power supply when we were debugging. The A2 debug pin is used when the debug flag is set to true and was used to determine when the Arduino was polling the IR signal line by setting it HIGH then LOW briefly. This let us scope both the signal and A2 to determine whether the Arduino code was functioning as intended.

# 13. Serial to Parallel Conversion

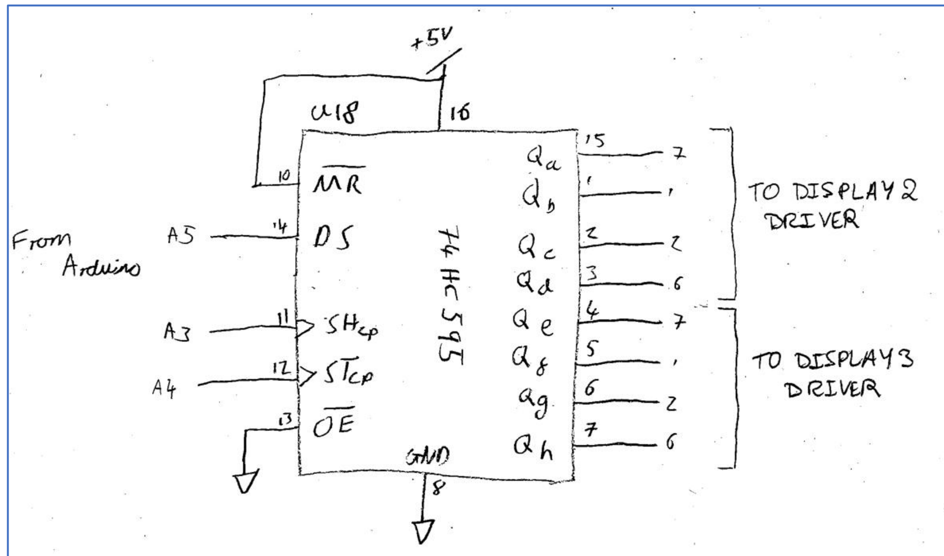



Figure 17

## Sub-Circuit Functionality and Explanation

The 74HC595 receives serial data on DS on a rising edge on SHcp, and as it is double buffered, an extra rising edge on STcp moves the data, that was just shifted into the outputs. This shift register receives two four-bit numbers from the Arduino that are used to display the address and question numbers on the Numitrons.

## Sub-Circuit Design Justification

The 74HC595 serial to parallel shift register lets us connect two 4-bit display drivers, thus using only three pins on the Arduino instead of eight. It also adds diversity to our digital circuitry as we are using a new component over a ‘161 counter again.

# 14. Question and Address Displays

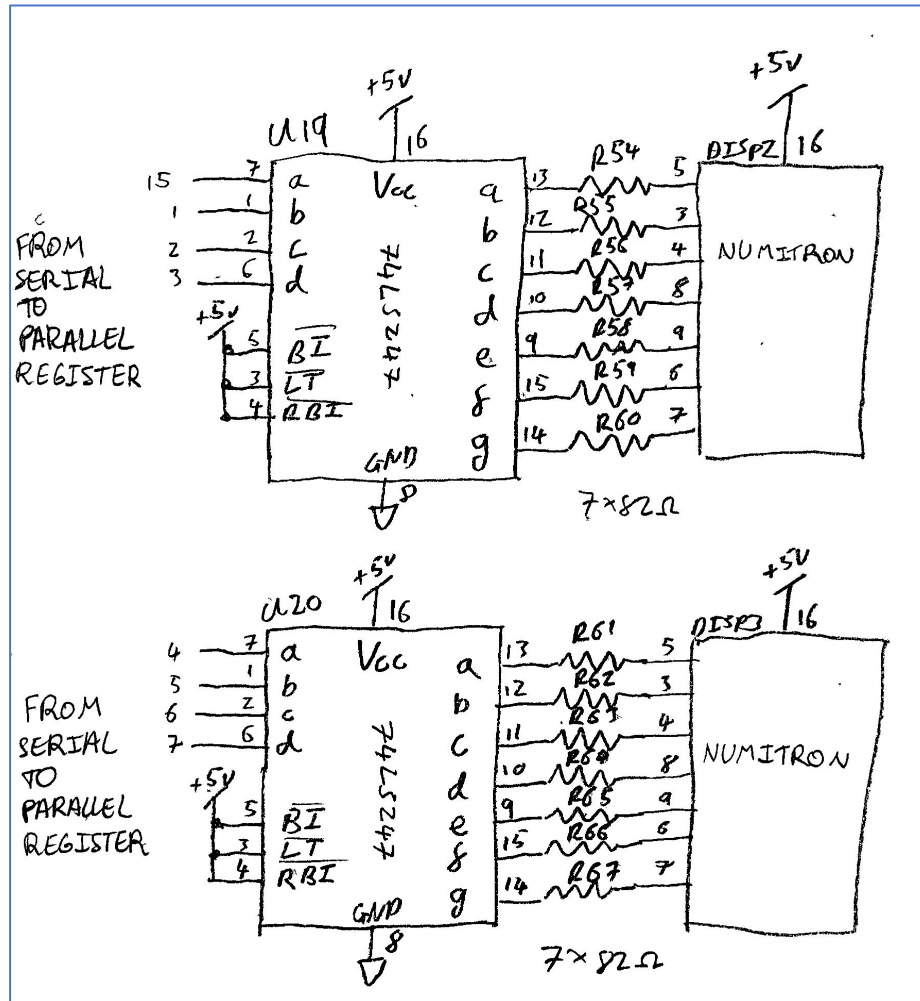



Figure 18

## Sub-Circuit Functionality and Explanation

The two 74LS247 7-segment drivers receive a 4-bit number from the shift register U18. DISP2 shows the address and DISP3 shows the associated question number. They are both connected to a Numitron with 82Ω resistors.

## Sub-Circuit Design Justification

The design justification for the resistor values follows along the same lines as it does for part 3. We decided to build this circuit to add functionality to the design, so that the user could see both the address and the question number simultaneously, rather than having only one or the other. It also allowed us to use all the bits on the shift register U18.

# 15. Harmonics Filtering

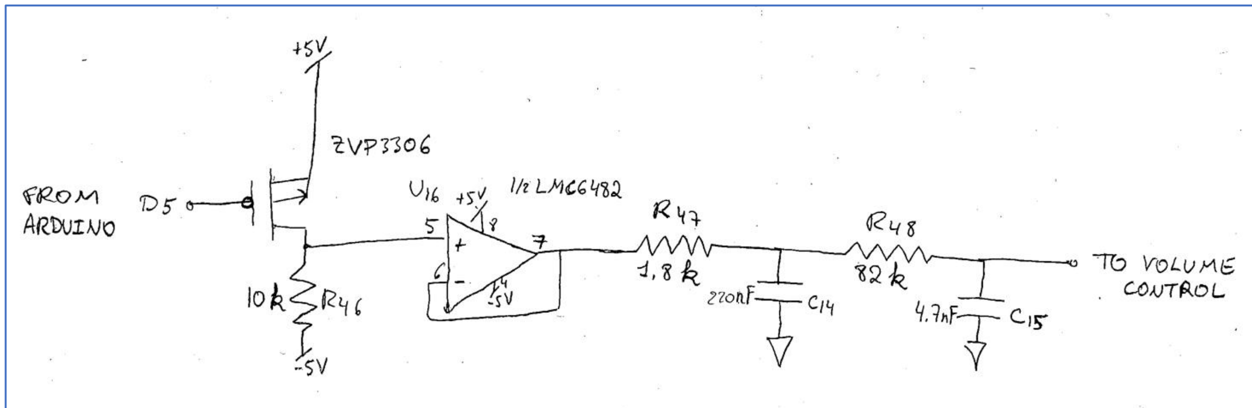



*Figure 19*

## Sub-Circuit Functionality and Explanation

A level shifter creates a -5v – +5v square wave from the 0v - 5v Arduino output. This is buffered to avoid thevenin loading issues and passed through two low-pass filters to filter out the higher harmonics of the square wave to attain an approximate sine wave, which is more pleasant to the ear of the frustrated and tired TFs and students alike (fig.19). This filtered signal is then connected to a volume control.

## Sub-Circuit Design Justification

We used the level shifter to maximise the volume of the speaker, by giving us a 10v difference rather than a 5v difference. The output is then buffered so that we are able to use a lower value resistor than 100kΩ to let us cascade two first order filters together. We initially tried to use an inverting integrator op-amp to convert the square wave to a triangle wave and then filter the attenuated harmonics, but when we built and tested this, we encountered the issue of not being able to centre the integration accurately, resulting in different DC offsets and amplitudes every time the square wave was generated. We thus decided to just filter the square wave directly. With Fpass = 200Hz – this being the maximum frequency we are driving the speaker at – the RC value we calculated for the two low-pass filters was 3.98E-4. With thevenin loading taken into account we chose the values R47 = 1.8kΩ, C14 = 220nF, R48 = 82kΩ and C15 = 4.7nF, based on what was available in the lab.

*Figure 20: The yellow trace shows the Arduino square wave output. The blue trace shows the signal after the first filter and the purple trace, after the third stage. The purple trace approximates a sine wave well.*

# 16. Volume Control

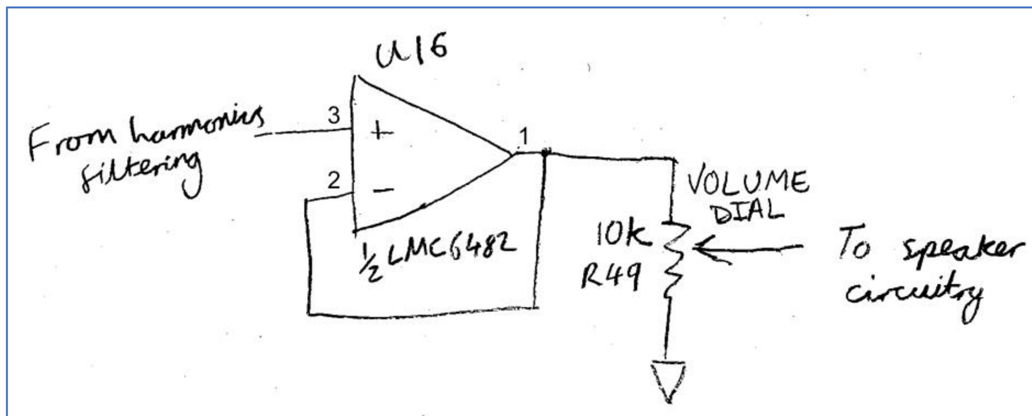



*Figure 21*

## Sub-Circuit Functionality and Explanation

The potentiometer acts as a volume control dial to allow the TFs to lower the volume of the speaker, should the students become too trigger-happy with requests or indeed, want to ignore the students entirely.

## Sub-Circuit Design Justification

This simple circuit buffers the voltage from the filtering circuit to avoid the thevenin loading issues with using a 10kΩ potentiometer. We had a spare op-amp in the U16 package so this was not a wiring issue.

# 17. Speaker Circuitry

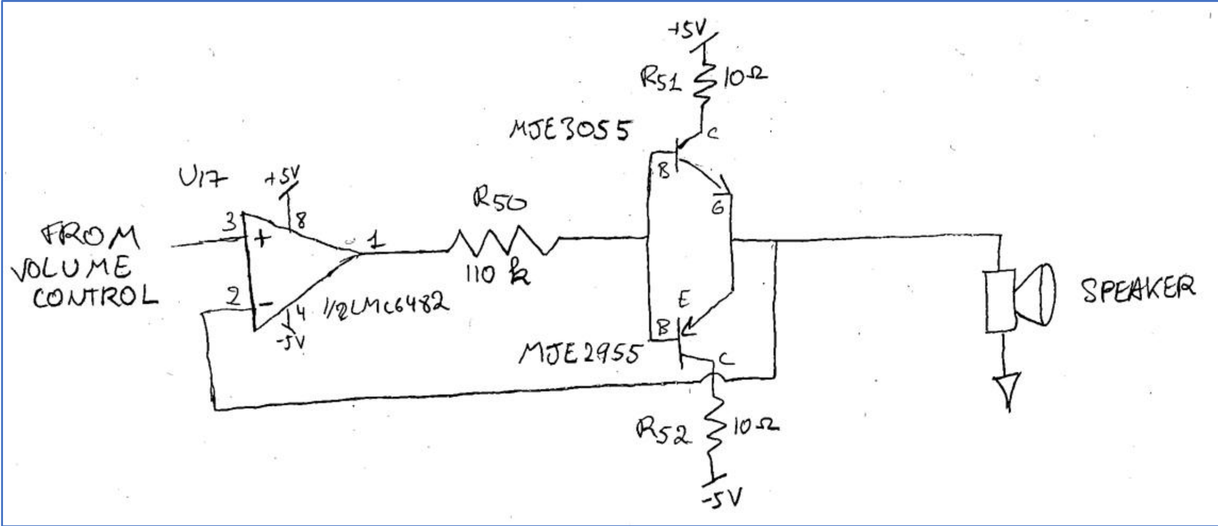



*Figure 22*

## Sub-Circuit Functionality and Explanation

The volume control output is connected to a Class-B push-pull amplifier that amplifies the current supplied to the speaker. This is used to give the TFs an audio indication of when a request is made and cleared.

## Sub-Circuit Design Justification

We chose the push-pull configuration as we had a design from the Theremin lab we could modify to suit our purposes, which was especially useful given the time pressure we were under at this stage of the process. The 10Ω resistors are simply fuse resistors that will burn up if there is a short circuit somewhere, rather than destroying the power supply or the BJTs. The negative feedback to the op-amp solves the issue of having the ±0.6v transition where the emitter is at 0v that is characteristic of BJTs.

# III. Software and Protocols

## Transmission Protocol

We are using a normally low asynchronous NRZ (non-return to zero) data communication protocol. To begin transmission and to indicate to the receiver that there is an incoming signal, the transmitter sends a start bit (0) followed by 8 bits of data, terminated with an end bit (0). In total, 10 bits are transmitted that can be parsed by the receiver to determine whether the signal is a valid one as described in the function comments.

| Bit 0 | Bit 1 | Bit 2 | Bit 3 | Bit 4 | Bit 5 | Bit 6 | Bit 7 | Bit 8 | Bit 9 |
|---|---|---|---|---|---|---|---|---|---|
| START | Question (1) | Question (2) | Question (4) | Question (8) | Address (4) | Address (2) | Address (1) | Clear | END |

*Figure 23: Bit protocol*

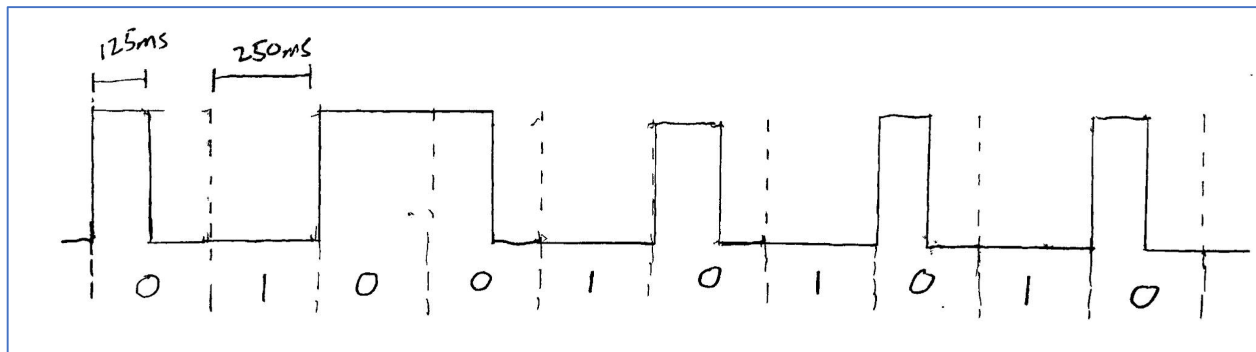



*Figure 24: Timing graph of NRZ protocol.*

## Transmitter Microcontroller

The Finite State Machine code on the transmitter reads the data from the address selectors, clear bit selector and question selector from the parallel to serial shift register and generates a series of outputs to transmit the data when the transmit button is pressed. It also illuminates an LED to indicate whether there is currently an active request or not.

When transmitting, the FSM loop looks at the next bit to transmit and goes to the appropriate transmission state accordingly, while incrementing a 'bits transmitted' counter. When the start bit, the eight data bits and end bit have been transmitted, the FSM loop returns to either state A or D depending on request status.

As the transmission circuit uses a p-channel MOSFET to switch the transmission frequency, the signal it transmits is an active LOW signal i.e. a high-bit at the receiver is a 0 transmitted bit.

This is because during the construction phase, we used an n-channel MOSFET before switching to the final solution. Rather than modify and test the code again, we decided that it would be simpler to solve the issue in software on the receiver microcontroller.

The push button is on an interrupt and when triggered on the falling edge, the interrupt service routine sets a flag to inform the foreground loop that there has been a button press. This means that the code can run at a 4Hz FSM loop frequency to make the states simpler with no loop counters and still respond seemingly immediately to a button press.

| State | Description | Transmission (A0) | Status LED (A1) |
|---|---|---|---|
| A | Non-active status Waiting for button press. When pressed, check clear bit is not asserted. | 0 | 0 |
| B | Activating Request. transmit 0. | 1 | 1 |
| C | Activating request. Transmit 1. | 0 | 1 |
| D | Active status. Waiting for button press. When pressed, check clear bit is asserted. | 0 | 1 |
| E | Deactivating request. Transmit 0. | 1 | 0 |
| F | Deactivating request. Transmit 1. | 0 | 0 |

*Figure 25: Transmission FSM states*

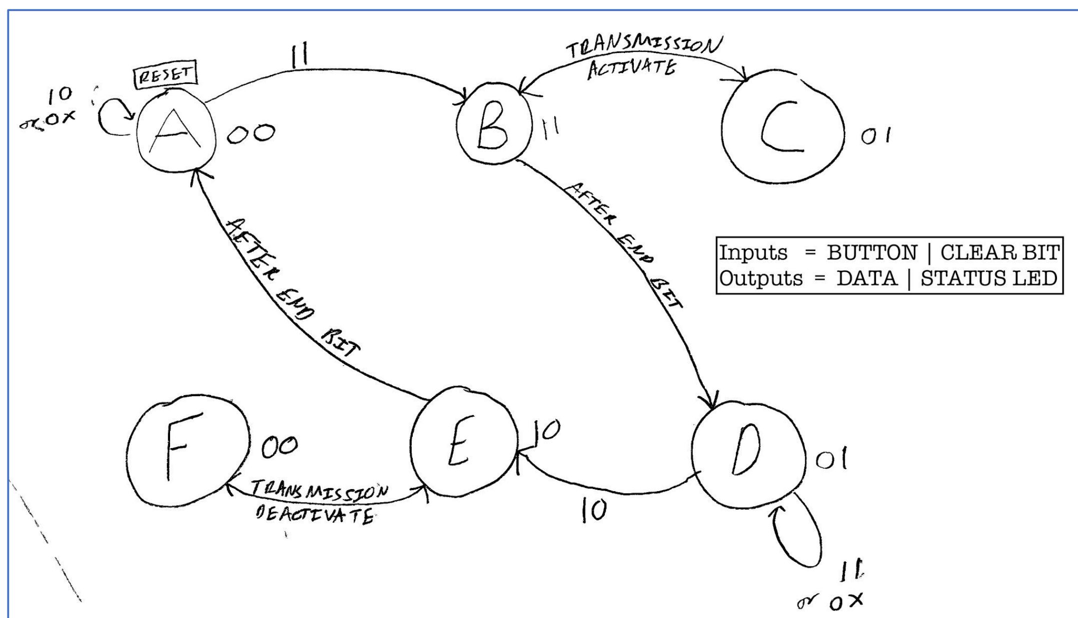



*Figure 26: FSM Directed Graph*

## Receiver Microcontroller

The receiver microcontroller software has two functions: to display the current address number and associated question number and to process the received signal and take action based on them. When the button is pressed, an interrupt flag is set that tells the foreground program to loop to the next address number.

The IR signal pin is not on an interrupt as if a false signal is repeatedly arriving, the program would stall. The signal pin is polled and when the start 0 bit is detected, it begins a routine, whereby before entering each next state, the program checks whether one period of the data rate has elapsed. When it does, the program polls the signal pin again and adds the value to a variable and increased a ‘bits received’ counter by 1. When the counter reaches 10, the program terminates the signal collection routine and parses the signal, checking whether it is a valid signal and if so, extracting the address, question and clear bit status. The program then displays this on the displays and makes a sound.

We decided to implement the signal validation function as we found during testing that when the PLL occasionally unlocked, the Arduino was returning incorrect data with no way of correcting itself. This would mean that we would have to reset the Arduino, which in the context of stressed Office Hours, is not the best user experience. The validation, while not foolproof, Allows the Arduino to discard many invalid signals due to PLL unlocking.

While the speaker is playing, the Arduino is still looping through the FSM loop and is still able to cycle displays when the button is pressed. It will not accept a new incoming signal until the speaker has stopped playing.

When debug mode is activated, the Arduino pulses HIGH very quickly on pin A2 in order to be able to see when the Arduino is sampling the IR signal. This was very useful in debugging the code and timings initially (fig. 28).

| State | Description | Displays | Speaker |
|---|---|---|---|
| A | Standby State. Awaiting button press. Awaiting rising edge on IR signal input. | No change | |
| B | Update displays state. Button pressed. Clear flag. | Cycle address display (0-7) and display question number for corresponding address. 0 if no active request. | |
| C | Start bit received. Begin signal acquisition. | No change | |
| D | When one transmission clock cycle has elapsed, poll the line and add the bit to the incoming signal. Terminate acquisition if final bit. | No change | |
| E | Signal acquisition complete. Verify signal. If verified, process signal and display data. Reset FSM for next acquisition cycle. | Address and question of received signal. | |
| F | Valid active request received. Speaker gives audio indication of this. Remain in this state for user-given amount of time. | No change | ‘Received’ tone. |
| G | Valid clear request received, speaker gives audio indication with different frequency. Remain in this state for user-given amount of time. | No change | ‘Cleared’ tone. |

*Figure 27: Receiver FSM states*

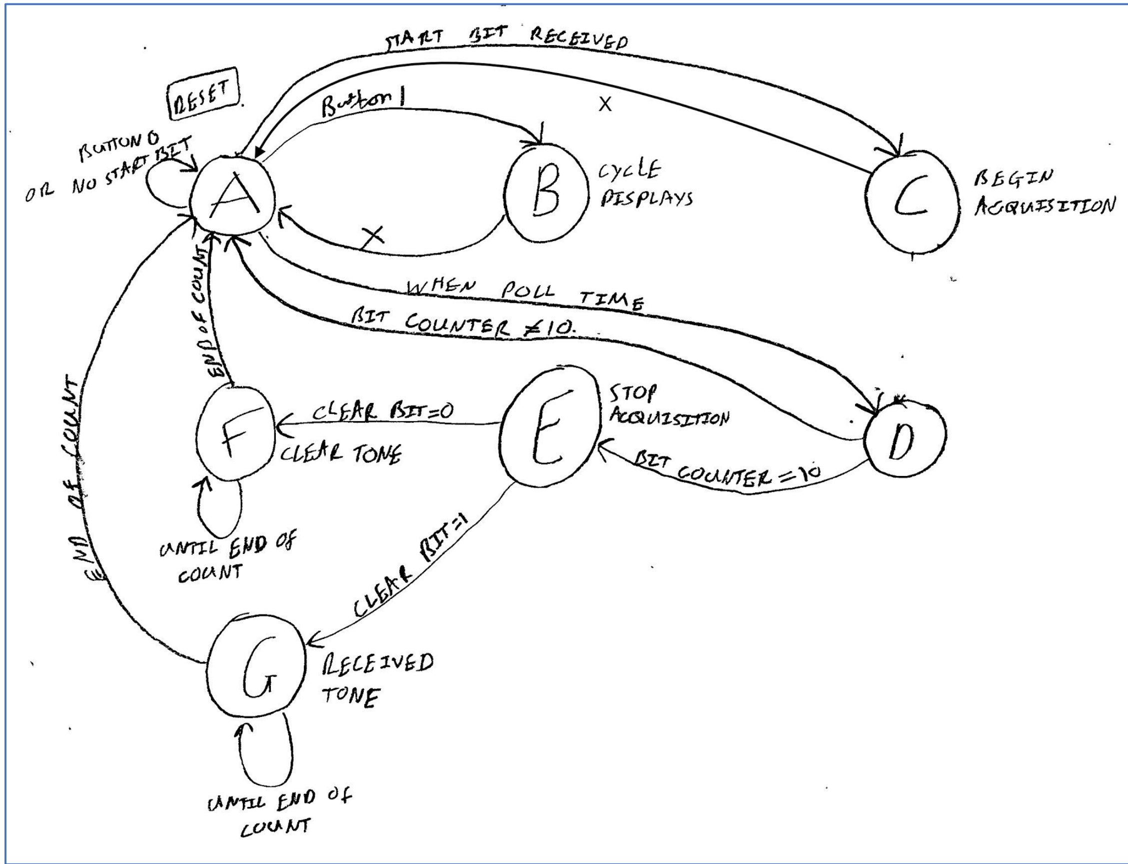



Figure 28: FSM Directed graph

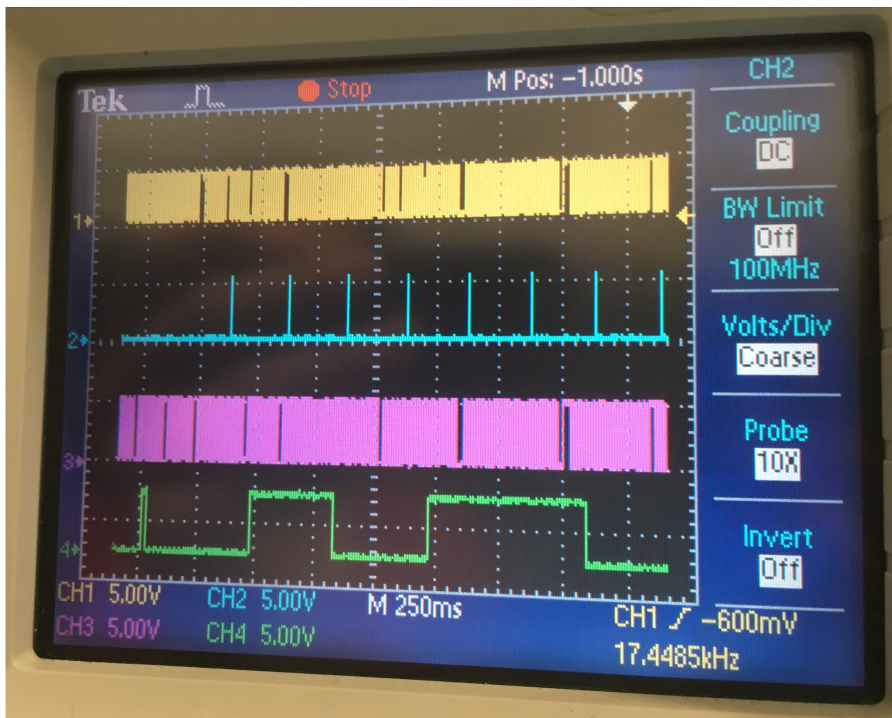



Figure 29: Oscilloscope demonstrating the debug pin A2 (blue) and the incoming IR signal (green)

# IV. Design Narrative

When we were discussing different ideas for the final project, we began to realise that we both wanted something that was fairly practical, and that we did not want to create a game. Then, as we have both been in office hours and experienced the difficulty of their being so busy that it is almost impossible to get a TF's attention, we had the idea for the TF button.

When we had decided on the theme of our project, we talked to Professor Abrams about it, who suggested that we look into IR transmission. We also spoke to Cal Miller about his project from last semester in which he used ultrasound, and we learned a bit about how he had used a phase-locked loop in order to do frequency demodulation. Andrea began to research the phase-locked loop, and Nicolas started on the emitter digital design.

At first, Andrea looked up the individual parts of the phase-locked loop, such as the phase comparator and the voltage controlled oscillator. She originally thought that we would have to build the entire thing ourselves; however, Professor Abrams mentioned that there was a PLL chip, the MC74HC4046 that we actually had in lab. At that point, Andrea's job became reading through the datasheets and application notes in order to figure out how to implement the chip in the way that we wanted. Eventually, she found the application note for the implementation of the PLL chip for frequency demodulation, and following the instructions and information provided she calculated the required resistor and capacitor values, and figured out how to configure the chip.

Nicolas, meanwhile, designed the emitter. Other than the digital logic we had learned in class (using components such as counters) he also looked into a new chip, a double buffered parallel to serial converter 74HC597. Looking through datasheets, he eventually understood how the chip worked and implemented it. He also wrote the FSM code for the Arduino, which controls the transmission circuitry.

We both worked on the receiver in order to sort out the unforeseen issues that cropped up. While we knew from the start that we wanted a speaker to make some noise and some numitrons to display the address and question number, we first had to clean up the signal that we received. The phototransistor voltage needed to be processed by amplifying it and turning it into a square wave. The PLL had to be tuned to the frequency of the emitter, which proved to be harder than expected. The output of the PLL was so noisy that we needed to modify it quite a bit before we could put it through a comparator.

The design of this part was done by both of us as we designed filters and op-amp circuits to get the result that we wanted. We ran into a big issue when we discovered that our ground rail itself was experiencing ±0.2v of noise, which rendered our filtering and signal processing efforts useless. After establishing that the PLL chip was the one component responsible for the noise, we tried several tricks suggested to us by Professor Abrams, such as running direct ground wires to the chip, to no avail. After much frustration and time, Professor Abrams suggested we use a difference amplifier to compensate for the noise. After Nicolas read through the

application notes and built the circuit, we had much less noise on the signal and were able to do further processing.

Once this was done, Nicolas dealt with the Arduino code once again and then we designed the speaker circuitry based on the labs that we had done previously. Andrea then dealt with the numitron display while Nicolas figured out how to use the serial to parallel converter 74HC595 drive the two displays from the Arduino. Once we had everything built, to our great delight, it actually worked!

# V. Results

| Parts/Goals | Minimum Goals | Expected Goals | Stretch Goals |
|---|---|---|---|
| **Range** | At least a 30 cm of distance | From one table to another at a distance of 3m | Across the Lowell dining hall |
| **Noise Tolerance** | Works consistently in lab conditions | Works consistently in indoor non-lab conditions | Works consistently outdoors (and at the SEAS design fair) |
| **Scale** | At least three unique button addresses are possible | At least three unique button addresses possible and Users can clear requests once they have been helped | Nine unique button addresses possible and Users can clear requests once they have been helped |
| **Power Supply** | Functions with lab power supplies | Transmitter works off of single supply | Single-supply operation |

**completed and tested successfully** (green)
**not tested** (yellow)
**not achieved** (red)

# VI. Conclusion

Barring a few small mistakes and some things to improve in the future, our project was a success as deemed by the specifications we set out to achieve at the start of the process. It works very well, is easy to use, and is fairly reliable.

In order to improve our design in the future there are a few modifications that we could make. The data rate, currently at a very low 4Hz, could be faster if the RC time constant on the transmitter LM555 control voltage circuit was smaller, but this would mean characterising the PLL in much more detail to determine how much the rate of change of input frequencies would affect it. Unfortunately, we did not have the time for this.

We could also improve our circuit on the receiver by using a more suitable hysteresis value for the post-PLL comparator: 2V would probably be the right value to choose, but as it is currently working we decided to leave our current build alone.

The receiver could also have been made using more digital hardware, by moving some of the functions that are currently being performed by the Arduino off it, such as the initial IR signal acquisition. This would make for a more complicated circuit and it would also rely less on code. Likewise, with the transmitter circuitry, the Arduino was designed with the circuit with the (correct) anticipation of having trouble with the PLL; however, with more time, most if not all of the functions of that Arduino could be done in hardware.

Finally, it would be interesting to add some way of indicating when a bad signal was received, such as when the PLL becomes unlocked midway through transmission. This would be done using code. It would allow the user to know that something had gone wrong with the signal rather than any part of the hardware if the TF button stopped working for some reason. A small LED would achieve this.

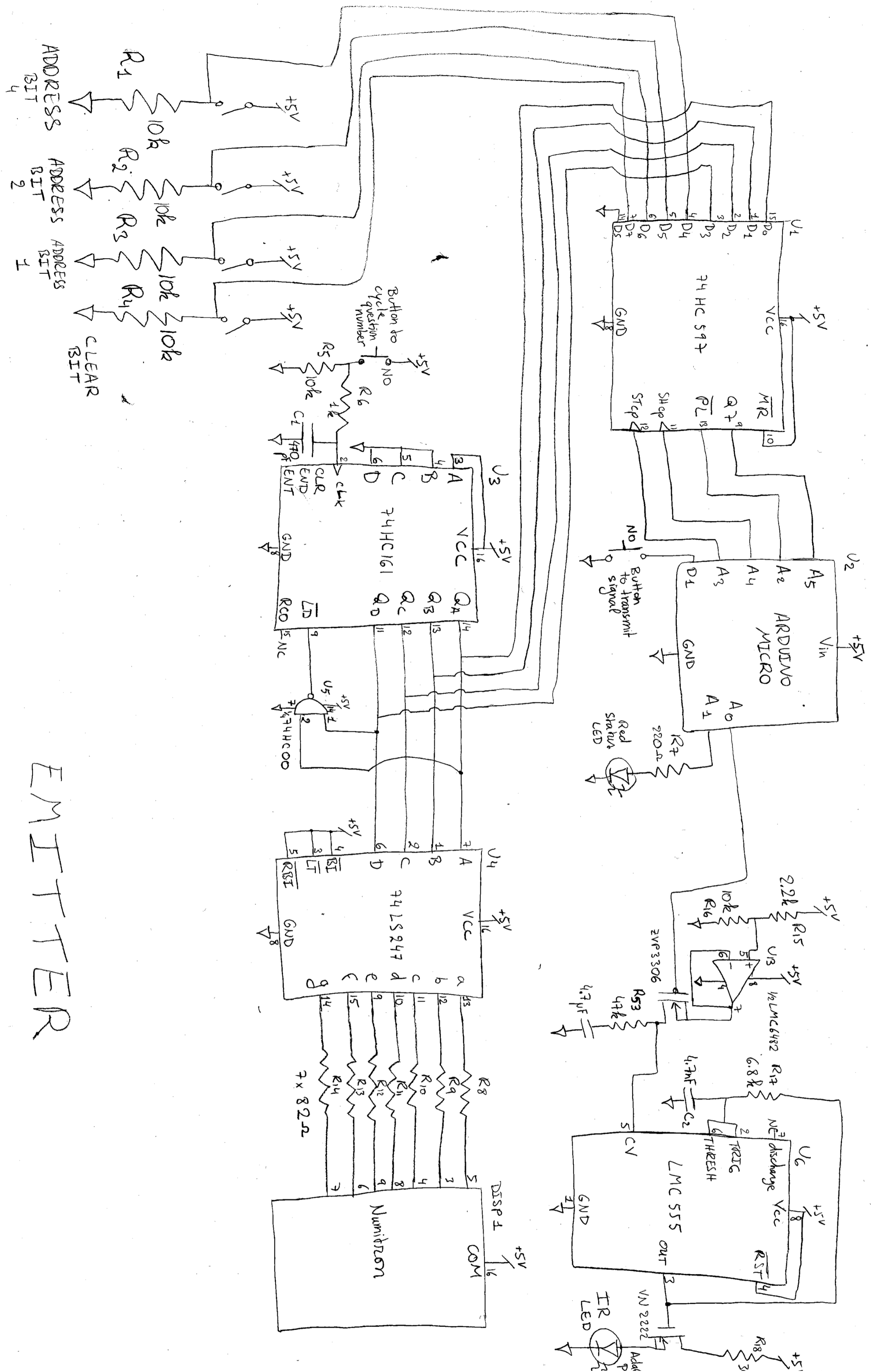
Overall, both of us thoroughly enjoyed ourselves doing this project and indeed over the course of the whole semester. We leave ES 52 with a sound understanding of fundamental circuit design and with the desire to lock ourselves in a lab several times more during final project seasons to come as we progress through our Harvard SEAS careers.

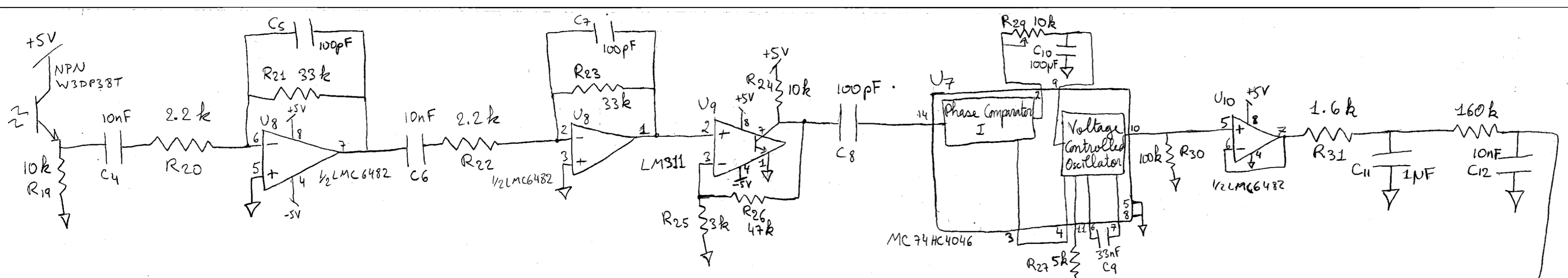
# VII. Bibliography

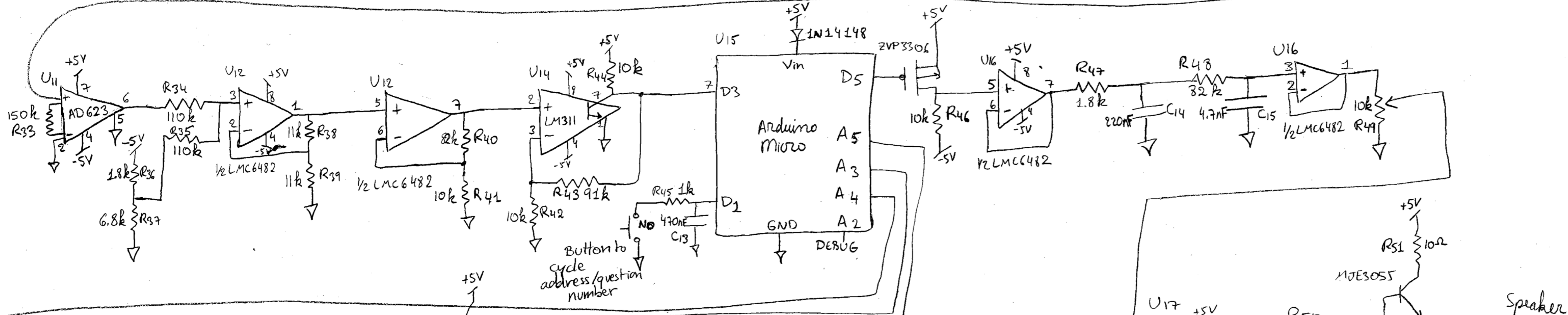
[1] Parallel to Serial Shift Register 74HC597 datasheet
http://assets.nexperia.com/documents/data-sheet/74HC_HCT597.pdf
[2] Serial to Parallel Shift Register 74HC595 datasheet
http://assets.nexperia.com/documents/data-sheet/74HC_HCT595.pdf
[3] PLL application notes
http://www.ti.com/lit/an/scha002a/scha002a.pdf
[4] Difference Amplifier datasheet
http://www.analog.com/media/en/technical-documentation/data-sheets/AD623.pdf
[5] PLL datasheet
http://www.ti.com/lit/an/scha003b/scha003b.pdf

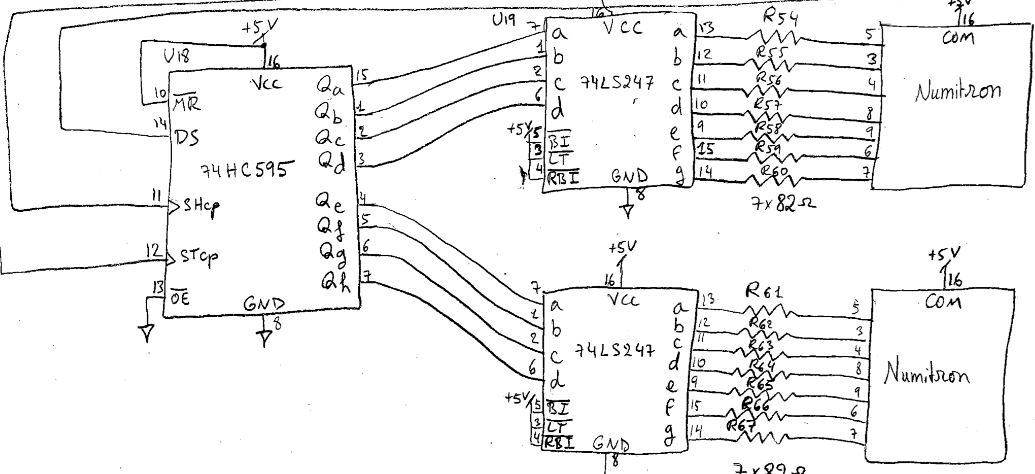
Other research:
IR information
https://learn.sparkfun.com/tutorials/ir-communication
Phase-locked loop overview
http://www.radio-electronics.com/info/rf-technology-design/pll-synthesizers/phase-locked-loop-tutorial.php
How demodulation works with the PLL
http://www.radio-electronics.com/info/rf-technology-design/fm-reception/fm-pll-detector-demodulator-demodulation.php
Infrared emitters
http://www.futureelectronics.com/en/optoelectronics/infrared-emitters.aspx
FM demodulation techniques
http://www.radio-electronics.com/info/rf-technology-design/fm-reception/fm-demodulation-detection-overview.php
One type of demodulator
http://www.eetimes.com/document.asp?doc_id=1275839
"PLL design fundamentals"
http://www.nxp.com/assets/documents/data/en/application-notes/AN535.pdf
PLL design tutorial for beginners
http://ece.wpi.edu/analog/resources/PLLTutorialISSCC2004.pdf
PLL chip datasheet
https://www.onsemi.com/pub/Collateral/MC74HC4046A-D.PDF
PLL chip application notes
http://www.onsemi.com/pub/Collateral/AN1410-D.PDF
Phototransistor datasheet
http://www.kingbrightusa.com/images/catalog/SPEC/WP3DP3BT.pdf

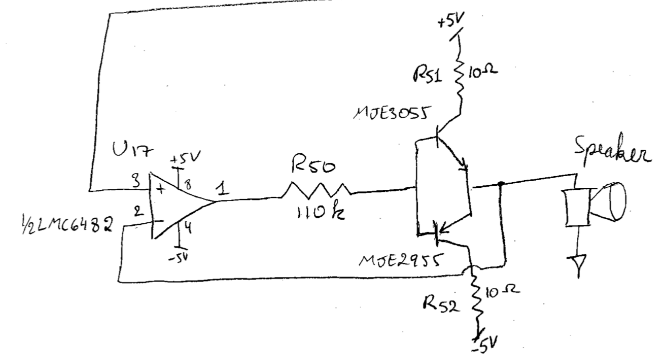
# VIII. Appendices






# RECEIVER

## B. Transmitter Code

```
/*****************************************************************
 * Nicolas Weninger and Andrea Andrea Rodriguez-Marin Freudmann
 *
 * 24 April 2017
 *
 * FSM Code for the transmitter
 *
 * States: (P: pulse-enable, S: status LED)                   OUTPUTS
 * A - Non-active status Waiting for button press. When
 *     pressed, check for clear bit.                           P0 S0
 * B - Activating Request. transmit 0.                         P1 S1
 * C - Activating request. Transmit 1.                         P0 S1
 * D - Active status. Waiting for button press. When
 *     pressed, check clear bit.                               P0 S1
 * E - Deactivating request. Transmit 0.                       P1 S0
 * F - Deactivating request. Transmit 1.                       P0 S0
 *
 * Based on the FSM skeleton by David Abrams
 *
 * MRU:
 * 01/05/2017: Slowed frequency to 4Hz for better transmission.
 *             Removed TRANSMISSION_ENABLE pin to reflect changes
 *             in hardware.
 *
*****************************************************************/

// Constant variables set by the user before compiling to control program behaviour
const boolean debug = true;   // when set to true, debugging statements sent to the PC
const int FSM_FREQ = 4;      // set to frequency of state machine clock in Hz (max
500Hz)

// Program Variables

const int FSM_TIME = 1000/FSM_FREQ; // milliseconds per state cycle (do not change
this variable)
unsigned long CycleStart;
int CurState;                 // holds current/next state
int BitCounter = 0;           // tells program what bit we are reading
volatile boolean startTransmission = 0;
byte ShiftData = 0;

// Hardware IO

const int SDIPIN = A5;
const int PLPIN = A2;
const int STCLK = A3;
const int SHCLK = A4;
const int BTNPIN = 1;

const int STATUS_LED = A1;
const int DATA_PULSE = A0;



void setup()
{
   CurState = 1;              // initialize for first state in the program loop

   /* 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.");
   }

    pinMode(DATA_PULSE, OUTPUT);
    pinMode(STATUS_LED, OUTPUT);
    pinMode(PLPIN, OUTPUT);
    pinMode(SHCLK, OUTPUT);
    pinMode(STCLK, OUTPUT);
    pinMode(SDIPIN, INPUT);
    pinMode(BTNPIN, INPUT_PULLUP);

    digitalWrite(DATA_PULSE, LOW);
    digitalWrite(STATUS_LED, LOW);
    digitalWrite(STCLK, LOW);
    digitalWrite(SHCLK, LOW);
    digitalWrite(PLPIN, HIGH);

    attachInterrupt(digitalPinToInterrupt(BTNPIN), ISRStartTransmission, FALLING);

}

//****************************
// FINITE STATE MACHINE LOOP
//****************************
void loop()
{

   CycleStart = millis();           // get time we started this FSM cycle

   if (debug)
   {
      Serial.print("Current State = ");
      Serial.println(CurState);
      Serial.println(readData(), BIN);
      Serial.println(BitCounter);
   }

  switch (CurState)
   {
      case 1:
         digitalWrite(DATA_PULSE, LOW);
         digitalWrite(STATUS_LED, LOW);

         if (startTransmission == 1 && !isClear() == 1) {
            startTransmission = 0;
            CurState = 2;
         }
         else {
            startTransmission = 0;
            CurState = 1;
         }
         break;

      case 2:
         digitalWrite(DATA_PULSE, HIGH);
         digitalWrite(STATUS_LED, HIGH);
         ShiftData = readData();
```

```
    if (BitCounter == 8) {
       BitCounter = 0;
       CurState = 4;
    }
    else if (bitRead(ShiftData, BitCounter) == 0) {
       BitCounter++;
       CurState = 2;
    }
    else {
       BitCounter++;
       CurState = 3;
    }
    break;

 case 3:
    digitalWrite(DATA_PULSE, LOW);
    digitalWrite(STATUS_LED, HIGH);

    if (BitCounter == 8)
       CurState = 2;
    else if (bitRead(ShiftData, BitCounter) == 0) {
       BitCounter++;
       CurState = 2;
    }
    else {
       BitCounter++;
       CurState = 3;
    }
    break;

 case 4:
    digitalWrite(DATA_PULSE, LOW);
    digitalWrite(STATUS_LED, HIGH);

    if (startTransmission && isClear()) {
       startTransmission = 0;
       CurState = 5;
    }
    else {
       startTransmission = 0;
       CurState = 4;
    }
    break;

 case 5:
    digitalWrite(DATA_PULSE, HIGH);
    digitalWrite(STATUS_LED, LOW);
    ShiftData = readData();

    if (BitCounter == 9){
       BitCounter = 0;
       CurState = 1;
    }
    else if (bitRead(ShiftData, BitCounter) == 0){
       BitCounter++;
       CurState = 5;
    }
    else {
       BitCounter++;
       CurState = 6;
```

```
        }
        break;

      case 6:
        digitalWrite(DATA_PULSE, LOW);
        digitalWrite(STATUS_LED, LOW);

        if (bitRead(ShiftData, BitCounter) == 0) {
          BitCounter++;
          CurState = 5;
        }
        else {
          BitCounter++;
          CurState = 6;
        }
        break;

      default:
        {
        }
    } // end of switch statement

    // wait one cycle before entering the next state (this simulates the FSM clock)
    while (millis() < (CycleStart + FSM_TIME)) {
    }                                                 // wait one FSM cycle
}


//**************************************************************************
// Function to read the data from the shift register.
//**************************************************************************
byte readData()
{
  digitalWrite(PLPIN, LOW);
  digitalWrite(STCLK, HIGH);            // extra clock pulse to load data into
register.
  digitalWrite(STCLK, LOW);
  digitalWrite(PLPIN, HIGH);            // the 74HC597 reads out the first bit BEFORE
the first rising edge.
  int data = digitalRead(SDIPIN) << 8;  // make space for the 8 bits from shiftIn()
  data = data + shiftIn(SDIPIN, SHCLK, MSBFIRST);  // shift in 7 bits and one
undefined bit.
  data = data >> 1;                     // shift out the LSB undefined bit and replace
MSB with first read bit
  return data;
}

//**************************************************************************
// Function to poll the clear bit. Returns true when clear switch is HIGH.
//**************************************************************************
boolean isClear()
{
  return !bitRead(readData(),7);

}

//**************************************************************************
// ISR for transmit button interrupt. Sets flag.
//**************************************************************************
void ISRStartTransmission()
{
```

```
  startTransmission = 1;
}
```

## C. Receiver Code

```
/**************************************************************
 * Nicolas Weninger and Andrea Rodriguez-Marin Freudmann
 *
 * 1 May 2017
 *
 * FSM Code for the receiver
 *
 * States:
 * A - Standby state. Awaiting button press. Awaiting rising edge on
 *     IR signal input.
 * B - Button pressed. Cycle address display (0-7) and
 *     display question number for corresponding
 *     address. 0 if no active request. Clear flag.
 * C - Start bit received. Begin signal acquisition.
 * D - When one transmission clock cycle has elapsed,
 *     poll the line and add the bit to the incoming
 *     signal. Terminate acquisition if final bit.
 * E - Signal acquisition complete. Verify signal.
 *     If verified, process signal and display data.
 *     Reset FSM for next acquisition cycle.
 * F - If active request received, Speaker gives audio
 *     indication of this. Remain in this state for
 *     user-given amount of time.
 * G - If clear request received, speaker gives audio
 *     indication with different frequency. Remain in
 *     this state for user-given amount of time.
 *
 * Based on the FSM skeleton by David Abrams
 *
 * MRU:
 * 2/5/2017 - Made the speaker output active low to reduce
 *            quiescent current draw.
 *            Fixed getQuestionNumber() function
 * 2/5/2017 - Added extra state to make speaker sound for a
 *            clear request.
 * 3/5/2017 - Fixed bug in code that would *occasionally* make
 *            project not work at the design fair. An unlocked
 *            PLL gives a high signal on IRSIGNALPIN, and the
 *            FSM was not resetting to a non-reading state
 *            correctly despite recognising that the signal was
 *            not valid. Added CurState = 1 to state E.
*************************************************************/
// define hardware IO pins
const int DATAPIN = A5;
const int CLOCKPIN = A3;
const int LATCHPIN = A4;
const int SPEAKERPIN = 5;
const int DEBUGPULSE = A2;

const int BTNPIN = 1;
const int IRSIGNALPIN = 3;
```

```
// Constant variables set by the user before compiling to control program behaviour
const boolean debug = true;   // when set to true, debugging statements sent to the PC
const int FSM_FREQ = 50;      // set to frequency of state machine clock in Hz (max
500Hz)
const int TRANSMITTER_FREQ = 4; // freqency the data is being transmitted at
const int SPEAKERFREQ = 261;    // frequency the speaker plays when a signal is
received
const int SPEAKERFREQ_2 = 196;  // frequency speaker plays when clear request received
const int SPEAKERONTIME = 2;  // length of time the speaker is on in seconds. (min 1
sec).


// Program Variables
const int FSM_TIME = 1000/FSM_FREQ; // milliseconds per state cycle (do not change
this variable)
const int SPEAKER_CYCLES = FSM_FREQ * SPEAKERONTIME;
unsigned long CycleStart;
int CurState;                 // holds current/next state

volatile boolean ButtonFlag = false;   // flag for when button pressed on interrupt
boolean SpeakerFlag = false;   // flag for when speaker is playing a tone
boolean SpeakerFlagClear = false;   // flag for when speaker is playing a tone for
CLEAR request
int CurrentAddress = 0;       // holds the current state of the 'address display'
int Questions[] = {0,0,0,0,0,0,0,0}; // holds the current question being asked by the
associated address
int IncomingSignal = 0;   // the incoming signal. ACTIVE LOW SIGNAL.
byte ProcessedSignal = 0;  // the processed signal. ACTIVE HIGH SIGNAL.
int StateCounter = 0;               // a counter that lets us delay the FSM without
using delay()

unsigned long SignalStartTime = 0;     // the time the start bit was detected
int CurrentBit = 0;                    // the current bit that was just read



void setup()
{
   CurState = 1;              // initialize for first state in the program loop

   /* 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.");
     pinMode(DEBUGPULSE, OUTPUT);
     digitalWrite(DEBUGPULSE, LOW);
   }

   /* Setup Hardware pins */
```

```
    pinMode(SPEAKERPIN, OUTPUT);
    pinMode(LATCHPIN, OUTPUT);
    pinMode(DATAPIN, OUTPUT);
    pinMode(CLOCKPIN, OUTPUT);
    pinMode(BTNPIN, INPUT_PULLUP);
    pinMode(IRSIGNALPIN, INPUT);

    digitalWrite(SPEAKERPIN,HIGH);
    digitalWrite(LATCHPIN,LOW);
    digitalWrite(DATAPIN,LOW);
    digitalWrite(CLOCKPIN,LOW);

    displayWrite(0,0);

    /* Setup interrupt pin for button */
    attachInterrupt(digitalPinToInterrupt(BTNPIN), ISRCycleDisplay, FALLING);
}

//****************************
// FINITE STATE MACHINE LOOP
//****************************
void loop()
{
  // Things to do before FSM loop
  CycleStart = millis();           // get time we started this FSM cycle


  if (CurrentBit != 0 && (millis() >= SignalStartTime + (CurrentBit*250)))  // read
next bit on the line when 250ms has elapsed
  {
    CurState = 4;
  }
  else if (SpeakerFlag)         // make noise if a valid signal has been received
  {
    CurState = 6;
  }
  else if (SpeakerFlagClear)  // make noise if valid clear signal has been received.
  {
    CurState = 7;
  }

  if (debug)
  {
    Serial.print("Current State = ");
    Serial.println(CurState);
  }

  switch (CurState)
   {
      // State A
      case 1:
         speakerOff();
```

```
         if (isButtonDown())
            CurState = 2;
         else if (pollIRSignal() && CurrentBit == 0)   // only begin signal aquisition
if there is no signal being aquired currently
            CurState = 3;
         else
            CurState = 1;
         break;

      // State B
      case 2:

         CurrentAddress = (CurrentAddress + 1) % 8;
         displayWrite(CurrentAddress, Questions[CurrentAddress]);

         CurState = 1;

         break;

      // State C
      case 3:
         IncomingSignal = 1;
         CurrentBit = 1;
         SignalStartTime = millis();

         if (debug)
         {
           digitalWrite(DEBUGPULSE, HIGH);     // Allows us to see on the oscilloscope
when the signal is being polled
           delay(3);                           // needed to see the trace
           digitalWrite(DEBUGPULSE, LOW);
         }

         CurState = 1;
         break;

      // State D
      case 4:

         IncomingSignal = (IncomingSignal << 1) + pollIRSignal();
         CurrentBit++;

         if (debug)
         {
            Serial.print("CurrentBit: ");
            Serial.println(CurrentBit);
            digitalWrite(DEBUGPULSE, HIGH);     // Allows us to see on the
oscilloscope when the signal is being polled
            delay(3);                           // needed to see the trace
            digitalWrite(DEBUGPULSE, LOW);
         }

         if (CurrentBit == 10)
```

```
        {
           CurrentBit = 0;
           CurState = 5;
        }
        else
        {
           CurState = 1;
        }

        break;

     // State E
     case 5:
        if (debug)
        {
           Serial.print("Signal: ");
           Serial.println(IncomingSignal,BIN);
        }
        if (signalCheck(IncomingSignal))    // if signal is verified, begin
processing
        {
           ProcessedSignal = processSignal(IncomingSignal);
           CurrentAddress = getAddress(ProcessedSignal);
           if (getClearBit(ProcessedSignal))  // the signal is a new request
           {
             Questions[CurrentAddress] = getQuestionNumber(ProcessedSignal);  // set
the question for the address
             if (debug){
               Serial.println(Questions[CurrentAddress], BIN);
               Serial.println(ProcessedSignal,BIN);}
             CurState = 6;                                                    // give
audio indication
             SpeakerFlag = true;
           }
           else                             // the signal is a clear request
           {
             Questions[CurrentAddress] = 0;  // clear the request
             CurState = 7;                   // give audio indication
             SpeakerFlagClear = true;
           }
           displayWrite(CurrentAddress, Questions[CurrentAddress]);  // display most
recent update on displays
        }

        else                         // signal is not a valid signal
        {
           IncomingSignal = 0;       // clear temporary variables and reset to state
A.
           ProcessedSignal = 0;
           CurrentBit = 0;
           CurState = 1;
        }
```

```
         break;

      // State F
      case 6:
         speakerOn();
         StateCounter++;

         if (StateCounter == SPEAKER_CYCLES)  // when speaker has been on for long
enough, reset to state A
         {
            CurState = 1;
            SpeakerFlag = false;
            StateCounter = 0;
         }
         else if (isButtonDown())       // Allow TFs to still cycle through requests
even if speaker is playing
         {
            CurState = 2;
         }

         break;  // if SpeakerFlag not cleared, speaker will continue to play

      // State G
      case 7:
         speakerOnClear();
         StateCounter++;

         if (StateCounter == SPEAKER_CYCLES)  // when speaker has been on for long
enough, reset to state A
         {
            CurState = 1;
            SpeakerFlagClear = false;
            StateCounter = 0;
         }
         else if (isButtonDown())           // Allow TFs to still cycle through
requests even if speaker is playing
         {
            CurState = 2;
         }

      default:
         {
         }
   } // end of switch statement

   while (millis() < (CycleStart + FSM_TIME)) {
   }    // wait one FSM cycle
}


//***************************************************************************
// Function to write to the numitron displays by shifting out to register
//***************************************************************************
```

```
void displayWrite(byte Address, byte Question)
{
  byte Data = Address + (Question << 4);
  shiftOut(DATAPIN, CLOCKPIN, MSBFIRST, Data);
  digitalWrite(LATCHPIN,HIGH);
  digitalWrite(LATCHPIN,LOW);               // clocking the register data to the
outputs
}

//**************************************************************************
// ISR when the push button is pressed and sets a flag
//**************************************************************************
void ISRCycleDisplay()
{
  ButtonFlag = true;
}

//**************************************************************************
// function to poll the button flag and clear button flag
//**************************************************************************
boolean isButtonDown()
{
  if (ButtonFlag == true)
  {
    ButtonFlag = false;
    return true;
  }
  return false;
}

//**************************************************************************
// function to poll the IR SIGNAL line
//**************************************************************************
boolean pollIRSignal()
{
  return digitalRead(IRSIGNALPIN);
}

//**************************************************************************
// function to turn speaker on when request received.
//**************************************************************************
void speakerOn()
{
  tone(SPEAKERPIN, SPEAKERFREQ);
}

//**************************************************************************
// function to turn speaker on when CLEAR request received.
//**************************************************************************
void speakerOnClear()
{
  tone(SPEAKERPIN, SPEAKERFREQ_2);
}
```

```
//************************************************************************
// function to turn speaker off
//************************************************************************
void speakerOff()
{
  noTone(SPEAKERPIN);
  digitalWrite(SPEAKERPIN, HIGH);
}

//************************************************************************
// function to verify that the unprocessed signal received was accurate. Returns true
if so.
//************************************************************************
boolean signalCheck(int Signal)
{
  if (bitRead(Signal, 0) && bitRead(Signal, 9)) // start and end bits confirmed
  {
    for (int i = 0; i < 8; i++)
    {
      if (bitRead(Signal, i+1) == 0)
      {
        return true;      // likely to be a valid signal
      }
    }
    return false; // no signal will be all high bits. Any signal that is will be false
positive or incorrectly read.
                  // potentially due to unlocked phase-locked loop or interference.
  }
  else
  {
    return false;
  }
}

//************************************************************************
// function to strip the first and last bit off the raw signal and handle active low.
//************************************************************************
byte processSignal(int Signal)
{
  bitWrite(Signal, 9, 0);
  return (byte)~(Signal >> 1);  // flip bits to make active high
}

//************************************************************************
// function to get address from processed signal. Returns address.
//************************************************************************
byte getAddress(int Signal)
{
  return ((bitRead(Signal, 3)<<2) + (bitRead(Signal, 2)<<1) + (bitRead(Signal, 1)));
}

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

```
// function to get clear bit from processed signal. Returns clear bit.
//***********************************************************************
boolean getClearBit(int Signal)
{
  return (bitRead(Signal, 0));
}

//***********************************************************************
// function to get Question Number from processed signal. Returns question number.
//***********************************************************************
byte getQuestionNumber(int Signal)
{
  return ((bitRead(Signal, 4)<<3) + (bitRead(Signal, 5)<<2) + (bitRead(Signal, 6)<<1)
+ (bitRead(Signal, 7)));
}
```