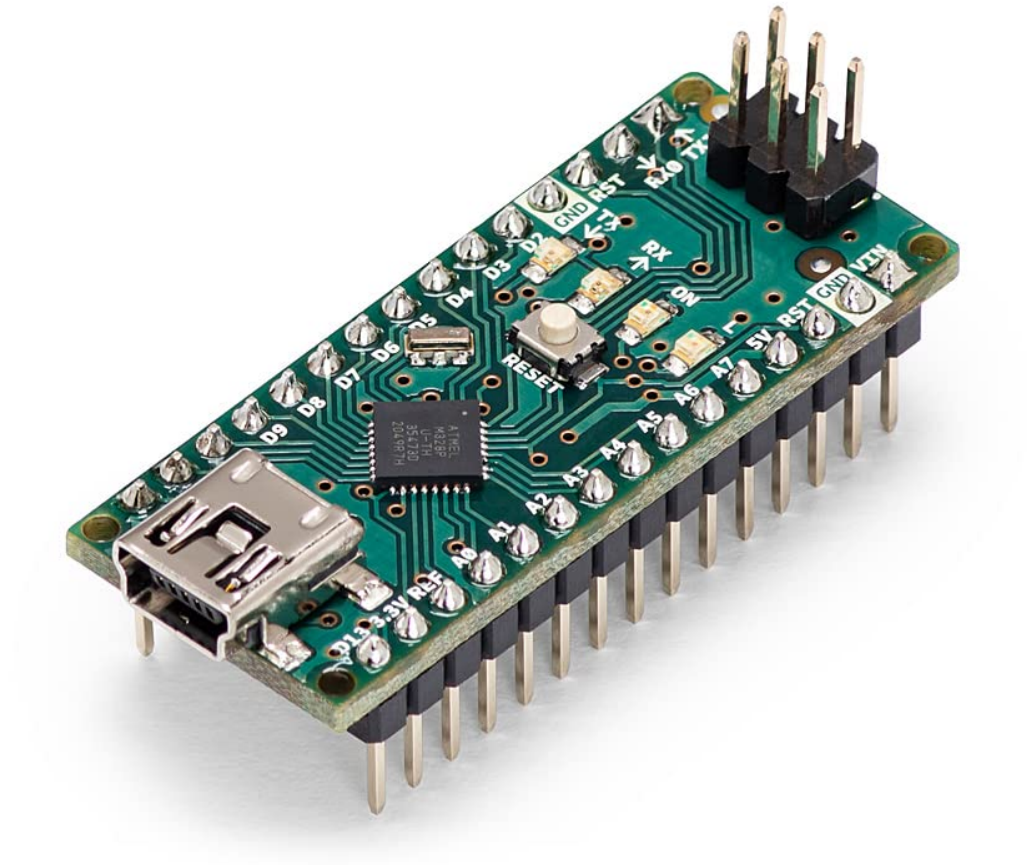
7 Sensor Fusion Project on the Arduino Nano

7.1 Introduction

In this section, we will cover the process of how to communicate messages in the mediums of light and sound. At the same time, we will explore the ideas of sensor fusion, in which we will employ the technique of an OR gate to effectively obtain the correct messages. In other words, this OR gate will allow the message to be received even when one medium of transmission fails during the period.

We will be utilizing the Arduino Nano along with the Arduino IDE to accelerate the message transmission process. The microcontroller allows for a variety of built-in operations, such as flashing an LED or read values of certain pins. This thus eases the burden of our objectives. With this, we will develop code to send and receive information from one Arduino Nano to another. Below is an example of an Arduino Nano.



7.2 Schematic Design

Mentioned above, we will be employing two different microcontrollers. With that said, we will create layouts of two different boards, one for the transmission and the other for the reception.

Our schematic of the transmission board includes an Arduino Nano, a button (connected to D2), an LED (to D3), and a buzzer (to D4). The main purpose of the button is to switch between the light and sound modality (if desired). Moreover, the LED allows for the transmission of a message through light, and the buzzer transmits the message in beeps (sound). The transmission board is as follows.

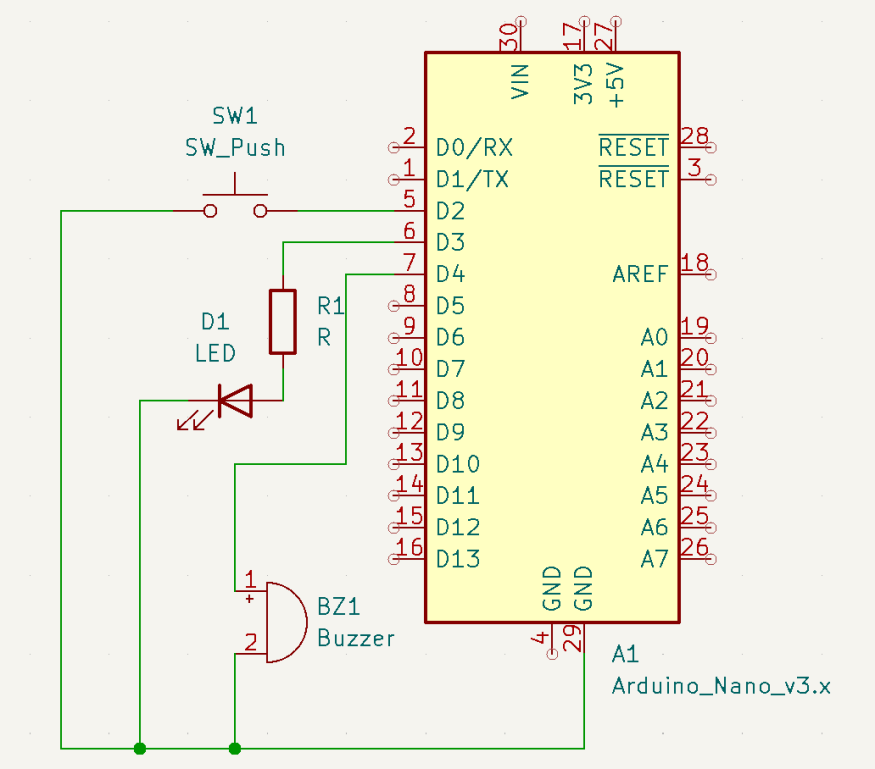


Figure 80: The big yellow board represents the Arduino Nano with all the pin designations, and the SW_PUSH represents the button.

At the same time, our schematic of the reception board includes an Arduino Nano, a button, an LED, an LDR, and a microphone. The button switches between the detection of light and sound. The LED signals the medium in which the message is being detected. The LDR and the microphone detect the message, through light and sound, respectively.

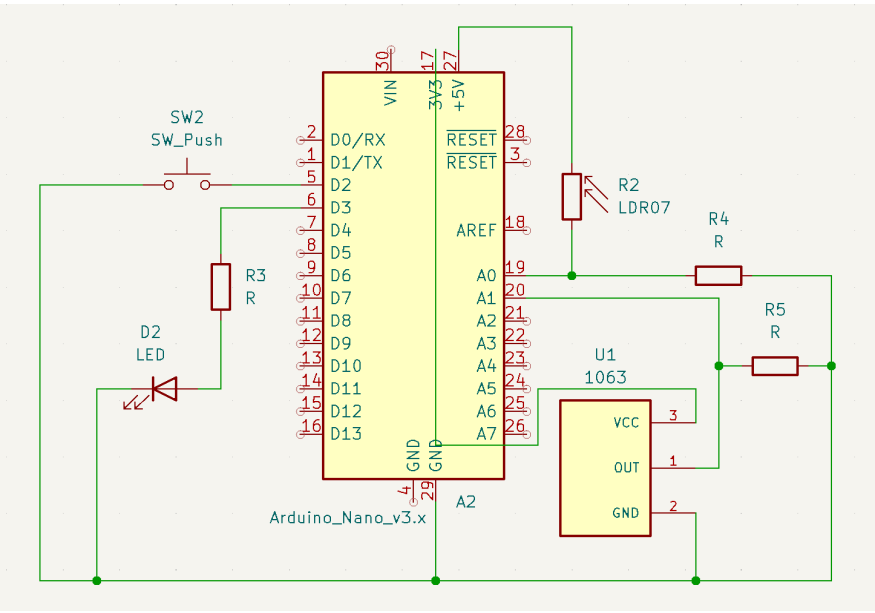


Figure 81: In addition to the aforementioned symbol names, the microphone is represented by U1 1063.

7.3 Binary Code Communication

We have chosen to communicate between the two different Arduino Nanos with binary code. This is because we can represent any character with a binary code, while a digital signal is easy to transmit with our transmitting LED and buzzer. With binary code, each character sent will be made up of a byte, or 8 bits. In this way, any typed message in the serial monitor of the Arduino IDE will be transmitted or received between the two boards. This communication can be conducted with our written Arduino IDE scripts of trans.ino and rec.ino.

7.3.1 Light Fidelity

In the case of light communication, we will turn the transmitting LED on every time the bit of a character is equal to 1 and turn it off when it is equal to 0. There will be a short period of delay between each bit, and there will be a long period of delay between each byte, or equivalent to one character. On the receiving end of our device, the LDR reads values depending on the amount of light from the surroundings in analog. When the transmitting light is on, the LDR reads a relatively high value, and this means that the light received can be digitized into 0s and 1s based on a certain threshold. Of course, the LDR detecting the light must have matching period delays as the light transmitter between each bit to correctly sample the light for each character. In addition, the Arduino code implements a start bit to separate each character in the message for the LDR, which is initiated when it first detects an amount of light below the threshold.

7.3.2 Sound Fidelity

In sound communication, a bit of 0 will be transmitted via a short beep produced by a buzzer while a "1" is represented by a longer beep. There is a short period of silence between each bit and a longer period for each byte, or equivalent to one character. The microphone on the receiver board will pick up these beeps and decode them into binary code based on the length of the sound it detects.

7.4 The Prototyped Project

7.4.1 Implementing Sound Detection

The original protoboard provided by our mentor Mr. Unglaub was capable of transmitting messages with light in the manner described in Section 7.3.1. To improve on Mr. Unglaub's work, we were tasked to implement a sound modality such that a message could be transmitted with a buzzer and received with a microphone. While we initially planned to use two different frequencies to differentiate between 0s and 1s, we struggled to accurately decode transmissions because the microphone response changed based on volume, and not directly on frequency. Moreover, the measured response on the microphone was extremely noisy, making it difficult to differentiate between sound and silence. On this line of thought, we considered the way Morse code transmission worked. If it was not possible to accurately measure frequency, then if we could remove the noise in the microphone to be able to accurately differentiate sound and silence, then it would be possible to send dots and dashes based on duration and silence to represent symbol or character gaps. After creating a working version with Morse logic (morse_with_buzzer.ino and morse_interpreter.ino), we implemented a binary version to create Version 1 that functions as described in Section 7.3.2, where sounds that last for a $dotDuration$ represent zeros and sounds that last for a $dashDuration$ represent ones.

7.4.2 Version 1

Version 1 is capable of sending messages via light or sound, but not both at the same time. In order to properly receive messages, the user must manually set the light and sound thresholds such that the receiver is able to accurately determine what is "bright" or "sound" (a one) and what is "dark" or "silence" (a zero). Sound transmission in this version is limited to 200ms for zeros/dots and 600ms for ones/dashes because the parameters, such as $bitGap$ and $charGap$, have not been properly adjusted in a way that allows the transmitter to send the message as fast as possible while maintaining accuracy.

The transmitter and receiver both have a button. When these buttons are not pressed, the transmitter sends the message via light, and the receiver "listens" for a light transmission. When these buttons are pressed, the transmitter sends the message via sound, and the receiver listens for a sound transmission.

The receiver end of Version 1 has four main functions: $pickUpSound()$, $senseLight()$, $decodeBinary()$, and $getLightByte()$.

The function $pickUpSound()$ returns a boolean and will read the voltage at the micPin every $sampleIntervalMs$ for a duration of $durationMs$, both of which are user-changeable variables declared at the top of the Arduino sketch. If

there is a loud sound, the voltage reading will increase. To determine if this change in voltage is just ambient noise or a sound emitted from the buzzer, the function calculates the $peakAmplitude$ (absolute value of the greatest voltage change during sampling) and compares it to $soundThreshold$. The function returns true if $peakAmplitude$ is greater than $soundThreshold$ and false otherwise.

The function $senseLight()$ returns a boolean and will read the voltage at the ldrPin at that instance. The reading increases when the transmitter LED is on and decreases when it is turned off. If the reading is above $lightThreshold$, the function returns true, and false otherwise.

The function $getLightByte()$ returns a char and is called whenever the receiver detects the start bit from the transmitter. It delays for $1.5 * period$ before calling $senseLight()$ 8 times with intervals of 1 $period$, building the received character bit by bit. The reason for delaying $1.5 * period$ is so that the receiver can read the ldrPin in the middle of each bit, when the transmitter LED is at its brightest.

The function $decodeBinary()$ is a void function that is called whenever the receiver detects a $charGap$ period of silence. The function reads an array of 8 bits that were transmitted through sound and prints the decoded character into the Serial monitor while simultaneously clearing the binary array so that a new character can be received.

7.4.3 Version 2

Version 2 implements automatic thresholding so that the receiver can autonomously adapt to varying ambient light or noise conditions. The light threshold is calculated by taking the average of the voltage values (brightness) read at the ldrPin over a 2 second period at start-up and adding a small buffer. The sound threshold is calculated by finding the $peakAmplitude$ (loudest sound) over the same 2 second period and adding a small buffer.

7.4.4 Version 3

Version 3 adjusts the sound transmission parameters by decreasing $dotDuration$, $dashDuration$, $charGap$, and $bitGap$ as much as possible while still maintaining reliable decoding on the receiver end. In this version, $dotDuration$ is set to 10ms, $dashDuration$ is set to $3 * dotDuration$, and $lightPeriod$ is set to 3ms. Additionally, button functionality has changed such that pressing the button activates light transmission mode, and letting the button go activates sound transmission mode.

7.4.5 Version 4

Version 4 implements a mechanic called stable silence. At lower $durationMs$, the $pickUpSound()$ function executes faster and as a result sometimes interprets dashes as dots because the receiver detects the tone as "on-off-on-off." This is a result of the fact that sound waves are sinusoidal, and so the volume will be near-zero at certain instances. In previous versions, the receiver believed that the instant it detected silence meant that the tone has ended, but that might not necessarily be the case now that $pickUpSound()$ executes faster.

In order to remedy this issue and prevent dashes (1s) from being interpreted as dots (0s), the receiver will check for a period of constant silence rather than an instance of silence to decide whether a tone has ended. This allows us to lower the $dotDuration$ down as low as 1ms after we convert all timing functions to $micros()$ instead of $millis()$. This is ten times as fast as Version 3 sound transmission.

7.5 OR Gate With Light and Sound

In previous versions, the transmitter and receiver were only capable of sending and receiving messages in one mode. In the following versions, we combine both modalities in order to create an OR Gate so that whenever one channel is blocked or noisy, the message is still received.

7.5.1 Fuse 1

Fuse 1 was being developed by Gary while Samuel created Version 4, so Fuse 1 builds off the structure of Version 3. As a result of this, Fuse 1 does not have the same speed capabilities as Version 4. In this version, a button is no longer required to switch modes because both modes are active at once. There is also a new implementation where typing the letter "l" in the receiver side prints the light-dominant message. This means that for each character in the received message, it will first check the light character. If it is a printable character (ASCII 32 to 126, American Standard Code for Information Interchange), that will be the character shown in the output. If it is not printable, then it checks to see if the sound character is printable. If neither are printable, then that character will not be shown in the output. Typing "s" will print the sound-dominant message, which does the same thing as "l" but by checking to see if the sound character is

printable first. The final command is "c", which clears both the light string and sound string so a new message can be received.

On the transmitter end, the sound transmission code was restructured so that instead of having a function that sends the whole message at once in beeps, it sends characters one at a time. This is what allows us to achieve sensor fusion as light characters and sound characters can now be transmitted simultaneously.

7.5.2 Fuse 2

Fuse 2 builds on Fuse 1 by adding the stable silence mechanic that was achieved in Version 4. As a result, $dotDuration$ has dropped from 10ms to 750μs.

7.5.3 Fuse 3

Fuse 2 was meant to be the final version, but one key question proposed by Professor A. F. J. Levi pushed for further development: what if we wanted to send multiple bits at once? The immediate thought was to simply have the buzzer emit different frequencies to represent different series of bits. For example, with 4 different frequencies, it would be possible to send 2 bits at once, that is, 00, 01, 10, and 11. Although coding the buzzer to emit different frequencies with the built-in $tone()$ function was trivial, actually measuring the frequency using our GY-MAX4466 microphone was a different story.

Both Gary and Samuel explored a plethora of frequency-measuring techniques from Fast Fourier Transform to Goertzel's Algorithm. However, such techniques were much too slow to send messages efficiently, effectively making multi-bit symbols pointless. However, by analyzing the amplitude output of the $pickUpSound()$ function on the Serial Plotter, Gary found that it was possible to count how many times the voltage reading crossed a certain threshold and to take the duration divided by the number of crossings in order to calculate a rough estimate of the frequency. This calculation is a form of measuring frequency through zero crossings.

With this technique, the receiver was able to accurately estimate frequencies with a margin of error of about $200-300$Hz. Unfortunately, the GikFun EK2146 buzzer on the board was limited in its frequency range, seemingly unable to achieve frequencies higher than roughly 3kHz. This limits our transmission speed, as higher frequencies would allow the receiver to spend less time calculating frequency. The receiver needs at least one period to determine frequency, and higher frequencies mean more periods in the same period of time. A frequency of 100Hz, for instance, would have a period of approximately 10ms. Taking these factors into account, Gary and Samuel decided to create 16 symbols by combining 2 different durations and 8 different frequencies, allowing for 4 bits to be sent per packet of information. Unfortunately, such parameters make this version's sound transmission slower than the previous one. However, Gary and Samuel still decided to move forward with such symbols in order to create a proof of concept demonstrating symbol creation that would theoretically be faster had higher frequencies been possible. The $dotDuration$ for Fuse 2 is 5ms.

Also included in this new version is the ability to send repeated messages continuously, allowing for the user to transmit them for a variable amount of times, until the user types "z" into the Serial monitor, which is a newly implemented stop command.

7.6 Three-Channel Communication and Error Detection

During Gary and Samuel's presentation of Fuse 3, Professor A. F. J. Levi proposed yet another question: given two different messages, one received by light and the other by sound, such as "12345" and "12378", how is the user supposed to determine which transmission to interpret as the intended message?

With this guiding question in mind, Gary and Samuel dived into three-channel communication and the vast world of error detection.

7.6.1 Comb 1

Fuse 3 to Comb 1 sees the return of the 1-bit sound transmission mode to allow 3 channels to send messages at once. If, for whatever reason, light transmission is blocked and frequency detection is unreliable, 1-bit sound transmissions will still allow the receiver to pick up a message. This addition is a form of redundancy that grants resilience whenever other systems fail.

Comb 1 also implements simple, yet robust, error-checking methods, described below.

Lastly, Comb 1 has updated displays with three strings for each modality: $displayString$, $realString$, and $trashString$. The display string is to store the supposed message received via either light or sound, and it in-

cludes all characters that passed the error checks. The real string includes all characters, including the ones that did not pass the error checks. Finally, the trash string includes all characters that failed the error checks, meaning they are characters the receiver knows for certain were not intended by the transmitter. This allows the user to see exactly which characters in the real string were not transmitted correctly.

Parity Bits

The ASCII numbers that represent printable characters number from 32 to 126. A character byte, however, includes 8 bits. To represent numbers from 32 to 126, we actually only need 7 bits. This means that the final bit, the most significant bit, can be used as something called a parity bit. If a bit string contains an even number of 1s, then it has even parity. If it has an odd number of 1s, then it has odd parity. To properly use a parity bit, choose even or odd. The transmitter and receiver must be the same parity. In this example, let us use even. The bit string in ASCII for the character "a" is 01100001. Since there is an odd number of 1s, we flip the parity bit to a 1, leaving us with 11100001. Now, when the receiver sees this transmission, it will see that the bit string has even parity, and decoding proceeds as normal. However, if one bit flips during transmission, and the receiver instead sees 11100101, it will see that it received a character with an odd number of 1s, and since the transmitter should be sending characters with even parity, something must have gone wrong during transmission. In other words, this one single parity bit allows us to detect a myriad of possible errors and identify corrupted transmissions.

However, parity bits are not foolproof. The receiver would indeed be able to identify a corrupted character if an odd number of bits are flipped. But if an even number of bits in a transmission were flipped, then the parity of the received byte would still be even, and thus no error would be detected. This is the fatal flaw of parity bits.

Checksums

In an effort to remedy the errors that slip through the cracks of parity bits, Gary and Samuel implemented another error detection method called a checksum. In essence, a checksum is a number calculated from a larger message. This number is then appended to the end of the transmitted message and sent along with it. Then, on the receiver side, another checksum is calculated with the same algorithm used on the transmitter side using the received message. If this receiver's checksum calculation does not match the checksum that came along with the transmission, then that means an error must have occurred during transmission.

The checksum used in Comb 1 takes the second bit (arbitrarily) of each character in a message and adds them together. It also takes the sixth bit (also arbitrary) of each character and sums them. Then, it divides both numbers by 12 and multiplies the remainders together. Finally, this calculated number is sent as a character byte appended to the transmission message, coming right after a stop byte. This stop byte essentially tells the receiver that the message has ended, and the very next byte it receives is the checksum byte.

7.6.2 Comb 2

After implementing error detection measures, Gary and Samuel wanted to take it one step further and develop some form of error correction. In Comb 2, they introduced a method of encoding known as the Hamming code.

Hamming Code

In the 1940s, a man named Richard Hamming grew frustrated working at Bell Labs when he left machines running calculations over the weekend only to find them paused early in their operation due to a detected error. As a result of this frustration, he invented the world's first error correction code, now known as the Hamming code.

Imagine a grid of three rows and four columns, indexed from 0 to 3 in the first row so on until 11 in the last row. We will use indexes 1, 2, 4, and 8 as parity bits, and the rest as data bits. If you look at the numbers at each index besides the parity bits in order, you will see that this grid spells out 01100001. Red bits represent parity, blue represent data.

0	1	0	1
1	1	0	0
1	0	0	1

If index 1 represents the parity of columns 2 and 4, and index 2 represents the parity of columns 3 and 4, then if any bit in one of these columns was flipped, it would be possible to track down the exact column in which it happened. Let us assume we are using even parity. For instance, if index 7 was flipped from 0 to 1, then the parity of columns 2 and 4 would be odd. Similarly, the parity of columns 3 and 4 would be odd. Therefore, the error must have happened in

column 4. We can apply similar logic to the rows, where index 4 represents the parity of row 2 and index 8 represents the parity row 4. In other words, adding 4 parity allows us to identify exactly which bit to flip to correct an error.

0	1	0	1
1	1	0	1
1	0	0	1

However, there is one special case that these four parity bits cannot inform us of, and that is if a bit error occurred at index 0, or if no error occurred at all. To combat this, we simply remove the first data bit and only retain 7 bits. And thus, we have created what is known as Hamming $(11, 7)$ code, meaning 11 total bits and 7 data bits, which is robust enough to detect and correct any single-bit error.

	1	0	1
1	1	0	1
1	0	0	1

However, if we bring back that bit we just removed as a parity bit, one that represents the overall parity of the entire block, we get something called an extended Hamming $(12, 7)$. Incorporating this extra parity bit allows us to detect but not correct double-bit errors. This is because the overall parity of the block would be correct, but the parity of the columns and rows would not check out, meaning a 2+ bit error occurred. Extended Hamming codes cannot reliably detect 3-bit errors because 3-bit errors can sometimes be interpreted as 1-bit errors. With that, we have achieved SECDED code, which SECDED stands for single-error correcting and double-error detecting.

1	1	0	1
1	1	0	1
1	0	0	1

To stack upon that, Gary and Samuel have also embedded commands to switch to manual mode (with the command "manual) and to auto mode (with the command "auto"). This user-friendly interface allows for a seamless switch between each mode without having to reset the respective Arduinos. The final change included is a "d" command that clears all buffers, including all real, display, and trash strings.

7.6.3 Comb 3

The unfortunate part about error checking is that it requires us to send more redundant information, thus increasing the transmission time. Gary and Samuel believed they had reached the speed limit in Fuse 2 with their current hardware. However, Samuel discovered one key detail about Arduinos: prescalers.

Arduino Nanos have an internal clock speed of 16MHz, which refers to the rate at which the microcontroller completes instructions. When reading the voltage at an analog pin, it takes an Arduino Nano's ADC (Analog-to- Digital Converter) 13 clock cycles to convert an analog signal to a digital number. The ADC runs at a lower speed than the clock speed to increase the accuracy of its readings, and this rate is calculated by taking the clock speed and dividing by a prescaler. The default prescaler set on the Arduino Nano is 128, meaning the ADC clock is 125kHz, which translates to an analog-to-digital conversion time of about 104μs. Having tested the actual conversion time through code that continuously runs $analogRead()$, Samuel found it took about 112μs per read. By decreasing the prescaler from 128 to 8, we can increase the ADC clock to 2MHz, which Samuel found to lower the read time to 8μs.

The drawback of such high speeds is lower accuracy voltage readings, but for our purpose of detecting sound and flashing LEDs, it is perfectly acceptable.

In Comb 3, Gary and Samuel managed to lower $dotDuration$ to only 100μs, a speed that more than makes up for the extra bits from Hamming encoding. The light period has also decreased to 2ms, and while this number could go lower, Gary and Samuel decided to keep it high to maintain a stable and high-level of accuracy. $fourDotDuration$ also decreased to 2000μs, as faster voltage readings allowed for more precise frequency readings, allowing us to use a smaller range of frequencies such as $1500 - 3000$Hz instead of $300 - 2600$Hz. As mentioned previously, the inability of the buzzer to emit higher frequencies limits the transmission speed of the 4-bit sound symbol mode.

7.7 Final Project Farewell

This final project concludes the journey that Gary and Samuel took at USC under the guidance of Walter Unglaub and Professor A.F.J. Levi. Now that Gary and Samuel have a working device in their hands to allow for such sensor fusion, only time will tell as to what will happen to this entirely new concept.