**ECE 5250**

Talking Clock

12/4/2018

Group 5

Lead - Benjamin Kueffler

Rafi Meguerdijian

Eugene Chiu

Raymond Mak

Note About Contributions

The project was split up by each individual doing various parts of the programming, hardware, and integration :

|  |  |
| --- | --- |
| Contributor | Section |
| Ben Kueffler | Main runtime sequence with ISRs & vocal trigger algorithm. HW implementation. |
| Rafi Meguerdijian | HW/SW Integration |
| Eugene Chiu | Software (Time to MP3) + Sound byte acquisition |
| Raymond Mak | Software (Time to MP3) + Sound byte acquisition |

**Table of Contents**

[**1. Introduction**](#_6x7uck8lv92l) **4**

[1.1. Application](#_oi0iiwsttixm) 4

[**2. Bill of Materials**](#_7tkmcolo8trf) **5**

[**3. Dependences**](#_9ollhp6r7ypc) **5**

[**4. Hardware**](#_sk5s3s3q2os) **6**

[4.1. Microphone](#_waxu6urtyh17) 6

[4.2. Seven Segment Display](#_txghegvlx5uv) 6

[4.3. Speaker](#_1evou4yu12mw) 7

[4.4. MP3 Player](#_83tgt34vss3f) 7

[4.5. User Interface Buttons](#_1z7t7p191eml) 7

[4.6. Product and Design](#_179zxtsfq3j0) 9

[4.7. Voice Rejector High Pass Filter](#_defna3v2rjei) 10

[**5. Software**](#_yl7t9kffjqno) **12**

[5.1. Interrupt Scheme and Display Update](#_6u6jxla1x02r) 12

[5.2. EEPROM Storage & Saver](#_la1dfxk1el1q) 15

[5.3. Clap Detector and Voice Rejector](#_2b6j9caum1au) 16

[5.4. MP3 Player](#_921za1iuzrf7) 19

[5.4.1. English (Military)](#_g0nwdqsr9jbj) 20

[5.4.2. English (AM/PM)](#_k0wld26wjsfo) 22

[5.4.3. Mandarin](#_7qdsjt33n8d) 23

[**6. Repository Information**](#_6q4q66elmgwc) **27**

[**7. Conclusion**](#_brc7ywt84um2) **27**

[**8. Reference**](#_pg6b9brg6ikz) **27**

# **1. Introduction**

Communication is a needed feature in having knowledge and being aware of our immediate and distant situations. Ease of availability of the time is also necessary in may work and business scenarios. The talking clock is intended to assist in this area, one can know the time by simply clapping and the time would be announced to the audience.

## 1.1. Application

The applications of our device can vary to fun little treats of motivation for someone, after all how cool is it that your clock can output multiple languages. The application can also be used in corporate environments; suppose you have group of construction workers and each division has a different primary spoken language, the talking clock will be the signal to inform workers of what time it is(say lunch or day’s work is over). Environments with diverse linguistic frameworks will also be able to make use our product by allowing people to know what time it is.

This could even be applied in more minor settings at home.Suppose couple wants an alarm for a grandparent to be orally reminded what time it is in order to feed their child, but the grand parent does not speak any common language of the grandchild. Worse yet, the grand parent often forgets to check the time to see if it is eating time(or whatever else is timed i.e sleep, medication) for the child. The talking clock will be an oral reminder to inform both the grandparent and the child the time orally. Thus informing both the persons, in a way that each may understand, though no way of communicating with each other. It is precisely why this product, we felt was a great idea. This product can also be helpful for blind persons being told what the time is by the talking clock.

# **2. Bill of Materials**

|  |  |  |
| --- | --- | --- |
| **Item** | **Price** | **Reference** |
| Gikfun 1.5” 4Ohm 3W Speaker | $4.00 | <https://www.amazon.com/gp/product/B01LN8ONG4/ref=oh_aui_detailpage_o04_s02?ie=UTF8&psc=1> |
| HiLetgo 7-Segment Display | $1.00 | <https://www.ebay.com/p/4-Bits-Tm1637-Digital-Tube-LED-Clock-Display-Module-for-Arduino-Due-UNO-2560-R3/704539444?iid=201415081217&chn=ps> |
| Longrunner DFPlayer MP3 Player | $8.00 | <https://www.amazon.com/gp/product/B01MXOFAE4/ref=oh_aui_detailpage_o04_s01?ie=UTF8&psc=1> |
| Aideepen Microphone w/ Amplifier | $2.80 | <https://www.amazon.com/gp/product/B07BBNBT83/ref=oh_aui_detailpage_o00_s00?ie=UTF8&psc=1> |
| Arduino/ATmega328 | $5.00 | Various vendors |
| **Total** | **$22.80** |  |

# **3. Dependences**

Open Source External Libraries Used:

|  |  |
| --- | --- |
| Name | Description / Usage |
| SoftwareSerial | Software serial library to send software UART cmds to MP3 player |
| TimerOne | Allows easy utilization of hardware timer on ATmega chips |
| EEPROM | Allows easy utilization of internal EEPROM on ATmega chips |
| DFRobotDFPlayerMini | Contains functions for commands over UART to MP3 Player |
| TM1637Display | Contains functions for commands to 7 Segment LED display |

In addition to these libraries, internal libraries were made, so that certain functions, constants, and declarations could be located separately from the main code.

|  |  |
| --- | --- |
| Name | Description / Usage |
| timetomp3 | Contains the function necessary to convert the time into audio commands to the MP3 player in multiple languages |
| talking\_clock | Contains the constants, dependences, pinouts, and functions utilized in the main code |

# **4. Hardware**

## 4.1. Microphone

The microphone we chose to use was the Aideepen Microphone with an Amplifier. The board came with a 20-20KHz electret microphone soldered on while using the MAX4466 op-amp. The amplifier has an excellent power supply noise rejection. The information and image were from [1]. 

The microphone is used in the design in order to read the noise level. The noise level is read by taking a rolling average and adding to it every time the processor samples. This prevents single noise sources from being accepted and interpreted as a clap. The microphone output is directly hooked up as the input to a high-pass filter, described later, in order to reject human voice frequencies Figure 1) Aideepen Microphone with an Amplifier

## 4.2. Seven Segment Display

The seven segment display used in this device is the HiLetgo 7-Segment Display. The driver IC is TM1637. The 4 digits are just what we needed to display the current time. The 4 pins are GND for ground, VCC for the power supply, DIO for the digital input and output pins, and CLK for the clock signal pin. All 4 pins are used for this project.

Figure 2) 7-Segment Display

## 4.3. Speaker

The speaker used was the Gikfun 1.5” 4Ohm 3W Speaker, the ordered 2pcs 1.5" 4Ohm 3W Full Range Audio Speaker Stereo Woofer Loudspeaker with a characteristic sensitivity of loudspeaker ranging from 83±3dB. The resonance frequency ranging 320 Hz±20%. Insulation Resistance 10MΩ with nominal impedance of 4Ω±15% and input-power rating : 2W. The rated frequency range will vary from 0-20KHz. Lastly, the distortion factor will be a maximum 5% [2]. This speaker is used in conjunction with the amplifier from the MP3 player in order to give loud readings to the user.

Figure 3) Gikfun 1.5” 4Ohm 3W Speaker

## 4.4. MP3 Player

The DFPlayer Mini MP3 Player for Arduino was chosen for its small size and low price. The supported sampling rates (kHz) are 8/11.025/12/16/22.05/24/32/44.1/48. Audio data is sorted by folder; 100 folders can be supported and each folder can hold 255 songs. There are 30 volume levels as well as a 6-level adjustable equalizer. The speaker can be directly connected to this module.

For this project, the MP3 player was used to store the sound bytes for the various languages. Using a particular file naming structure, the code would call out the files that needed to be played according to the data structure given to the timeToMp3 function.

Figure 4): DFPlayer Mini MP3 Player

## 4.5. User Interface Buttons

The user interface consists of an ON/OFF button alongside three push buttons. The push buttons are utilized in order to set the hour and minute. The final push button sets the mode. The mode can be changed to be *English Standard*, *English Military Time,* or *Mandarin*. Each of the interfacing buttons are debounce in software to prevent unintended inputs into the system. The user buttons are interrupt based as described in the software section.

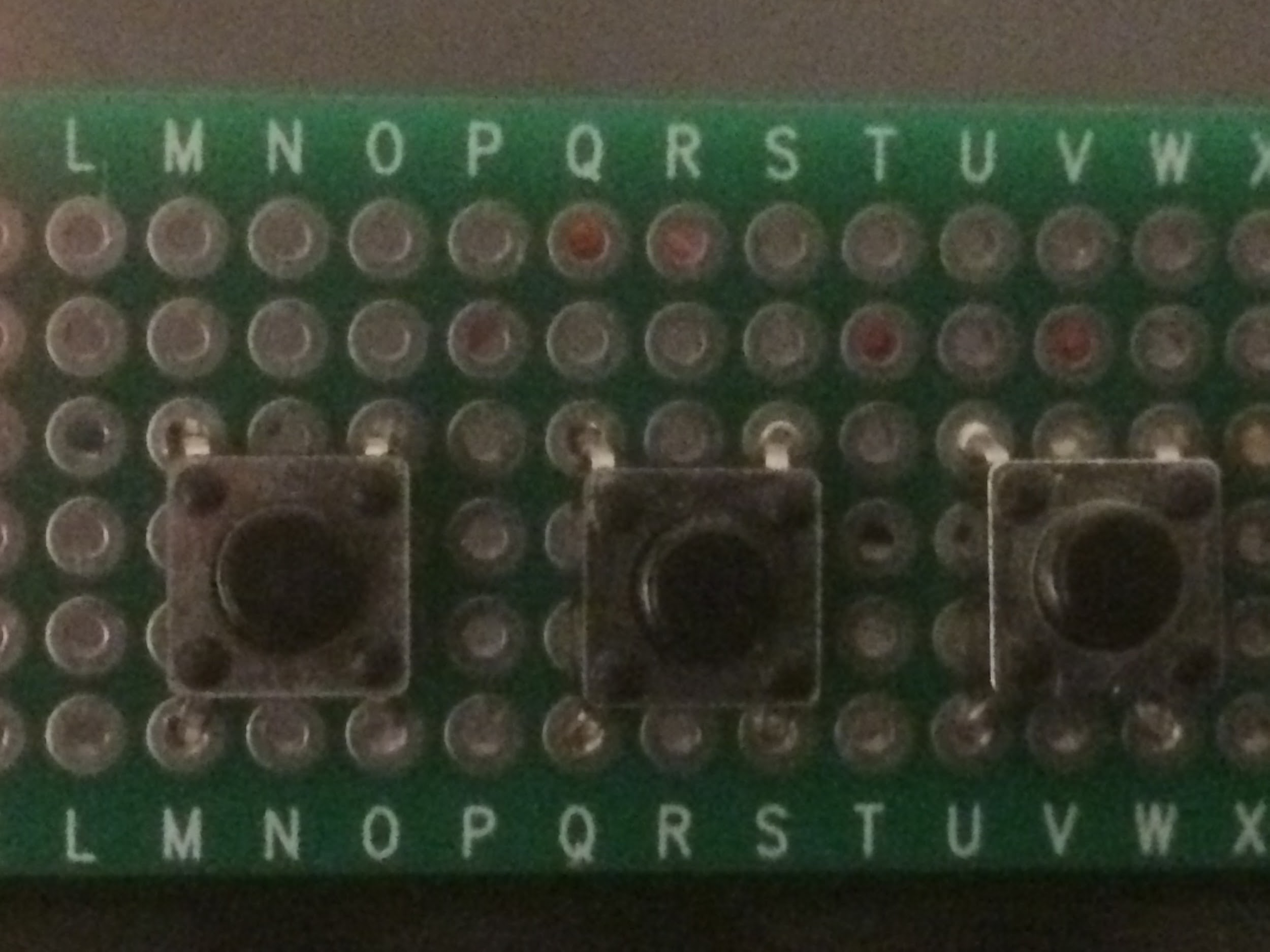


Figure 5) Custom Button PCB (Hr, Min, Mode)



Figure 6) Left: Interface buttons. Right: Power ON/OFF

## 4.6. Product and Design



Figure 7) Design as integrated on the breadboard

The hardware was integrated together on a breadboard as depicted in the figure. A few discrete components are shown that were not discussed prior. The pull down resistors are connected to the buttons in order to not short the design when a button is pressed. Other resistors exist on the UART line connected to the MP3 player, this allows for reflections on the UART line to be damped so that they do not cause accidental edges of the signal.

This breadboard design was put into a box in order to provide an enclosure which allowed for convenient access to the user buttons and provided a clean ascetic that would be desired in the final product.

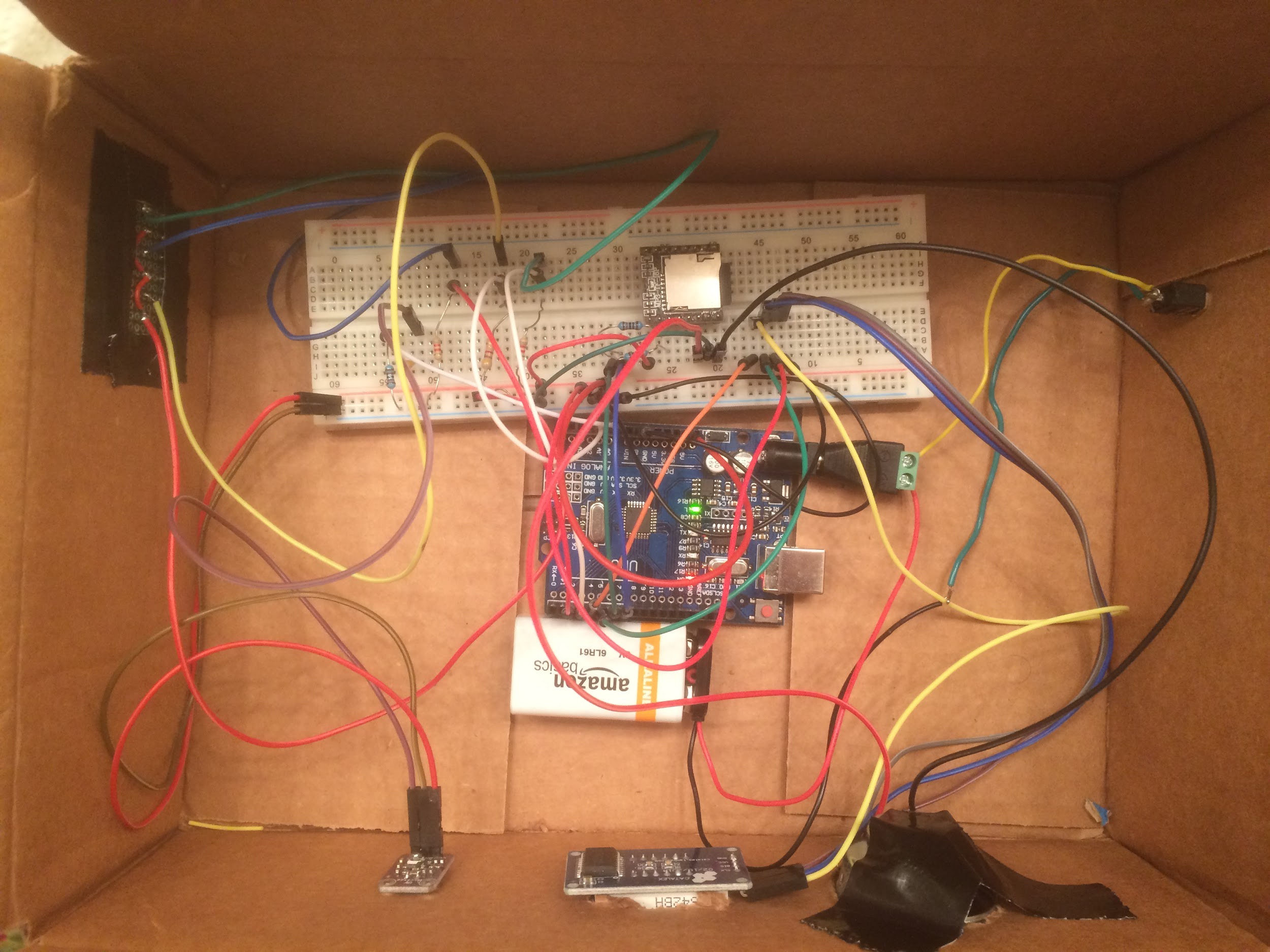
Figure 8) The inside of the box after all components are integrated



Figure 9) The front of the box after power on.

## 4.7. Voice Rejector High Pass Filter

In between the microphone output and the analog input on the ATmega328 is a high pass filter. To understand the importance of this high pass filter, it’s necessary to analyze the frequency spectrum of a human clap against the human voice.

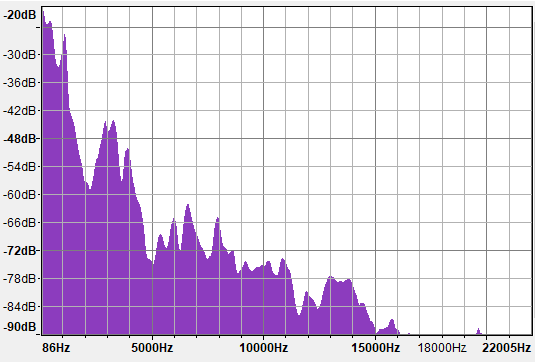


Figure 10 : Human voice saying “Hello world” recorded in Audacity

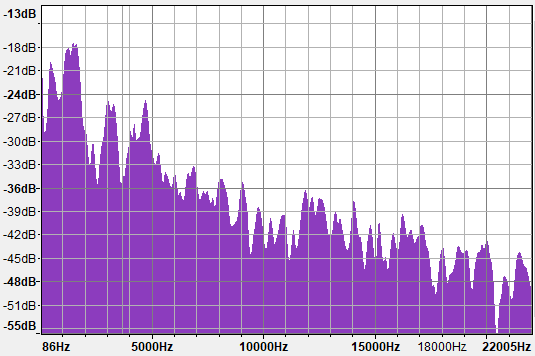


Figure 11 : Flat handed clap recorded in Audacity

The human voice is typically between 60Hz-150Hz for normal speech, and around a few thousand Hertz for human singing. This type of frequency is narrowband, it encompases a relatively small band of the frequency spectrum. Differing from this, the human clap seems to peak at around 1000-2000 Hz, but can be seen affecting the entire audible spectrum. This makes the human clap a broadband signal. Although there is plenty of noise in these figures, due to the low and narrow frequency of the human voice, adding a high pass filter can be used to reject some human voices. Note that this approach is not as effective as using the time-domain properties of the voice and clap, as is demonstrated later.

For implementing a high pass filter in hardware, a simple circuit comprised of a 100nF capacitor in series with a 680 Ohm resistor in parallel. This creates a high pass filter with a -3db cutoff at 2.34kHz.

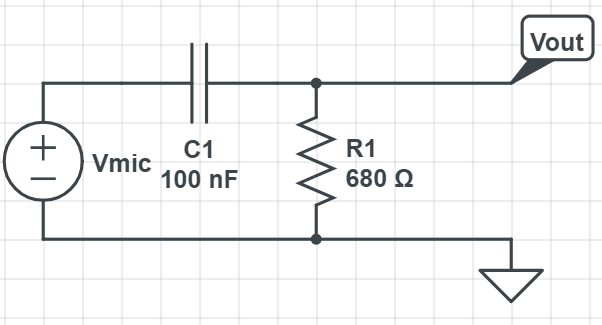


Figure 12: High Pass Filter connected to microphone output

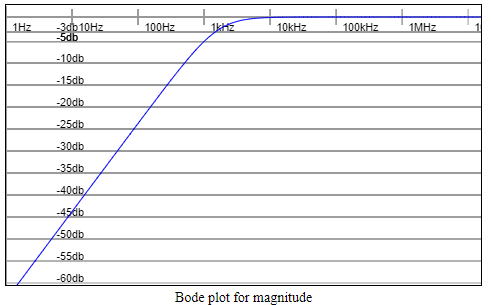


Figure 13: Bode Plot of high pass filter (C = 100nF, R = 680 Ohms)

# **5. Software**

## 5.1. Interrupt Scheme and Display Update

In order to update the time being displayed, a data structure must be kept and properly updated through various means. There exists three different methods of loading the current time. The first method is on power on start up, the time is stored in EEPROM from previous runs and then is loaded one time on start up. The method of continually storing the known time in EEPROM is discussed in latter sections. The second and most commonly utilized method of getting the time is through the timing interrupt. The TimingISR() is run after an internal interrupt triggers every 500 ms. This allows the internal time variables to update no matter where the program is at. Every assigned variable in this ISR was designed to not conflict with other parts of the code, so that re-entry into the main loop does not cause issues.

|  |
| --- |
| // The interrupt service routine for keeping time. This does not update the display by itself. void TimingISR() {  // During every interrupt, increment the subsecond, this is the smallest resolution of our clock  subsecond ++;  Update = true;  if(subsecond == 1000000/kTimeResolution){  second ++;  if(second == 60)  {  minute ++;  if(minute == 60)  {  hour ++;  if(hour == 24) hour = 0;  minute = 0;  }  second = 0;  }  subsecond = 0;   } } |

The display update routine was purposefully not put into this interrupt service routine due to its long period of operation. Interfacing to the 7-segment LED display takes a large amount of time, so it’s not desired operation to keep this functionality in an interrupt. Instead, the time variables are updated, and during the main loop, the display shall be updated as long as the *Update* variable is true. This allows a fast ISR, and puts off the low speed serial write to the display until control returns to the main loop.

|  |
| --- |
| void loop() {   // At the top of the loop, update the LED segments if the subseconds have changed  // If a valid input is detected, the LED will not update, but the time will continue counting, so subseconds will not be lost  if(Update)  {  // If Hours/Minutes have changed, erase and write to the EEPROM  updateEepromTime();   // Update display with current hour and minute  TimeUpdate();    #if defined(TEST\_MODE)  Serial.print(String(time\_disp[0]));  Serial.print(String(time\_disp[1]));  Serial.print(':');  Serial.print(String(time\_disp[2]));  Serial.println(String(time\_disp[3]));  #endif  } |

The start of the main loop is shown here, the EEPROM will be updated and time variables will be written to the display at this point if our TimingISR() routine has recently run.

The final method where the time could be updated is if the user wants to set the time via using the configuration buttons. Both the minute and hour setting buttons are based on the ATmega328’s two dedicated pin interrupts. This allows for the minute and hour to be set, regardless of where the main program is in the loop. The mode configuration button is not a true interrupt, but instead, the user holds the mode button and views the selected mode on the LED screen. Due to the limited number of available interrupts, and the fact that it made sense to have the mode button as a “hold” button, the mode was chosen to not be interrupt based. Each of these three buttons is debounced, otherwise, the software would see multiple switch toggles when only one was intended. The debounce interval is set via a compile time parameter kDebounceInterval, and is set to 200ms.

|  |
| --- |
| // Updates the minute by one every time the minute button is pushed void changeMinISR() {  static unsigned long last\_interrupt\_time = 0;  if (millis() - last\_interrupt\_time > kDebounceDelay){  minute ++;  if(minute == 60) minute = 0;  // Wait for the button to be depressed  while(digitalRead(btn\_config\_min) == 1);  }   last\_interrupt\_time = millis(); } |

## 5.2. EEPROM Storage & Saver

In order to allow for quick resets or power surges to not interrupt the clock, it’s a desired feature to be able to recall the last time displayed after shut down. This feature is available using the internal EEPROM on the ATmega328 device. Basic operation of the EEPROM is simple, during startup, the EEPROM can be read to recall time and mode. During operation, every time the display updates, the EEPROM can also update with the current time and mode.

However, the EEPROM has limited erase cycles, on the order of 100,000 erases. This presents a problem, because the *minutes* slot in the EEPROM memory will be updated every minute on average. This means that after just 69 days of operation the minutes may fail to be stored. To combat this problem, an EEPROM saving algorithm was developed for this project.

The solution to this problem is to occasionally move the location of the time data within the EEPROM, this extends the life of a single EEPROM byte significantly, because the damage is done evenly over the entire 1kB EEPROM rather than over a single byte. However, using this method requires the processing routine to know where the valid address is located! The EEPROM saver defines a new memory location on the EEPROM, labeled as *kTimePointerAddr*, this 2 byte location is static and keeps the value of where the *hours* address is located. The location of this pointer is updated every twelve hours in order to keep the damage done to the EEPROM constant across all addresses. The *kModeAddr* was also chosen to have a static address, because it only changes when the user holds a button, and 100,000 cycles is unlikely to ever occur on this address.

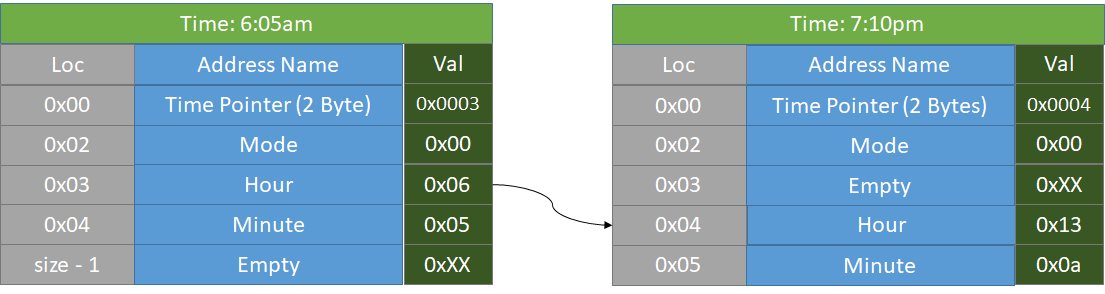


Figure 14) Example of EEPROM locations during 13 hour operation

|  |
| --- |
| // Time (hour, minute) are written to the EEPROM as permanent memory when this function is called // Update Count is an incrementing counter that counts to kStAddrUpdateTime seconds // After that time, the pointer to the Time (hour, minute) will increment by one to prevent EEPROM degredation void updateEepromTime() {  // Enter this statement every update\_count seconds  if ((kTimeResolution \* update\_count++) / 1000 == kStAddrUpdateTime)  {  // The starting address must be incremented every so often so that no particular byte of the EEPROM  // degrades to the point of value  // Protect Special Address 0, if we're about to roll over, start at address 0x01  // instead of 0x00. Otherwise, increment address by one  time\_addr = (time\_addr >= kEepromSize - 1) ? kTimeAddr : time\_addr + 1;   // Write the pointer to the time\_disp data into special address 0x00  EEPROM.put(kTimePointerAddr, time\_addr);  update\_count = 0;  }  // If the hours or minutes in the EEPROM memory do not match our current  // hours/minutes, burn the new hours/minutes, otherwise no write/erase will occur  EEPROM.update(time\_addr, hour);  EEPROM.update(time\_addr + sizeof(hour), minute); } |

## 5.3. Clap Detector and Voice Rejector

In order to understand how to implement a software solution for detecting human clapping, it’s important to understand the theory and challenges that this involves. In order to isolate clap detection via software methods, some time-domain analysis of clapping needs to be taken into account. More importantly, analysis of the human voice needs to be taken into account, since the main cause of issues is misdetection due to human voice interference.

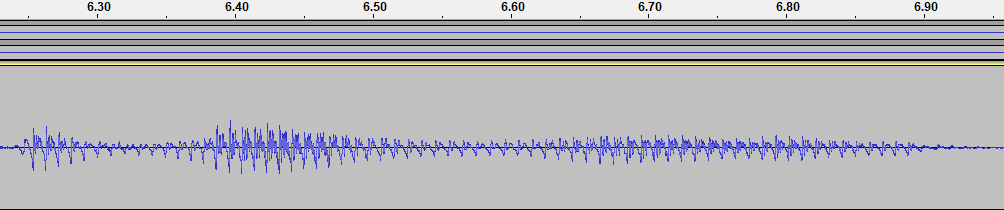


Figure 15) Human Voice saying “Hello World” as recorded in Audacity

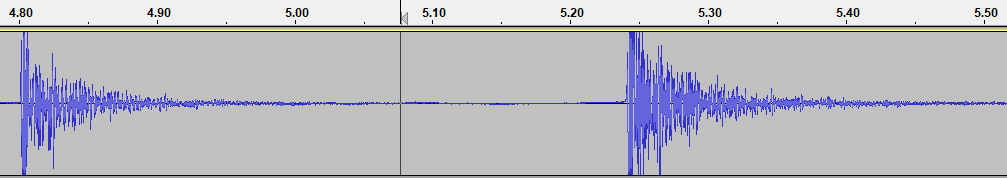


Figure 16) Human claps as recorded in Audacity

Based on the following time domain results, it can observed that a human clap maintains a high amplitude for a much shorter time (25-50ms) than the human voice (100-500ms). This is the main method that claps can be differentiated from the human voice. Other methods were utilized, such as exploiting the low frequency nature of the human voice, but this remains the single most effective method of eliminating issues with the clap detector.

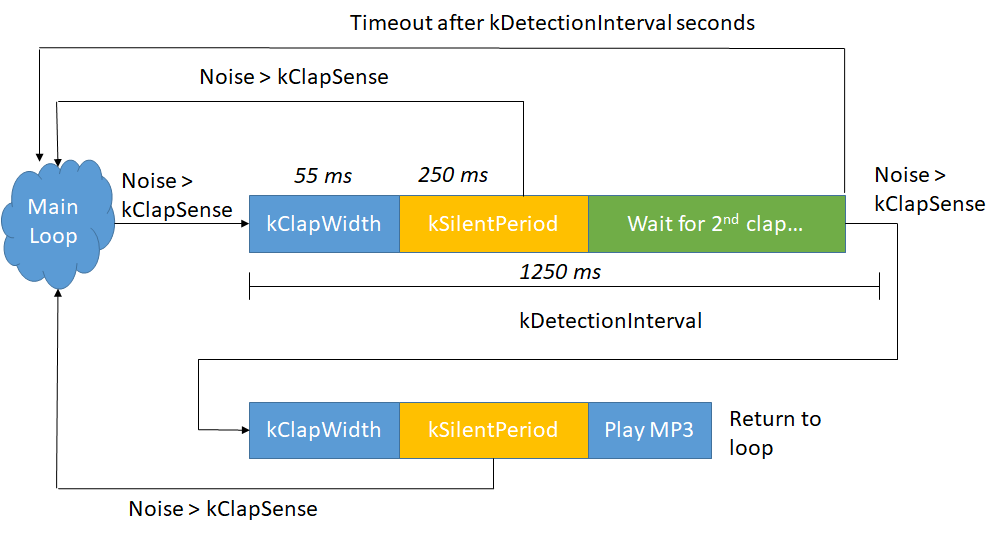


Figure 17) Block diagram depiction of clap detection algorithm

The clap detection algorithm keeps a variable known as *rolling\_average*, which records the noise of the last *kNumReadsInAverage* samples of the microphone. This number is represented by Noise in the figure. The purpose of a rolling average is to eliminate spikes of noise, as well as get an accurate RMS value of the waveform. This noise will be an input into our algorithm, whenever the rolling average noise exceeds the *kClapSense* value chosen at compile time, the algorithm begins. At this point, there is a delay of *kClapWidth*, this is put into place in order to give a buffer of space equivalent to the width in time of a human clap. This buffer is necessary due to the next phase, the silent period, where our algorithm expects no noise to be detected. The *kSilentPeriod* time, which is set to be around 250ms, allows the algorithm to reject human voices and other noises. The theory behind this is that, after the width of a clap, there should be a short period of time before the second clap can be heard. This period of time should be at least 250ms, if there is still continuous noise during this period, then the noise source must be something other than a clap.

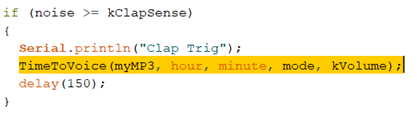
If the algorithm does not detect sound during that silent period, then it will wait for a second clap to be received. This waiting time is given by the constant *kDetectionInterval*, and is set at compile time to be 1250ms. If a second clap is not received in this amount of time, then the algorithm will exit and go back into the main loop. If a clap is detected, then the clap width will be observed, and the silent period will take place again. If the silent period is respected on the 2nd clap, then the time will be sent to the timeToMp3 function in order to announce the time of day.

The following code is the implementation of the algorithm described, it’s location is within the main loop of the source code.

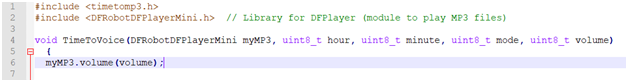
|  |
| --- |
| // Calculate a rolling average of the microphone inputs, we generally want the number  // of elements computed in this average to be about 5 ms worth of stuff  rolling\_average = (analogRead(microphone) + rolling\_average\*(kNumReadsInAverage - 1)) / kNumReadsInAverage;    if (rolling\_average > kClapSense)  {  Serial.println("Noise > Sense");  do  {  uint32\_t last\_period = millis();  while (millis() - last\_period < kClapWidth)  {  // This while loop runs during the width of the clap, during this time, the rolling average should be recorded  // It is expected that by the time we exit this loop, the rolling average should be a very low value (No sound should be recorded)  rolling\_average = (analogRead(microphone) + rolling\_average\*(kNumReadsInAverage - 1)) / kNumReadsInAverage;  }   while (millis() - last\_period < kSilentPeriod)  {  // This loop expects to hear very little noise, if it does, then the noise is probably not a clap (may be a human voice)  rolling\_average = (analogRead(microphone) + rolling\_average\*(kNumReadsInAverage - 1)) / kNumReadsInAverage;  if (rolling\_average > kClapSense)  {  clap = 0;  rolling\_average = 0; // Stop from entering the loop again immediately, this should give a few milliseconds of buffer  break;  }  else if (clap < 2)  {  clap = 1;  }   }  if (clap == 2)  {  //play mp3  TimeToVoice(myMP3, hour, minute, mode, kVolume);  clap = 0;  break;  }  if (clap == 1)  {  Serial.println("Clap 1 reached");  while (millis() - last\_period < kDetectionInterval)  {  // This loop waits for the second clap  clap = 0;  rolling\_average = (analogRead(microphone) + rolling\_average\*(kNumReadsInAverage - 1)) / kNumReadsInAverage;  if (rolling\_average > kClapSense)  {  // Second clap detected within proper interval  clap = 2;  break;  }  }  }  } while (clap > 0);  } |

## 5.4. MP3 Player

The code for the MP3 player was written as a separate function that is called in the main loop. The function is named “TimeToVoice” and takes 5 inputs of the MP3 playing object named “myMP3”, the hours named “hour”, the minutes named “minute”, the mode for English (military)/ English (AM/PM) / Mandarin named “mode”, and the volume level named “kVolume”. The picture below highlights exactly where the function is called.



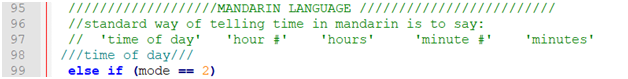
At the beginning of the function, the volume is set.



After which we go into 1 of the 3 cases of English (military), English (AM/PM), and Mandarin.







The sound clips are played by calling the name of the mp3 files. This specific module requires the files to be named in the format of “XXXX” where X is any digit. For example, 59 is “0059”. We have the sound files for 0-59, AM(0100), PM(0101), oh(0102), hundred(0103), and hours(0104) for English

### 5.4.1. English (Military)

Military time has many cases that one who was not familiar with it would not know about. We coded this mode with two major steps. The first is the hours. If time is 0000, Zero Zero is read. If the time is 00\_ \_ (where blanks represent non-zero numbers in the hour slots from here on), only Zero is read. If the time is 0X \_ \_ (where Xs represent non-zero numbers in the minute slots from here on), Zero and X are read. If the time is XX \_ \_, whatever number greater than 9 is read as is. For example, 11 \_ \_ is just read as eleven. The next step is the minutes. If the time is \_ \_ 00, Hundred is read. If the time is \_ \_ 0X, Zero and X are read. If the time is \_ \_ XX, whatever 2 digit number is read normally just like in the case of hours. Lastly, at the end of every time read the word “Hours” must be read. A block diagram and the code can be seen below.

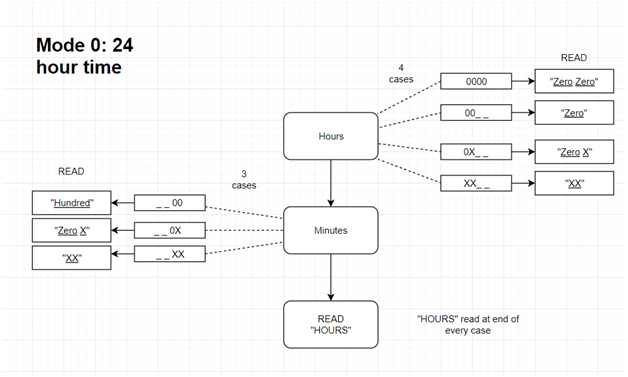
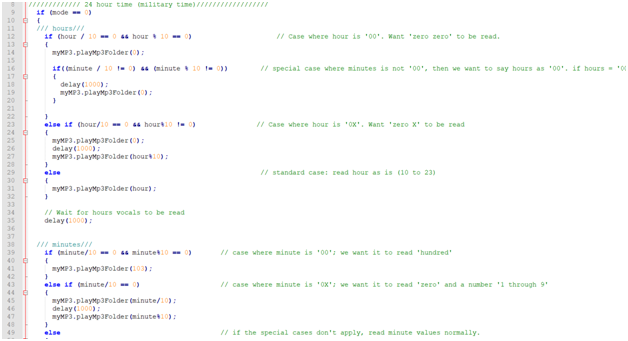
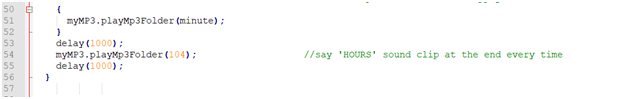


Figure 18) Flow diagram of military time





### 5.4.2. English (AM/PM)

The AM/PM mode has 5 major steps but there are less cases. First a variable is set to determine whether or not it is the morning or afternoon. Next the hour variable is changed from 24 hour format to 12 hour format if need be. So if the hour is 0 it is changed to 12 and if the hour is 13 or greater, it is changed to 12 as well. Otherwise it is left as is. The hours are then read as is and we move to the minutes. There are only 3 cases for the minutes. The first is if the minutes is 10 or greater. In this case the minutes can be read as is. If the minutes is less than 10 and not 0, we have the word “oh” be read and then the minutes can be read. For example, 1009 is read as ten oh nine. The last case is if the minutes is equal to 0. In this case nothing is read. Lastly we check the variable set in the beginning and read AM or PM based on that variable. The block diagram and code can be seen below.

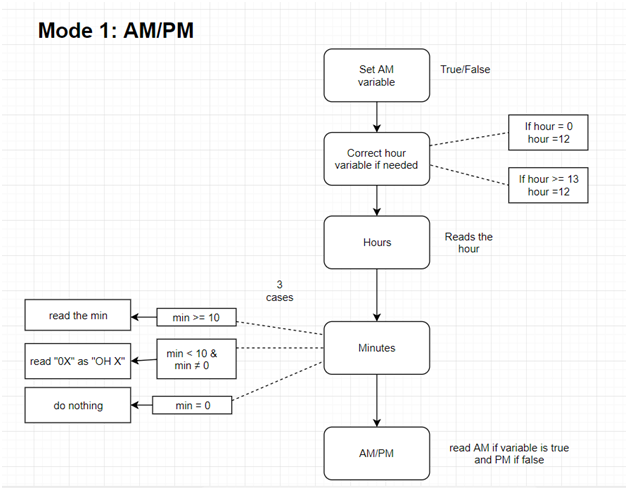
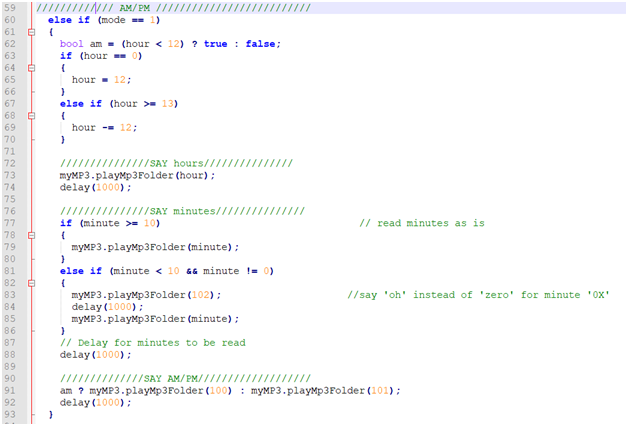


Figure 19) Flow diagram for English standard time



### 5.4.3. Mandarin

Telling time in mandarin (applies to MOST cases):

* say one of the 8 phrases to depict time of day
* say a number (1-12) to depict hour
* say the word hour (dian)
* say a number (0-59) to depict minutes
* say the word minutes (fen)

That’s the general way to read time aloud in mandarin. When you say the numbers, you say them as if you were counting normally in mandarin. The challenge when implementing it in a program are the many exceptions within the language and when implementing it into code.

To count from 0 to 10 there is exactly one chinese character to represent each number. When you get to the teen numbers, you would say ‘shi’ to depict 10 and then the number in the ones digit. For example, you would say thirteen as ‘shi san’ or ‘ten three.’ You DO NOT say ‘yi shi san’ or ‘one ten three’

For all other numbers 20 and above, you would start with the number in the 10s digit, read 10, then read the number in the 1’s digit. For example 23, would be read as ‘er shi san’ or ‘two ten three’ and 39 would ‘san shi jiu’ be read as ‘three ten nine’

When translating it into code, we used one sound file each for numbers 1 to 10 then used programming logic to mix and match those numbers to get bigger numbers. Along with that there are language exceptions in Mandarin when reading time. For example, when you read 2 o’clock, the 2 is read differently from a regular 2. You say ‘liang’ instead of the standard way of counting which is ‘er.’ Another exception is in the minutes. Just like how we say 1:09 as “one oh nine,” in Mandarin it’s read similarly. For 1:09 in Mandarin, you say “yi dian ling jiu fen” or “ one hour zero nine minutes” to translate each character literally. When you combine the rules from the language and the programming logic to read the sound files, you end up with a lot of cases as you can see in the tree diagram.

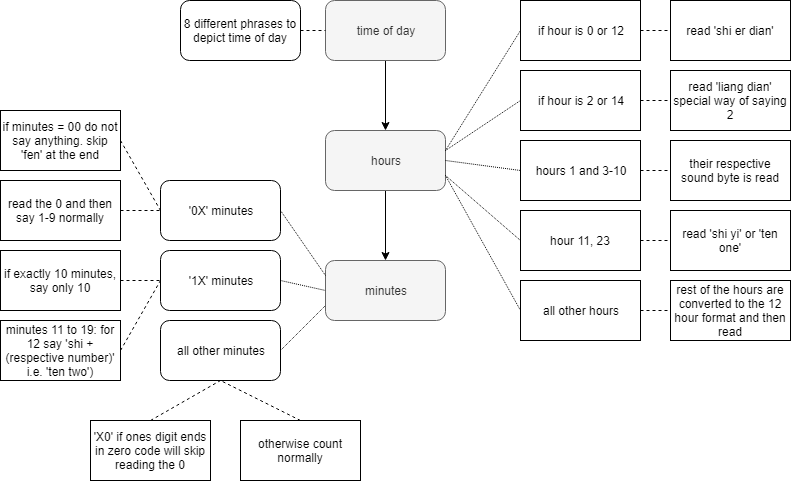
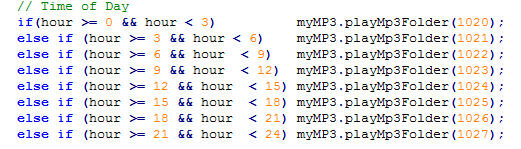


Figure 20) Mandarin code tree diagram

Time of day

Instead of the AM/PM system we have, in mandarin time of day is denoted by several phrases. We use 8 different phrases in our code, but in the Mandarin language these are not strict definitions. People use these loosely just to denote a general time of day. It would be most similar to how in English we say ‘early morning’ or ‘late afternoon.’ The phrase ‘early morning’ does not associate with a strict time period. We just divided it in intervals of 3 hours just to approximate.



Mandarin code for the Time of day(left); labels for the sound file(right)

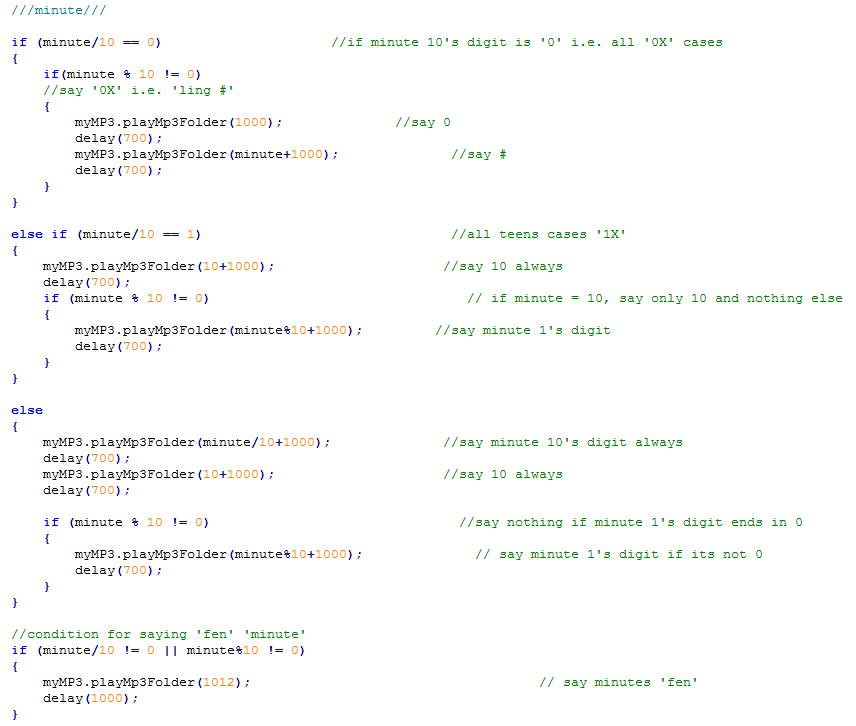
Hours

To explain all the hours cases, we have to go in order. The first special case is at 0 hour or midnight. We want it to read it as 12 so we just combine it with the 12 hour case. The next special case is the 2 case we talked about earlier. We have to put aside a special sound file to read this special version of 2. This takes care of all the special language cases, but not we have to consider the cases where some numbers only require one sound file to be read and some numbers require two sound files. The former applies to numbers 1, 3-10 and the latter applies to all other numbers that do not have a special case.

Mandarin code for reading hours

Minutes

Counting the minutes has even more special cases than the hours since the numbers go from 0 to 59. These cases can be broken down into three cases and each of these parts have their own subcases. The three main cases are single digit numbers, 2-digit numbers less than 20, and numbers 20 and higher. The reason for these cases is that each of these cases have similar ordered sound files to be played. We can choose to omit or add sound files to read aloud the right number. In the first case, when reading single digit numbers, we want it to read the exact sound file associated with the number, except for the number 0 we don’t want anything to be read at all. In the second case, when reading the teen numbers we generally want to read two sound files: one sound file to read 10 and one sound file to read the one’s digit number. The exception to this case is when reading the number 10. For 10, we skip reading the one’s digit by using a subcase. The third case applies to numbers 20 and higher. Most of these numbers require 3 sound files to be read: first, the 10’s digit, then the number 10, and the 1’s digit. The exception to this rule is when there is a 0 in the 1’s digit, so 20, 30, 40, 50. For those numbers, we read regularly and then there’s extra logic to determine if we want to read the 1’s digit or not.



Mandarin code for reading minutes

# **6. Repository Information**

A repository was used in order to track changes over time. Additionally, it allows others to reference our design for their own implementations.

|  |  |
| --- | --- |
| Github | [Talking-clock](https://github.com/bkueffle/Talking-clock) |

# **7. Conclusion**

One can appreciate how such a simple idea requires so much work and devotion. Despite the display, being in cardboard box does not look to be the most interesting, it most certainly has the potential of becoming a very exciting idea and product for industry and personal consumers to use. It equally amazing to see that such cheap products can make such a device. This echos the phrase “knowledge is power” for had we not known programing and embedded systems, such device would only be true merely in the abstract but with knowledge of engineering and embedded systems, we know have shown we can actually develop such devices our very selves. Arduino has the best open source prototyping platform for such products.

# 8. Reference

[1]https://www.amazon.com/gp/product/B07BBNBT83/ref=oh\_aui\_detailpage\_o00\_s00?ie=UTF8&psc=1

[2]https://www.amazon.com/gp/product/B01LN8ONG4/ref=oh\_aui\_detailpage\_o04\_s02?ie=UTF8&psc=1