Term Project

DTMF Control System

Author:	Caleb Begly
Course Name/Number:	CS370
Due Date:	Wednesday, May 4, 2016

Introduction:

Dual Tone Multi Frequency Signaling (DTMF) is a system that was historically used to signal telephone equipment. More commonly known to the general public as “touch tone” [1].

Although DTMF tones are becoming more obsolete as traditional dialing systems are switched to digital input and softphones, these tones are commonly still used to control amateur radio (“Ham Radio”) equipment at remote sites. The idea is that the tones are low bandwidth, and can easily be sent from most commodity handheld radios. This allows people to call and configure radio repeaters and other automatic equipment remotely, even over great distances.

One specific application is a phone patch (also known as an autopatch) that provides a way to make phone calls by tying a radio to a phone line with a DTMF controller. dialing, these tones can be used to send information back to a control system. It works by using a 4x4 matrix of distinct tones to produce a total of 16 possible combinations (0-9, *, #, and A-D)

These can be used for certain types of communication needed during natural disasters. A phone patch setup can allow people to call out and can also allow emergency responders to keep in contact with outside support groups.

However, the downside of many of these existing phone patch systems is that they have a high cost, hard to find components, and they are often stuck at a repeater site because they need a physical phone line, and they use a lot of power. This paper describes how to create a portable DTMF Control system using the raspberry pi, and demonstrates how it can be used to control a phone patch. The overall flow is shown in Figure 1.

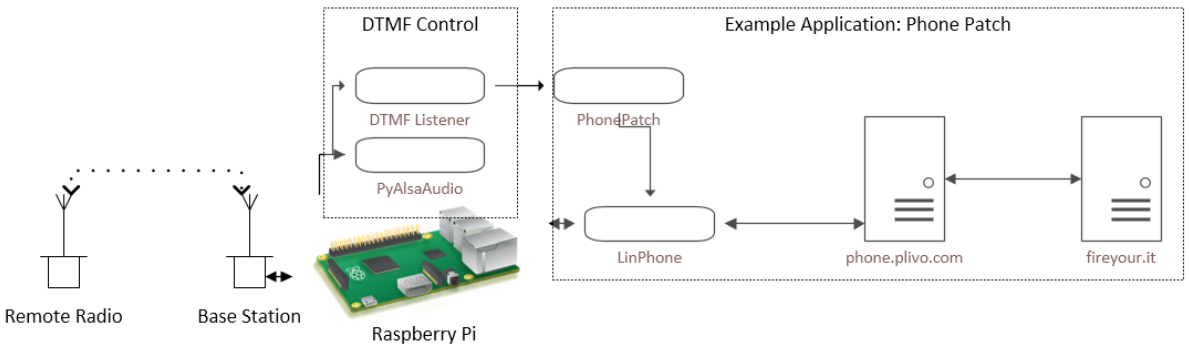

Figure 1: Design for DTMF Control and Phone Patch

Motivation:

DTMF control systems can be used for a variety of applications, including remote control, system interfaces, control for things like phone patches, and other general UI needs. The need to have a better, more accessible DTMF control system stems from several concerns including cost, accessibility, openness of parts, and power efficiency/portability. While there are commercial products or projects available, they usually fail to meet at least one of these three criteria.

The commercial offerings have some of the highest cost. For example, the popular low-[2]. Kits (which are much cheaper) can range upwards of $10 retail for the basic components, but many of them are not manufactured anymore so they go for much more on used item sites and at ham fests (for example, the PC-1A for about $55 used[3]). Due to limited availability of used parts, and because parts eventually wear out, this is not a viable option for the future.

Several guides on building kit DTMF decoders for a phone patch recommend using a discrete decoder chip like the Zarlink MT8870[4], or one of the National Semiconductor or TI chips. Some even recommed a combination of filters built from discrete components[5]. However, many of the discrete decoder chips are not manufactured anymore (presumably because the functions can be performed in software now in commercial applications), and the discrete component option may be difficult for people with little or no analog circuit design skills, as well as for people without good soldering skills.

Another concern is the openness of the hardware products. The Open Source Hardware Association (OSHWA) defines open source hardware as "hardware whose design is made publicly available so that anyone can study, modify, distribute, make, and sell the design or hardware based on that design". OSHWA goes on the clarify that, "Ideally, open source hardware uses readily-available components and materials, standard processes, open infrastructure, unrestricted content, and open-source design tools to maximize the ability of individuals to make and use hardware"[6]. While the complex discrete component circuits use open hardware (as they use basic components), they are not at accessibly to hobbyists due to the technical details. The custom DTMF ICs are in many cases proprietary so they don't meet the requirements to be considered open hardware. Soldering surface mounted devices with dozens of pins can be difficult for hand soldering. So overall, a design that can be easily

budget offering (MFJ-624E) retails for $159.95

hand assembled from readily available COTS (consumer off-the-shelf) parts would be useful and more futureproof.

Finally, there comes the concern of power and portability. The MFJ-624E is an example of a standard commercial product that takes a decent amount of power to run because it is expected to operate in a fixed base station. In addition, they are not as portable because they are bulky and have additional weight. Due to the scarcity of exact specification for many of these products (many are old and not manufactured anymore), it is difficult to get a good idea of exactly how bad their power consumption is. Also for portability, all of the phone patches that I was able to find use a standard phone line connection, so they can only be deployed in an area with equipment to support a standard phone line.

All of these criteria have influenced the decision to come up with a solution that is cheaper, easier to assemble, more portable, and more flexible.

Project Characteristics and Methodology

The system that we design for DTMF control needs to be capable of reading DTMF tones and detecting the characters they represent. In addition, it needs to be comfortable with varying rates of the tones, and it should have decent noise tolerance.

After it reads a tone from an audio frame, it needs to be able to take sequences of these tones and use them to detect commands and data entry. This requires a state machine that is capable of keeping track of what mode the system is in, and storing information as needed. Finally, the state machine needs to be able to interact with a system that it needs to control. In our case, this is the phone patch hardware or software. It needs to be able to create and terminate a connection based on the DTMF control codes that it receives.

The interface to the module should be able to be swapped out so it can be used with different radios. For this project, we focused on

the Kenwood style connector which is a popular connector, but the design could be extended to any style of connector.

In addition to the overall flow, the design needs to use parts that are readily available. They should be parts that can be purchased or scavenged easily, and should be capable of assembly on a simply breadboard. The total cost of the components should be low, roughly $50.

Finally, the power consumption needs to be such that it can be run off of emergency or battery power. Unlike units that are made to be in a server room, these components should be able to be tuned to reduce power consumption.

Proposed Solution

For this project, we start with the Raspberry Pi 2 for the microcontroller. There are several features that make it a good choice. First, it is readily available, and many hobbyists already have one. Although there are other boards that would be better for a single use case, the Pi runs Raspbian[7] (a variant of Debian Linux) which means it can support a much larger base of programs easily than a simple embedded system or specialized chip with custom OS could.

The Raspberry Pi 2 ($35) also comes with the ability to support large storage devices, as well as built in HDMI support, and a host of GPIO pins that can be used to programmatically interface with hardware. This increases the flexibility, and allows the designer to use the basic DTMF control proposed in this document to control their project, whether it be hardware or software.

There are other alternatives that could be used. The Raspberry Pi 3 would also be a good choice because it has built in WiFi and Bluetooth [8] which would remove the need to buy a separate wireless dongle. However, the added processing power may not be needed (and comes at a power cost). Depending on the application that is being controlled, a Raspberry Pi Zero could be used to get even lower cost (about $5) and lower power usage[9]. However, for this project, we used the Raspberry Pi 2 because it was available, and it has enough processing power for the application we are interested in.

The audio interface between the PI and the radio consisted of 4 audio stereo jacks, a handful of wires, and a 1uF capacitor. The design was based off of a schematic from the Rhe PeaterPiPhy project[10]. The difference was that we didn't need the PTT control because we used the radio's VOX system instead. Also, we increased the size of the capacitor that blocks the DC current (to save power, and protect the radios) as suggested by the Rhe PeaterPiPhy project[11]. The total cost for all these components was negligible (less than a dollar) so the interface (radio harness) is fairly inexpensive. The interface schematic is shown in Figure 2.

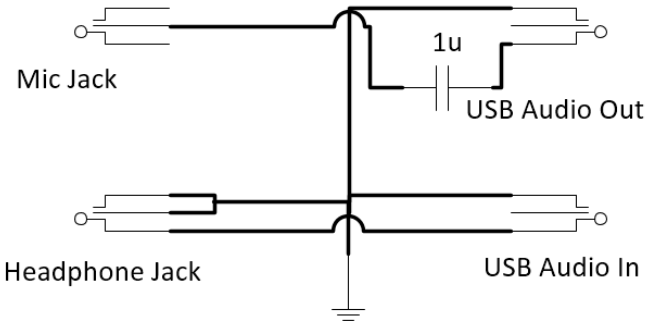

Figure 2: Audio Interface for Kenwood-Style Radio

The Pi was fitted with a wireless dongle ($3), micro SD card ($5), USB sound card ($5), and power adapter ($5). The total off-the-shelf price for the complete project is about $53 retail which comes close to our desired price point of $50. This does not include the cost of the radio, of course, because the user can hook it up to whatever radio they want based on their own needs.

On the software end, for this demonstration we chose to use a VOIP connection for several reasons. First, it is readily available anywhere you can get internet. Having the ability to drop the base station at nearly any

house or business will improve the flexibility of the system in the event of an emergency. We chose to not use a traditional phone interface because there would be more hardware needed, and because traditional phone lines are becoming less common, especially in urban areas.

A survey performed by the CDC on data from 2012 found that 35.8% of respondents didn't have a traditional landline phone, and of the households that had both a wireless and a landline phone, "29.9% received all or almost all calls on the wireless telephones"[12]. They further noted increases in both adults and children that had wireless only service. Since we want to build this for the future, it is better to assume some form of internet connection than to assume a landline phone connection is present.

Software and Libraries

The first part of the software needs to be able to listen on the USB microphone for audio coming in. The radio has its own squelch settings (to ensure it doesn't send anything to the audio out until the receive signal goes above a certain threshold), so we only had to listen continually. Another option the user could do, if their radio supported it, would be to set receive tones on the monitoring radio to prevent the radio from breaking squelch for other traffic. However, this isn't needed in most cases, so it was omitted from this project.

To listen to the audio card, we used the python library PyAlsaAudio. Alsa has low overhead and should allow good performance. PyAlsaAudio is really a python interface written in C for the Alsa Audio API. It was written by a single person to deal with the lack of good Python libraries for audio in Linux[13]. The API is poorly documented, and has stability issues with different types of configurations, especially on the Raspberry Pi (since the API was made for x86 Linux).

The primary reason to use this library was simply because there are not any other better alternatives for Alsa. However the library needs work. The Mixer API does not match the Alsa Audio API Mixer, and it is poorly documented. Critical parts of the API (like period size) do not match the Also Audio API settings, and are not documented at all, resulting in a lot of guess and check work. Ultimately, I ended up getting it working with this, however if I continue this project, I would likely either write my own abstraction module, or switch to Pulse Audio, even though it has higher overhead.

For the DTMF decoding, I used an excellent Python module that John Etherton wrote for DTMF detection from Nokia PyS60 raw wav files[14]. The algorithm uses the Goertzel Algorithm[15] to compute the Discrete Fourier Transform (DFT). After computing this, the library checks the energy content (from Pareseval's Theorem)[16] to determine if a frequency is "present" in a given frame. It then uses a small state machine to combine the frames, and separate out the distinct keypresses.

However, the module was built to operate on a prerecorded file, so the state machine used future values in its processing – something we can't do if we are streaming. To fix this, I began by modifying the module to take in its values from a stream of frames (similar to how the Alsa audio module would provide it). The data then had to be processed into the state machine, and the state machine had to be modified to not use any future data (while still having the same level of error correction). Finally, I updated the interface to allow binding a listener to the event when a character is read, and I updated the state machine to fire off the event whenever it is sure a character was read successfully.

This made the programming of the rest of the application much easier because, after testing this thoroughly with prerecorded audio streams (generated by AudioCheck's DTMF Tone Generator[17]), I was able to code the rest of the control logic based solely off these events. At this point, the DTMF controller is really done, because all that is left for the programmer to do is write a listener for their specific application.

For the phone patch, I first came up with the simple control sequence below:

- User enters *11# if they want to connect to the phone patch
- User can then enter the number they want to dial, followed by the pound key
- Finally, the user can enter *22# to disconnect and terminate the call.

This control sequence was realized by the state diagram shown in Figure 3. As the diagram shows, it starts in state S0, and moves to S4 only after it has successfully read *11# (if it detects any other numbers or patterns here, it resets to S0). In S4, it reads in the key presses as the telephone number until the user presses #. Once they press pound the call is connected (more on that later) and then it starts waiting for the "hangup" sequence of *22#. After it receives the hangup sequence, it hangs up the phone and returns to the reset state S0. The listener that is used in the code to listen for each of the key presses follows the same flow.

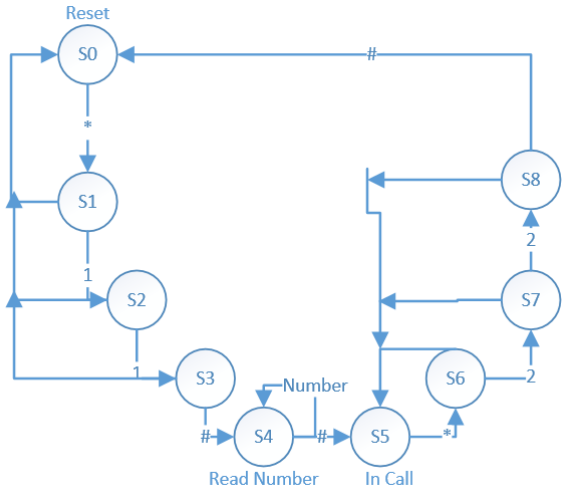

Figure 3: State Diagram for Phone Patch Control

The last piece of the puzzle is making the phone call. Since we wanted Python to be able to call it with a simple subprocess.Popen command, we needed a SIP (session initiation protocol, the underlying protocol for many VOIP implementations) client that is able to be controlled via command line. The SIP client I chose was Linphone[18] which is an open source SIP client that is easy to install on the Raspberry Pi. The command line feature for this was not documented online, however, the executable's --help command provided enough information to guess how to get it to work.

The SIP client can then be started and stopped by Python to make the phone call and hang up. The only downside of this is that the client ends up registering and unregistering every time a phone call is made. At the time of this writing, there is a better command line interface for Linphone that would allow registering the phone once and then making several calls. However it currently has several bugs that prevent it from working on the Raspberry Pi.

To make the phone call, we need a SIP trunk. A SIP trunk is a service that allows SIP phones to call traditional phone numbers. The SIP trunk acts as a proxy, and usually is controlled by a phone branch exchange (PBX) server. For this application, I chose to use Plivo[19] which is an inexpensive SIP trunk (fractions of a penny per minute for US). They have an easy to use interface, and the setup was smooth once they confirmed my identity. While there are some free SIP trunk providers, they are often rate limited, have poor documentation for their APIs, or they have poor quality/reliability.

For the PBX, I simply modified the direct dial PHP application[20] that Plivo provided and hosted it on my own website, fireyour.it. The code in the GitHub repository was out of date, so I had to hunt through their API docs and update the code so it would run properly. The server URL (called the response url) on the PBX takes a query string and produces XML to send back to the SIP trunk that instructs it how to hande the call. This could be commands to play a message, use text to speech, use a vXML phone menu system, or just simply make a phone call. In this case, since we just wanted a basic phone call, we left that portion unmodified.

With all of these pieces together, the system is able to initiate phone calls based on

what the person with the radio dials, and it outputs the audio so it is transmitted back over the radio. Note that since the radios are operating in simplex mode, only one person can talk at a time, and they have to take turns. This is standard with radio communication. However, it would be trivial to make it work in full duplex mode as well by adding a second radio, or using a radio that can operate in full duplex mode.

As far as power goes, the Raspberry Pi can be under clocked and run in low power mode because the audio quality is band limited to 8kHz. The interface to the radio is also low power because the capacitor is added to block the DC current (the Pi USB audio card can also put out more current then the Kenwood standard allows the radio to sink, so the capacitor is needed for safe operation). In ultra-low power situations, a capacitor could also be added to the other audio path, however, those are better matched, so there is no need in most cases.

Conclusions:

This project demonstrated the power and flexibility of the Raspberry Pi. It is capable of real time audio processing, even when working with janky software libraries. The hardware/software interfacing is an added plus of the Pi, and its emphasis on Python has improved the availability of software libraries and modules for the platform. Overall, it is a solid platform, and this project will allow users to build their own DTMF controlled systems on it.

The DTMF controller project also showed that there is a real need for Python to better support audio interfaces. Python has support for the legacy OSS audio standard in its core (ossaudiodev[21]) libraries, however OSS has been deprecated for a while (it also doesn't usually work with USB sound cards), and having a generic interface that could work with W32Audio, ALSA, Pulse Audio, or whatever is available on the system would be really nice.

The ALSA audio subsystem was difficult to work with, and ultimately ended up being an issue because of devices taking exclusive control of the card, even when it was operating in shared mode.

There is another aspect to this project which did not need to be addressed in the initial design, but would need to be addressed in a public release of the project. The FCC rules are pretty specific for communications over the Amateur bands, and specify requirements like alternate remote control (which could be provided by an internet link) and identification (which could simply be playing a sound file periodically). These are simple to address, and they will need to be added before releasing the project to the public.

The DTMF control project provided a good experience working with the integration of software and hardware. It provides a basis for anyone that wants to use DTMF control in their application on-the-cheap. Finally, it added a variant of the DTMF decoder code that operates on streams instead of prerecorded files. This could be useful to other projects once it is released to the public. Overall, this project produced several things that may further the area.

dtmflistener.py

```
"""
Caleb Begly
dtmflistener
Created for CS370
Due 5-4-16
"""
import sys
import time
import getopt
import alsaaudio
import struct
from DtmfDetectorStream import DtmfDetectorStream
import phonePatch

#Code below for basic read modified from documentation example
card = "default"
detector.inp = alsaaudio.PCM(alsaaudio.PCM_CAPTURE,
alsaaudio.PCM_NONBLOCK, card)

# Set attributes: Mono, 8000 Hz, 16 bit little endian samples
detector.inp.setchannels(1)
detector.inp.setrate(8000)
detector.inp.setformat(alsaaudio.PCM_FORMAT_S16_LE)

# The period size controls the internal number of frames per period.
detector.inp.setperiodsize(160)
# End code from documentation

#Include detector
detector = DtmfDetectorStream()
detector.setCharacterListener(phonePatch.phoneListener) #Attach
listener to the stream detector

#Read dtmf tones
while True:
    # Read data from device
    packet, data = detector.inp.read()

    if packet:
        samples = [data[i:i+2] for i in range(0, len(data), 2)] #Break
into samples
     for sample in samples:
         (point,) = struct.unpack("h", sample)
         detector.goertzel(point)
        time.sleep(.001)
```

phonePatch.py

```
"""
Caleb Begly
phonePatch
Created for CS370
Due 5-4-16
"""
import subprocess
import time

def phoneListener(char):
    print "Recieved Character: "+char
    if(phoneListener.state == 0):
        print "State 0"
        if(char == "*"):
            phoneListener.state = 1
    elif(phoneListener.state == 1):
        print "State 1"
        if(char == "1"):
            phoneListener.state = 2
        else:
            phoneListener.state = 0
    elif(phoneListener.state == 2):
        print "State 2"
        if(char == "1"):
            phoneListener.state = 3
        else:
            phoneListener.state = 0
    elif(phoneListener.state == 3):
        print "State 3"
        if(char == "#"):
            phoneListener.numberBuffer = "" #Reset the buffer
            phoneListener.state = 4
        else:
            phoneListener.state = 0
    elif(phoneListener.state == 4): # Read in the number and store it
        print "State 4"
        if(char == "#"): #Done entering
            #Call Number
          print "Calling "+phoneListener.numberBuffer
          phoneListener.connection = subprocess.Popen(['/bin/bash', '-
c', '/usr/bin/linphonec -c /home/pi/.linphonerc -s
'+phoneListener.numberBuffer])
            phoneListener.state = 5
        else:
            phoneListener.numberBuffer += char #add to buffer
    elif(phoneListener.state == 5):
        print "State 5"
        if(char == "*"):
            phoneListener.state = 6
    elif(phoneListener.state == 6):
        print "State 6"
        if(char == "2"):
```

```
            phoneListener.state = 7
        else:
            phoneListener.state = 5
    elif(phoneListener.state == 7):
        print "State 7"
        if(char == "2"):
            phoneListener.state = 8
        else:
            phoneListener.state = 5
    elif(phoneListener.state == 8):
        print "State 8"
        if(char == "#"): #End the phone call
          phoneListener.connection.terminate()
            phoneListener.state = 0
        else:
            phoneListener.state = 5

#Init
phoneListener.state = 0
phoneListener.numberBuffer = ""
phoneListener.connection = None
```

```
###############################################
## last updated: 2/2/2007
##
## This is written by John Etherton - john@johnetherton.com
## http://johnetherton.com/programming/DTMFdetect/
## http://johnetherton.com/projects/pys60-dtmf-detector/
##
##
## Please send me any questions or comments about this code.
## I made this so I could do DTMF detection with python on a
## Nokia Series 60 phone. So that's all it really does. I made it as
## as quickly as I could. This is my first attempt at making a general,
## reusable library for PyS60.
##
## LICENSE STUFF:
## This code is released to the public for any use. I wish I knew
which
## license that is but I don't know. I should probably find out.
##
## This is code is offered as is with out any warranty. If it breaks
## and does something bad, don't sue me. I have tested it to the
## best of my ability, but I have no doubt that it has numerous
## faults
##
## The Goertzel algorithm used in this script was shamelessly stolen
## from this wikipedia entry:
http://en.wikipedia.org/wiki/Goertzel_algorithm
## All the code not given in that entry I wrote myself.
##
## IMPORTANT:
## for this to work with PyS60 1.3.17 I had to install the wave.py
## and chunk.py files from my python 2.2 installation onto
## the phone as libraries.
##
## Usage example:
##
## -Assume we have a single channel 16 bit, 8000hz file
"audioFile.wav"
## -that was recorded on a phone when someone pressed 8,3,3,4,5
## CODE:
##         from DTMFdetector import DTMFdetector
##         dtmf = DTMFdetector()
##         data = dtmf.getDTMFfromWAV("audioFile.wav")
##         print data
## OUTPUT:
##         "83345"
##
## It's that simple
##
## Oh and I can't spell so please forgive all my mistakes
###############################################
```

DtmfDetectorStream.py

```
import chunk
import wave
import struct
import math

################################################
## This class is used to detect DTMF tones in a WAV file
##
## Currently this only works with uncompressed .WAV files
## encoded 16 bits per channel, one channel, at 8000 hertz.
## I know if I tried I could make this so it'd work with
## other sampling rates and such, but this is how PyS60 records
## audio and I've got a bunch of other things to work on right
## now. If you want to add this functionality please go ahead
## and do it
class DtmfDetectorStream(object):


      ###############################################
      ## The constructor
      ##
      ## initializes the instance variables
      ## pre-calculates the coefficients
      def __init__(self):

            #DEFINE SOME CONSTANTS FOR THE
            #GOERTZEL ALGORITHM
            self.MAX_BINS = 8
            self.GOERTZEL_N = 96
            self.SAMPLING_RATE = 8000

            #the frequencies we're looking for
            self.freqs = [697, 770, 852, 941, 1209, 1336, 1477, 1633]

            #the coefficients
            self.coefs = [0, 0, 0, 0, 0, 0, 0, 0]

            self.reset()

            self.calc_coeffs()


      ###########################################
      ## This will reset all the state of the detector
      def reset(self):
            #the index of the current sample being looked at
            self.sample_index = 0

            #the counts of samples we've seen
            self.sample_count = 0

            #first pass
```

```
            self.q1 = [0, 0, 0, 0, 0, 0, 0, 0]

            #second pass
            self.q2 = [0, 0, 0, 0, 0, 0, 0, 0]

            #r values
            self.r = [0, 0, 0, 0, 0, 0, 0, 0]

            #this stores the final string of characters
            #we believe the audio contains
            self.charStr = ""

            self.previousTime = 0.00;
            self.lastChar = "";
            self.currentCount = 0;
            self.glitch = False;

      ##########################################
      ## Post testing for algorithm
      ## figures out what's a valid signal and what's not
      def post_testing(self):
              #this is nothing but a fancy state machine
            #to get a valid key press we need
            MIN_CONSECUTIVE = 2
            #characters in a row
            #with no more than
            MAX_GAP = 0.1000 #changed to 100ms because after test I
found this is the longest "distinct" issue
            #seconds between each consecutive characters
            #otherwise we'll think they've pressed the same
            #key twice
            row = 0
            col = 0
            see_digit = 0
            peak_count = 0
            max_index = 0
            maxval = 0.0
            t = 0
            i = 0
            msg = "none"

            row_col_ascii_codes = [["1", "2", "3", "A"],["4", "5", "6",
"B"],["7", "8", "9", "C"],["*", "0", "#", "D"]]

            #Find the largest in the row group.
            for i in range(4):
                  if self.r[i] > maxval:
                        maxval = self.r[i]
                        row = i

            #Find the largest in the column group.
            col = 4
```

```
            maxval = 0
            for i in range(4,8):
                  if self.r[i] > maxval:
                        maxval = self.r[i]
                        col = i

            #Check for minimum energy
            if self.r[row] < 4.0e5:
                  msg = "energy not enough"
            elif self.r[col] < 4.0e5:
                  msg = "energy not enough"
            else:
                  see_digit = True

                  #Normal twist
                  if self.r[col] > self.r[row]:
                        max_index = col
                        if self.r[row] < (self.r[col] * 0.398):
                              see_digit = False
                  #Reverse twist
                  else:
                        max_index = row
                        if self.r[col] < (self.r[row] * 0.158):
                              see_digit = False


                  #signal to noise test
                  #AT&T states that the noise must be 16dB down from the
signal.
                  # Here we count the number of signals above the
threshold and
                  #there ought to be only two.
                  if self.r[max_index] > 1.0e9:
                        t = self.r[max_index] * 0.158
                  else:
                        t = self.r[max_index] * 0.010

                  peak_count = 0

                  for i in range(8):
                        if self.r[i] > t:
                              peak_count = peak_count + 1
                  if peak_count > 2:
                        see_digit = False
                        #print "peak count is to high: ", peak_count

                  if see_digit:
                        #print row_col_ascii_codes[row][col-4] #for
debugging
                        #stores the character found, and the time in the
file in seconds in which the file was found
```

```
                                #Caleb Begly: The code for the rest of
this function was added (and is based on the cleanup function) to
support the signalling and processing of stream information.
                    currentChar = row_col_ascii_codes[row][col-4]
                    time = float(self.sample_index) /
float(self.SAMPLING_RATE)
                    timeDelta = time - self.previousTime
                    #print "curr char:", currentChar, "time delta:",
timeDelta #for debugging
                    self.previousTime = time #Now this sample is the
previous time

                   #print "curr char:", currentChar, ", last char: ",
self.lastChar, ", time delta:", timeDelta #for debugging

                    #Set first character, if needed
                    if(self.lastChar == ""):
                         self.lastChar = currentChar

                    #check if this is the same char as last time
                    if self.lastChar == currentChar:
                         self.currentCount += 1
                         self.glitch = False
                    else:
                         #some times it seems we'll get a stream of
good input, then some erronous input
                         #will pop-up just once. So what we're gonna
do is peak ahead here and see what
                         #if it goes back to the pattern we're
getting and then decide if we should
                         # let it go, stop th whole thing
                         # Make sure we can look ahead
                         if(self.glitch == True): #We already gave
them a chance
                              self.glitch = False
                              if(self.currentCount >=
MIN_CONSECUTIVE):

     self.signalCharacterRecieved(self.lastChar) #We have moved on to
the next character. Signal the end of the recieve
                              self.lastChar = currentChar
                              self.currentCount = 1
                              return
                         else: #Give it one chance
                            self.glitch = True
                            return

                    #check to see if we have a big enough gap to make
us think we've
                    #got a new key press
                    if timeDelta > MAX_GAP:
```

```
                              #so de we have enough counts for this to be
valid?
                              if (self.currentCount - 1) >=
MIN_CONSECUTIVE:

      self.signalCharacterRecieved(currentChar) #The gap was large
enough, so it is a new keypress
                                    self.currentCount = 1
                              self.lastChar = currentChar


      ##########################################
      ## the Goertzel algorithm
      ## takes in a 16 bit signed sample
      def goertzel(self, sample):
            q0 = 0
            i = 0

            self.sample_count += 1
            self.sample_index += 1

            for i in range(self.MAX_BINS):
                  q0 = self.coefs[i] * self.q1[i] - self.q2[i] + sample
                  self.q2[i] = self.q1[i]
                  self.q1[i] = q0

            if self.sample_count == self.GOERTZEL_N:
                  for i in range(self.MAX_BINS):
                        self.r[i] = (self.q1[i] * self.q1[i]) +
(self.q2[i] * self.q2[i]) - (self.coefs[i] * self.q1[i] * self.q2[i])
                        self.q1[i] = 0
                        self.q2[i] = 0
                  self.post_testing()
                  self.sample_count = 0



      ##########################################
      ## calculate the coefficients ahead of time
      def calc_coeffs(self):
            for n in range(self.MAX_BINS):
                  self.coefs[n] = 2.0 * math.cos(2.0 * math.pi *
self.freqs[n] / self.SAMPLING_RATE)
                  #print "coefs", n, "=", self.coefs[n] #for debugging

      #Caleb Begly: Calls the character listener
      def signalCharacterRecieved(self, character):
            self.characterListener(character)

      #Caleb Begly: Sets the character listener
      def setCharacterListener(self, funct):
            self.characterListener = funct
```

Bibliography

[1] https://en.wikipedia.org/wiki/Dual-tone_multi-frequency_signaling
[2] http://www.mfjenterprises.com/Product.php?productid=MFJ-624E

[3] http://www.ebay.com/itm/KENWOOD-PC-1A-Phone-Patch-Controller-/322089939201?hash=item4afe0e7101:g:Pk4AAOSwFMZWqXa9
[4] http://www.microsemi.com/document-portal/doc_view/126495-msan108-appnote
[5] http://www.hamtronics.com/pdf/AP-3.pdf
[6] http://www.oshwa.org/definition/
[7] https://www.raspbian.org/
[8] https://www.raspberrypi.org/products/raspberry-pi-3-model-b/
[9] https://www.raspberrypi.org/products/pi-zero/
[10] http://www.twotonedetect.net/raspberry-pi-simplex-repeater-the-peaterpipyr/
[11] http://www.twotonedetect.net/raspberry-pi-simplex-repeater-the-peaterpipyr/
[12] https://gigaom2.files.wordpress.com/2012/12/wireless201212.pdf
[13] http://larsimmisch.github.io/pyalsaaudio/pyalsaaudio.html
[14] http://johnetherton.com/projects/pys60-dtmf-detector/
[15] https://en.wikipedia.org/wiki/Goertzel_algorithm
[16] https://en.wikipedia.org/wiki/Parseval%27s_theorem
[17] http://www.audiocheck.net/audiocheck_dtmf.php
[18] http://www.linphone.org/technical-corner/linphone/overview
[19] https://www.plivo.com/
[20] https://github.com/plivo/directdial/tree/master/php
[21] https://docs.python.org/2/library/ossaudiodev.html