Final Project
Christopher Bero
EE384

DTMFE

Dual Tone Multiple Frequency - ASCII Extended

Project Description

This project is the groundwork for taking concepts introduced in EE381 and EE384 to further an extracurricular investigation. MODAC is the name of my pet project to integrate an embedded computer into the peripherals and CAN bus of an old Honda Civic. The system currently lacks remote access, and I'm unwilling to involve a cellular carrier. This DTMFE project is a proof of concept for implementing my own digital communications with the MODAC computer.

To fill space on this pace,
here's the solar charging circuit for MODAC

DTMFE Theory

DTMF is a well adopted tool for bridging traditional voice carriers and low speed character transmission. The standard, pioneered by AT&T during the 1950's, currently supports digits 0 through 9 and characters A, B, C, D (1). By increasing the number of frequencies available for generating a dual tone, this set should be able to cover most of the common characters found in serial connections.

To start out, we look at the original DTMF frequencies:

	1209	1336	1477	1633
697	1	2	3	A
770	4	5	6	B
852	7	8	9	C
941	*	0	#	D

From the lonestar article on DTMF, we know that AT&T picked these odd looking frequencies specifically to decrease the probability that they will be mistaken for one another. The values minimize overlapping harmonics that will appear when the signal is analyzed in the frequency domain. I'm not very familiar with tools for designing such a grouping, so I found a correlation between the base values and extrapolated to fill the 12x12 extended ASCII table. Then the values were manually inspected and adjusted. This component of the system could use a lot more development to clean up the overlap between the row and column frequencies.

Multiplication between row and column:

	1209	1336	1477	1633	1805	1994	2201	2427	2672	2938	3226	3537
697	842673	931192	1029469	1138201	1258085	1389818	1534097	1691619	1862384	2047786	2248522	2465289
770	930930	1028720	1137290	1257410	1389850	1535380	1694770	1868790	2057440	2262260	2484020	2723490
852	1030068	1138272	1258404	1391316	1537860	1698888	1875252	2067804	2276544	2503176	2748552	3013524
941	1137669	1257176	1389857	1536653	1698505	1876354	2071141	2283807	2514352	2764658	3035666	3328317
1035	1251315	1382760	1528695	1690155	1868175	2063790	2278035	2511945	2765520	3040830	3338910	3660795
1132	1368588	1512352	1671964	1848556	2043260	2257208	2491532	2747364	3024704	3325816	3651832	4003884
1230	1487070	1643280	1816710	2008590	2220150	2452620	2707230	2985210	3286560	3613740	3967980	4350510
1307	1580163	1746152	1930439	2134331	2359135	2606158	2876707	3172089	3492304	3839966	4216382	4622859
1421	1717989	1898456	2098817	2320493	2564905	2833474	3127621	3448767	3796912	4174898	4584146	5026077
1510	1825590	2017360	2230270	2465830	2725550	3010940	3323510	3664770	4034720	4436380	4871260	5340870
1592	1924728	2126912	2351384	2599736	2873560	3174448	3503992	3863784	4253824	4677296	5135792	5630904
1665	2012985	2224440	2459205	2718945	3005325	3320010	3664665	4040955	4448880	4891770	5371290	5889105

Addition between row and column:

	1209	1336	1477	1633	1805	1994	2201	2427	2672	2938	3226	3537
697	1906	2033	2174	2330	2502	2691	2898	3124	3369	3635	3923	4234
770	1979	2106	2247	2403	2575	2764	2971	3197	3442	3708	3996	4307
852	2061	2188	2329	2485	2657	2846	3053	3279	3524	3790	4078	4389
941	2150	2277	2418	2574	2746	2935	3142	3368	3613	3879	4167	4478
1035	2244	2371	2512	2668	2840	3029	3236	3462	3707	3973	4261	4572
1132	2341	2468	2609	2765	2937	3126	3333	3559	3804	4070	4358	4669
1230	2439	2566	2707	2863	3035	3224	3431	3657	3902	4168	4456	4767
1307	2516	2643	2784	2940	3112	3301	3508	3734	3979	4245	4533	4844
1421	2630	2757	2898	3054	3226	3415	3622	3848	4093	4359	4647	4958
1510	2719	2846	2987	3143	3315	3504	3711	3937	4182	4448	4736	5047
1592	2801	2928	3069	3225	3397	3586	3793	4019	4264	4530	4818	5129
1665	2874	3001	3142	3298	3470	3659	3866	4092	4337	4603	4891	5202

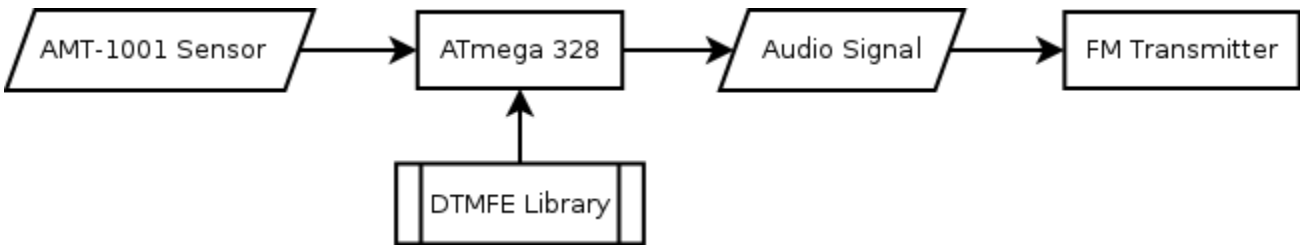
The final ASCII table:

	1209	1336	1477	1633	1805	1994	2201	2427	2672	2938	3226	3537
697	1	2	3	A	R	S	r	s	/	NUL		
770	4	5	6	B	Q	T	q	t	.	SOH		
852	7	8	9	C	P	U	p	u	-	STX		
941	*	0	#	D	O	V	o	v	,	ETX		
1035	H	G	F	E	N	W	n	w	+	EOT		
1132	I	J	K	L	M	X	m	x	)	ENQ		
1230	d	c	b	a	Z	Y	l	y	(	ACK		
1307	e	f	g	h	!	j	k	z	'	BEL		
1421	<	;	:	DEL	~	}	|	{	&	BS		
1510	=	>	?	@	Space	!	"	$	%	TAB		
1592	DC3	DC2	DC1	DLE	SI	SO	CR	FF	VT	LF		
1665	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	RS	US	

We're going to gloss over the Transmission side of the project, as it had more to do with embedded controllers and didn't require much signal processing theory.

The ATmega328 code has two libraries associated with it. One is a modified AMT1001 suite that will take analog data from the temperature and humidity sensors and deliver an integer to my process. The other is a tone generation tool which is attached to two PWM outputs and sends one of the chosen DTMFE waveforms to each pin (2).

The system follows this design:

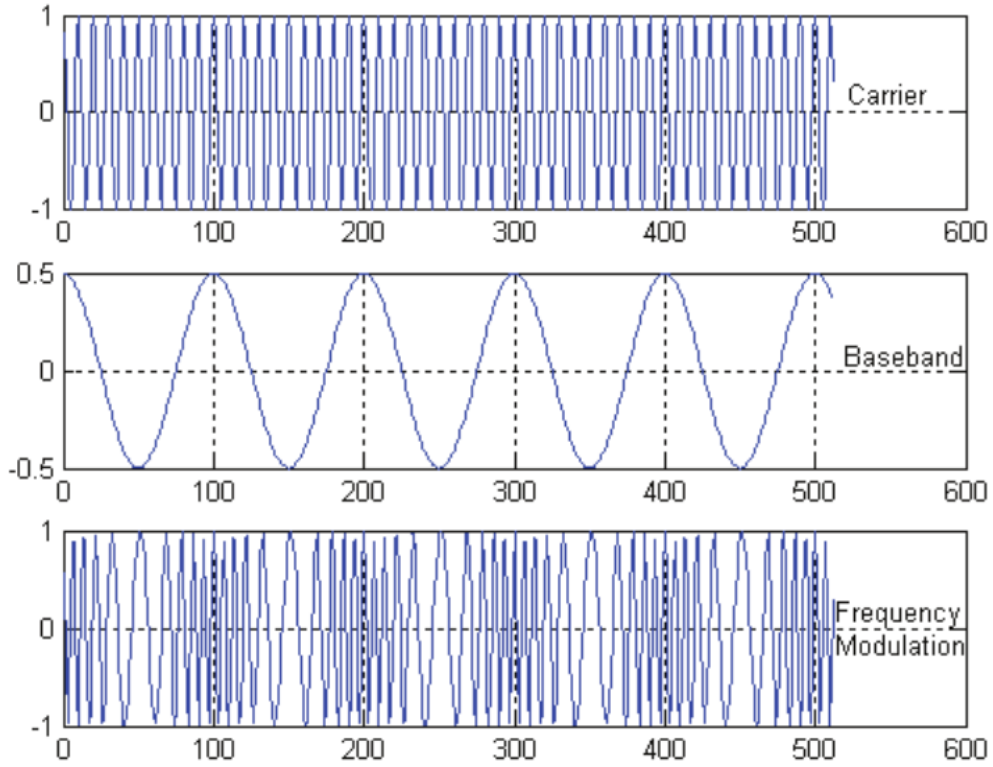


Now we have a standard for what the signal should look like, and a way to broadcast it as wideband FM. The signal has to be gathered and processed in Matlab. Gathering the signal requires a radio receiver and program such as rtl_sdr, the raw IQ data is written to a .dat file.

Reading .dat file
The .dat information is separated with alternating real and imaginary components. The values are 8 bit unsigned integers. Getting them read into Matlab is pretty simple, the elements are converted to doubles, and 127.5 is subtracted from each. Now Matlab has a complex signal.

Signal detection
Detection, or demodulation, is the process of taking our FM carrier signal and recovering the message stored in it. With AM, we know that the carrier can be used to retrieve the message directly. FM appears to be more complicated, but is resilient to the variations in amplitude present in radio.

FM demodulation (3):

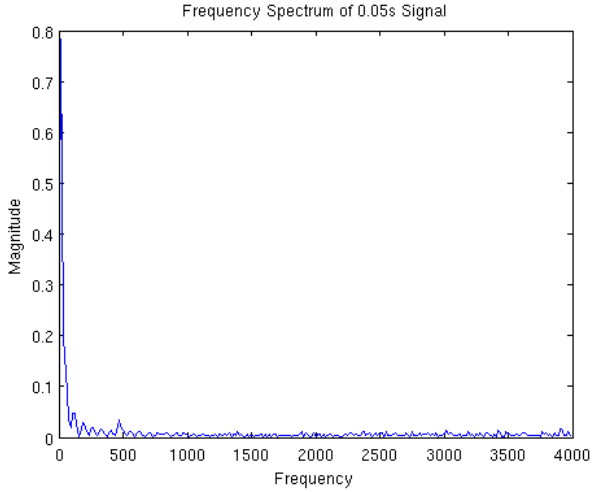


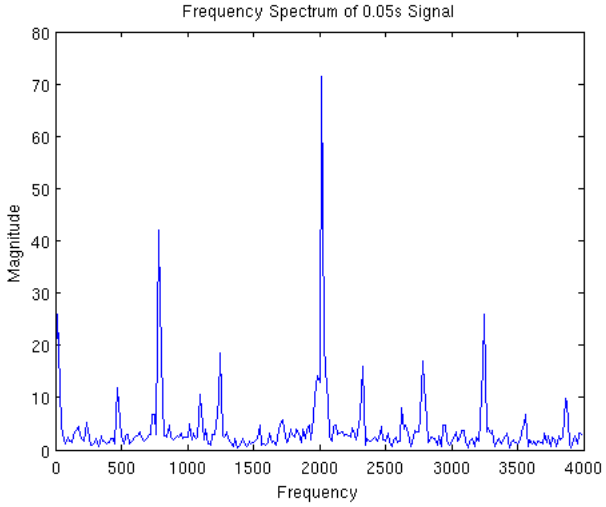
http://www.acfr.usyd.edu.au/pdfs/training/sensorSystems/02%20Signal%20Processing%20and%20Modulation.pdf

This image helped in understanding how recovery of the message is possible. The modulated signal is convolved with an FIR differentiation filter. Matlab's conv function will multiply these two vectors together, which allows us to take the slope of the signal in the frequency domain. This slop then corresponds to the original message signal's amplitude, and can be used to reconstruct the message.

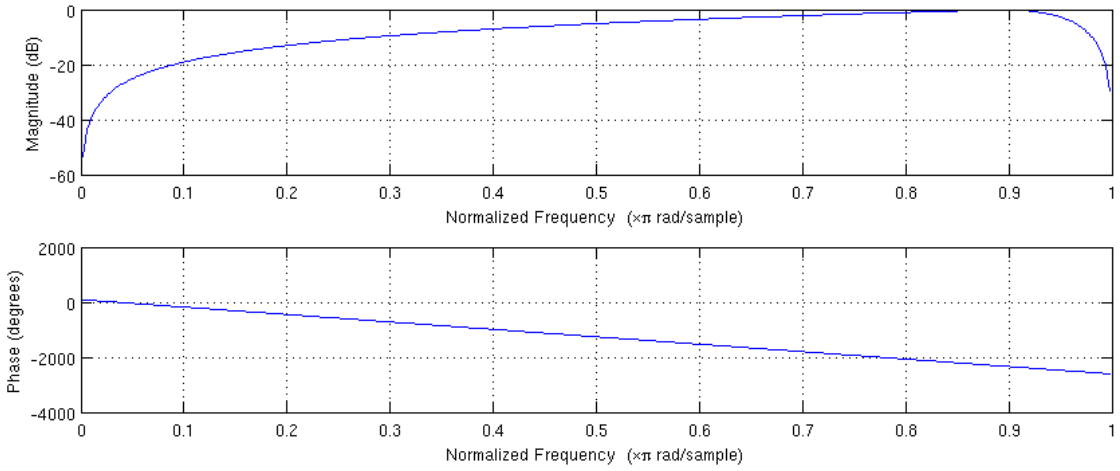
Once the audio has been recovered, a modified version of the course's dtmf_decode.m file is used to find the pair of frequencies and collect the ASCII characters. This file operates by taking small 0.05 second slices of the input signal and referencing them individually to control a state machine. The process lacks a lot of engineering and refinement, since it simply guesses at the ratio of noise found in silent portions to signal present during the tones. The governing state machine has two states to administer action on either silence or signal. When silence is detected, the system will discard the slices of signal until one has a cumulative power some factor higher than the others. Once a signal is detected the Matlab function will accumulate the small slices of signal until one has a lower cumulative power. The accumulated signal is then processed by a FFT to find its frequency spectrum. Here summation of the absolute value of the frequency representation is used as a rough stand in for an actual power calculation. Frequencies that land close to the DTMFE frequencies and have a local maximum are chosen and used in a lookup cell array to find the associated ASCII character.

The frequency spectrum of a section of silence compared with the frequency spectrum of a section of signal (two largest spikes are DTMFE tones)





The differentiation filter response:



Implementation

Using the Matlab files is straightforward.

To capture IQ data, use the command:

```
rtl_sdr -f 103800000 -s 2048000 signal.dat
```

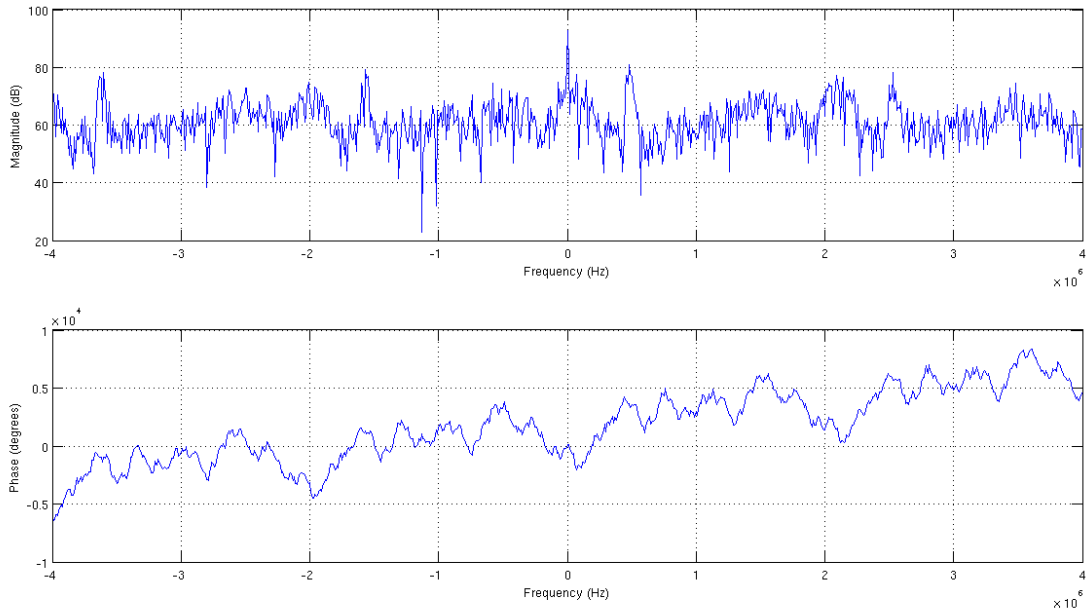
We're gathering a 2.048MHz bandwidth signal centered around the 103.8MHz broadcast. Its important that the signal we wish to decode is located at the center, the scripts do not currently support shifting to another section of the bandwidth before demodulating.

To process the file, use proc.m with the file's paths set accordingly. This should return a variable called “chars” which will contain the extracted characters.

Results

When there is no audible interference, the process will reliably return the correct characters. If there's audio, such as another radio station's broadcast leaking into our 103.8MHz target, as was the case during the presentation, then the interfering audio will cause the decoding function to incorrectly detect the change in states, which will cause character to either fail to detect or to be detected twice in a row.

I've saved the IQ data from class, and have processed a section of the raw signal's spectrum:



Our broadcast is centered at 0Hz on this plot, and we can see just half a megahertz to the right is 104.3MHz, a radio station in Huntsville. I believe with a better fitted filter, that voice could be eliminated from the signal sent for decoding.

Code

loadIQdat.m

```
function [ dat ] = loadIQdat( file )
% loadIQdat
% This function will parse a dat file output by rtl_sdr
% The .dat file is unsigned, so we have to center the signal.

% Open file handle
fid=fopen(file,'rb');

% Convert to double
dat=fread(fid,'uint8=>double');

% .dat is unsigned 0-255, center signal
dat=dat-127.5;

% Combine Real and Imaginary elements
dat=dat(1:2:end) + (1i*dat(2:2:end));

end
```

FM_IQ_Demod.m

```
function [y_FM_demodulated] = FM_IQ_Demod(y)
% FM_IQ_Demod
% This function demodualtes an FM signal. It is assumed that the FM
signal
% is complex (i.e. an IQ signal) centered at DC and occupies less
than 90%
% of total bandwidth.

% Design filter / differentiator
filter=firls(30,[0 .9],[0 1],'differentiator');

% Get the normalized signal?
diff=y./abs(y);

% Real portion of the signal
rd=real(diff);

% Imaginary portion of the signal.
id=imag(diff);

% Detected signal
y_FM_demodulated=(rd.*conv(id,filter,'same')-
id.*conv(rd,filter,'same')).(rd.^2+id.^2);

end
function [ chars ] = dtmfeDecode( tone, fs )
% dtmfeDecode - Extract ASCII from audio
% input - 1x? audio matrix, sampling frequency
% output - 1x? matrix of chars

% From previous dtmf homework:
```

```
% I plan to check 0.05 second chunks since the smallest
% audio (silence) is 0.1s. So we're employing Nyquist's Frequency
% to make sure that the silence is noticed. Then we'll parse the
% incoming signal by using silence as a cue to move on to the
% next button tone. In order to make the frequency detection more
% accurate, the tone slices of 0.05s will be stacked until silence
% is detected. Then the stacked tone is analyzed for frequencies.

% Specific to this file:
% Things left to do include:
% - Creating some graphical output of the signal during manipulation
% and the text recovered
% - Implementing a more robust detection for silence between tones.
% I don't understand the derivation of power for a signal very well,
% and believe there are better ways to calculate it.
% - Perform a rough size estimation before detecting characters, and
then
% initialize 'chars' with a reasonable guess at the # of chars.
%

% Variables
rd=0.05; % Reference duration
cols=[1209 1336 1477 1633 1805 1994 2201 2427 2672 2938 3226 3537];
% Column frequencies
rows=[697  770  852  941  1035 1132 1230 1307 1421 1510 1592 1665];
% Row frequencies
len=length(tone); % Length of input
chars_index=1; % Index for list of keys
prev_sig=zeros(1,rd*fs); % Holds the audio for the precious rd*fs
chunk
prev_power=sum(prev_sig); % Holds the cumulative power of the
previous chunk
stacked_sig=zeros(1,len-1); % Stacked sig holds all previous chunks
for a given tone
% States
% 1 - Consuming silence
% 2 - Stacking tone chunks
state=2;
chars_list={
'1'     '2'     '3'     'A'     'R'     'S'     'r'     's'     '/'
'NUL'   'NA11'  'NA12';
'4'     '5'     '6'     'B'     'Q'     'T'     'q'     't'     '.'
'SOH'   'NA10'  'NA13';
'7'     '8'     '9'     'C'     'P'     'U'     'p'     'u'     '-'
'STX'   'NA9'   'NA14';
'*'     '0'     '#'     'D'     'O'     'V'     'o'     'v'     ','
'ETX'   'NA8'   'NA15';
'H'     'G'     'F'     'E'     'N'     'W'     'n'     'w'     '+'
'EOT'   'NA7'   'NA16';
'I'     'J'     'K'     'L'     'M'     'X'     'm'     'x'     ')'
'ENQ'   'NA6'   'NA17';
'd'     'c'     'b'     'a'     'Z'     'Y'     'l'     'y'     '('
'ACK'   'NA5'   'NA18';
'e'     'f'     'g'     'h'     'I'     'j'     'k'     'z'     ''''
'BEL'   'NA4'   'NA19';
'<'     ';'     ':'     'DEL'   '~'     '}'     '|'     '{'     '&'
```

```
'BS'    'NA3'   'NA20';
'='     '>'     '?'     '@'     'SPACE' '!'     'â€œ'     '$'
'%'     'TAB'   'NA2'   'NA21';
'DC3'   'DC2'   'DC1'   'DLE'   'SI'    'SO'    'CR'    'FF'    'VT'
'LF'    'NA1'   'NA22';
'DC4'   'NAK'   'SYN'   'ETB'   'CAN'   'EM'    'SUB'   'ESC'   'FS'
'RS'    'US'    'NA23';
}; % Go backwards from frequency index to key



for section = 0:1:uint64(floor( ( (len)/fs )/rd )-1)

    % Extract 0.05s chunk of signal
    this_sig=tone( ((section*fs)*rd)+1 : ((section+1)*fs)*rd );

    % Get the power spectrum of this chunk
    [p_vals,p_sig]=freqSpec_1s(this_sig,fs);

    %figure();
    plot(p_vals, abs(p_sig));
    title('Frequency Spectrum of 0.05s Signal');
    xlabel('Frequency');
    ylabel('Magnitude');

    % Get a rough overall magnitude for comparison
    p_power=sum(abs(p_sig));

    % State 1, listen for start of new tone
    if state == 1

        % Check for start of new tone
        if p_power > (prev_power*1.1)
            stacked_sig=zeros(1,len-1);
            state=2;
        end

    % State 2, college new tone and find ASCII frequencies
    elseif state == 2

        % Stack signal slices
        stacked_sig( ((section*fs)*rd)+1 : ((section+1)*fs)*rd ) =
this_sig;

        % Check for end of this tone
        if p_power < (prev_power/1.1)
            state=1;

            % Get the rough power spectrum
            [stacked_vals,stacked_p]=powerSpec(stacked_sig,fs);
            [peak_power, peak_I]=max(stacked_p);

            col_index = 1;
            row_index = 1;
            col_range_min = 1;
            row_range_min = 1;
```

```
            for iterator = 1:12

                % Create subranges to match the frequencies we're
looking for
                range=10;
                max_col_range=find(stacked_vals>(cols(iterator)-
range) & stacked_vals<(cols(iterator)+range));
                max_row_range=find(stacked_vals>(rows(iterator)-
range) & stacked_vals<(rows(iterator)+range));

                % Get the index of maximum value
                [C_col,I_col]=max(stacked_p(max_col_range));
                [C_row,I_row]=max(stacked_p(max_row_range));

                % Safety net to ensure only suitably large tones are
                % recognized
                if C_col > (peak_power/3) && C_col >
stacked_p(col_index)
                    col_index = I_col;
                    col_range_min = min(max_col_range);
                end

                if C_row > (peak_power/3) && C_row >
stacked_p(row_index)
                    row_index = I_row;
                    row_range_min = min(max_row_range);
                end

            end

            % Find the corresponding frequency

col_button_freq=floor(stacked_vals(col_index+col_range_min-1));

row_button_freq=floor(stacked_vals(row_index+row_range_min-1));

            % Get the chars_list index
            col_index=find(cols>(col_button_freq-10) &
cols<(col_button_freq+10));
            row_index=find(rows>(row_button_freq-10) &
rows<(row_button_freq+10));

            % Safety net, only stored detected chars
            if isempty(col_index)
                continue;
            elseif isempty(row_index)
                continue;
            end

            % Find the key value and append to list
            chars(chars_index)=chars_list(row_index,col_index);
            chars_index=chars_index+1;
        end % End of frequency gathering
    end % End of state model
    prev_power=p_power;
```

```
end

if isempty(chars)
    chars(1:5)={'E' 'r' 'r' 'o' 'r'};
end

end
```

powerSpec.m

```
function [ p_vals, p_sig ] = powerSpec( signal, sfs )

t_len=length(signal); % number of samples
nfft=2^ceil(log2(t_len)); % number of FFT bins

p_vals=sfs*(0:nfft/2-1)/nfft; % Frequency range (positive only)
y1_f=(fft(signal,nfft)); % Two sided, zero centered FFT
Py1=y1_f.*conj(y1_f)/(nfft*t_len);
p_sig=Py1(1:nfft/2);

end
```

proc.m

```
% Process an IQ .dat file

% Input IQ data:
% Fs - 2048000, 2.048 MHz
% Center freq - 130800000, 130.800 MHz
%
% Sample:
% 15 seconds
% 30720000 complex samples
%
% DTMF
% 8kHz audio

% Get the data
raw_iq=loadIQdat('/home/berocs/Documents/uah/ee384/final/data/signal
.dat');

% Setup some convenience variables
iq_fs=2048000;
iq_size=size(raw_iq);
iq_seconds=(iq_size(1)/iq_fs);

% Filter other radio stations
decim=decimate(raw_iq,16,'fir');

% Get demodulated signal
demod=FM_IQ_Demod(decim);

% Lower sampling rate to 8KHz
demod_8k=decimate(demod,16,'fir');
```

```
% Write the audio
wavwrite(demod_8k, 8000,
'/home/berocs/Documents/uah/ee384/final/data/signal.wav');

% Get audio signal
[tone,fs]=wavread('/home/berocs/Documents/uah/ee384/final/data/signa
l.wav');

% Parse with dtmfeDecode
keys=dtmfeDecode(tone,fs);
```

Resources

[1] http://nemesis.lonestar.org/reference/telecom/signaling/dtmf.html

[2] https://code.google.com/p/rogue-code/wiki/ToneLibraryDocumentation

[3] http://www.acfr.usyd.edu.au/pdfs/training/sensorSystems/02%20Signal%20Processing%20and%20Modulation.pdf

[4] http://www.aaronscher.com/wireless_com_SDR/RTL_SDR_AM_spectrum_demod.html