300 Baud Soft Modem

Wade Berrier

Directed Readings
CS 6950
Utah State University

21 June, 2012

300 Baud Soft Modem

Wade Berrier

Directed Readings
CS 6950
Utah State University

21 June, 2012

1. Project Description

Implementing a 300 baud modem [5] in software is a great way to become familiar with some key digital signal processing primitives. The project is laid out in terms of milestones that lead up the to completion of the modem. The ultimate goal of the project is to be able to communicate with an actual 300 baud hardware modem, but that may be out of the scope of the project, and is left as extra credit. In order to exercise most of the functionality, the more attainable goal is to have data modulated out a sound card, recorded and processed in real-time by another sound card input.

Several resources [7, 10] and fellow work colleagues will be consulted throughout the course of the project.

2. Milestone: Generating Tones

The first milestone is to generate a sine wave tone in a digital form to be played by a sound card. A raw *pulse code modulation* (PCM) format is used to store the tone in a file in digital form. Various programs can be instructed to read and play this file through the sound card.

Periodic Sampling [7, chapter 2] was referenced in order to understand and implement the first milestone. This is the process of turning a continuous sound wave into series of discrete data values. Although a sound interface supports many formats, *32 bit floating point* was chosen instead of *16 bit signed integers* (which is more common) to avoid issues of scaling and overflow. Although not difficult to deal with, it is one less thing to worry about, and again, the sound hardware interface can understand both formats.

The main guts of the process relies on the following equation:

$$x(n) = \sin(2\pi f_0 n t_s)$$

where f_0 is the frequency of the resulting tone, n is the index of the sample, and t_s is the period of the sample in the time domain, which is the inverse of the sampling rate, which is in Hertz[1]. The duration of the tone can be calculated by the number of desired samples multiplied by the sample rate.

Several methods were used to test the tone generation algorithm. The first was generating audible tones of a constant frequency. Playing those tones[2] results in a smooth audible note.

Another method used were Dual-tone

[1] the measurement for cycles per second

[2] See *Hearing Data* Appendix on page 5

multi-frequency (DTMF) tones, which are a combination of two tones which are used to control dialing through a telephone network. These are created by generating two independent tones, and then summing the tones together, sample by sample, to create a new signal consisting of both tones. This result then was scaled to be within range the sound card was expecting[1] . These tones were casually verified by comparison to the tones on a telephone.

The final method used was generating tones from the frequencies of the musical scale. Translating letter notes into frequencies and then a series of tones was the final verification that the tones were being generated correctly.

Special consideration must be taken into account in order to avoid phase discontinuities when generating samples that have successive frequencies. If this is not taken into account, the discontinuities result in glitches in the audio output. This is because the samples for a frequency may not end on a period boundary, and if the next frequency starts at zero again, this creates the discontinuity. In order to have a continuous waveform, the next samples for a frequency need to start where the previous set of samples left off.

Continuous waveforms consisting of multiple frequencies can be generated using an oscillator [3, section 4.3]. The optimized technique consists of sampling a single period of a waveform. The fidelity of the waveform depends on the number of samples used to represent this single period[2]. These samples are stored in a table. In order to generate samples of a particular frequency, a sampling interval can be derived from the frequency and used to step through the table. The table is treated as a ring buffer, so that when the end is reached the increment continues at the beginning of the table. The equation to calculate the sampling index is as follows:

$$SI = N\frac{f_0}{f_s}$$

where SI is the sampling increment, N is the number of samples in the table, f_s is the sampling rate, and f_0 is the target frequency to be generated. This allows for continuous wave generation, which helps alleviate some problems discussed later in *MileStone: Demodulating Data in Realtime.*

3. MileStone: Modulating Data Using Frequency Shift Keying

The telephone network uses a *voice grade channel* which is capable of sending 64 kilobits of digital data. Analog sound is sampled and sent over these channels. A 300 baud modem is full-duplex, meaning it is able to send and receive data at the same time. This is accomplished via a method called *Frequency Shift Keying.* The sender represents binary data by transmitting one tone for a *zero*, and another for a *one.* One side uses frequencies 1180 and 980 Hertz, and the other side uses 1850 and 1650 Hertz. By using different frequencies, each side is able to transmit and receive at the same time.

One aspect that was unknown at the start of the project was line encoding. V.21 uses *asynchronous serial communication* [1] to encode the data. This means that while no data is being sent, the line is held high[3]. Just before a byte of data is transmitted, a *start bit* is sent by pulling the line low[4] for a symbol length. The

[1] floating point has a range between -1 and 1

[2] 512 samples were used in this project

[3] represented by a bit of value 1

[4] represented by a bit of value 0

8 bits of data are then clocked on the line, followed by putting the line high with a *stop bit*. The next time the sender wants to send a byte of data, this process is repeated.

Once the desired data is encoded for a serial line, it can be modulated using FSK. The methods used for generating tones described earlier are used to encode a string of bits into a waveform. The V.21 standard describes sending 300 bits per second. That means that a timeslot for one of these bits is 1/300 of a second. Waveforms at a specified rate at a specific frequency are generated for this length of time to represent a bit. These waveforms are concatenated together and stored as little endian 32 bit binary floats in a PCM file. Because V.21 is full duplex, waveforms representing the sender and receiver's data can be added together and scaled as a new waveform. Listening to these waveforms spawn a flood of nostalgia.

4. MileStone: Demodulating Data Using Frequency Shift Keying

Demodulating is done by loading the samples, separating the samples into groups to represent symbols, and deciding what symbols (1 or 0) are represented by the samples. Once these samples come from a sound card they will contain noise. Because these waveforms at this point are of high quality without noise, it is relatively straightforward to decide the symbols.

A *Fast Discrete Fourier Transform* (FDTF) [7, chapters 3-4, 10, chapter 12] was used in analyzing the samples. Source code [6] was adapted to do a transform on the symbol data. The input to the FFT is *complex* (imaginary) data. The symbols from the waveform are *real*. The real data was padded with zero for the complex portion. The numbers of these samples must be a power of 2 and greater granularity can be gained by padding the samples with zeros up to the next power of 2.

Taking the absolute value of the complex values returned by the transform represent the accumulation of frequencies found in the samples. Each of the returned numbers represent a bin or bucket of frequencies. The frequencies contained in each bin have the frequency width of half the sampling rate[1] divided by the number of bins (which is also the same number of inputs to the FFT). Because of the lack of noise, the bins with the highest count can be used to determine the frequency/tones/symbols/bits in the waveform. This is repeated for all the chunks of the waveform to pull out the bits. The data is then decoded using the same serial line scheme that is used for encoding.

Because sample boundaries are known, discontinuities in the waveform did not hinder the demodulation of the symbols. This is not the case when it comes to reading samples from the sound hardware as the symbol boundaries will not be known and will have to be detected.

5. MileStone: Modulating Data in Realtime

Up to this point all processing has been done on samples that have been read or stored in PCM formatted files. This has avoided the problem of keeping in sync since the start and stop of each of the symbols is known. It has also avoided accounting for noise introduced by recording from the sound card. These issues become more of a challenge when processing happens in real time.

Handling timing and synchronization during modulation is mostly trivial. The main addition to the algorithm so far was to send

[1] Nyquist-Shannon Sampling Theorm [9]

mark tones while there was no data to transmit. When the user presses a key, the byte is encoded and modulated and the samples representing the waveform are put on the output queue. The output audio stream is configured for a particular rate (48000 Hz in this case) and the audio driver calls a preconfigured callback in order to keep the output buffer full. The callback buffers the samples from the output queue that contains the user inputted data, and if the queue is empty, samples representing the mark tone are buffered instead. The result is the program plays a tone while idle, and when the user presses a key, modulated gibberish[1] is momentarily played through the sound card speakers.

6. MileStone: Demodulating Data in Realtime

Much of the framework used to modulate in real time was also used to demodulate the samples. Data was pulled off the sound hardware and placed in an input queue. The streaming demodulator then runs in a separate thread and processes the samples in the queue in order to transform them into bytes.

The main challenge with the real time demodulator is trying to figure out where the symbol boundaries are. Again, when reading samples from a generated vector or a file, the samples boundaries are known making it more straightforward to demodulate the symbols. This is where the data encoding [1] is helpful. In order to detect the start bit, the number of samples for half a symbol is analyzed to detect whether it is solely a *one* or *zero.* When this is detected, it can be assumed the demodulator is in the middle of a symbol boundary. At this point, an additional 9 half symbol samples[2] are collected and demodulated, knowing that these samples are also most likely inside the symbol boundaries.

Problems can arise when discontinuities or garbage samples are recorded from the sound card. Not only does this introduce bit errors, it throws off the synchronization of the symbol boundaries. When this occurs, the stream is simply put into the *mark* state and synchronization recovery usually happens within a few seconds.

7. Future Work

More investigation needs to be done in order to find out why glitches are being modulated out the sound card. This is causing the real time demodulator to occasionally loose sync and drop bits.

Plus, if the data being modulated is coming in at a higher rate than 300 baud that is being processed by the audio hardware and demodulator, memory eventually increases until all memory is consumed. This could be alleviated with a simple flow control technique by using a maximum audio output queue size.

Currently a 4096 point DFFT is performed in order to detect the frequencies in the presence of noise. Passing the data through a *window function* may help to get sharper Fourier transform results, which may make it possible to low the number of points of the DFFT, drastically decreasing processing time. It may also be worthwhile to use Goertzel's algorithm [7, section 13.7.1] to detect the tones. This basically computes the frequency distribution for a limited number of bins instead of all the bins represented by the full Fourier transform.

Ultimately the software modem needs to be able to communicate with a commercial modem. Even though this soft modem can communicate with itself, there may be other stip-

[1] a highly technical yet fitting description

[2] 8 for the data bits, and an additional bit for the stop bit

ulations or requirements that were not noticed during the course of the project. Another interesting use case would be to communicate with Amateur Packet Radio networks (aka: HAM radio). They use FSK in their AX.25 protocol [2].

Towards the end of the project it was noticed that commercial modems may also use some form of Non-Return-to-Zero (NRZ) encoding in order to normalize the number of transitions through the signal. Implementing this would be trivial, but may be required by the standard for 300 baud modems.

Also, there are many more modulation techniques used in more advanced modems, including *Phase Shift Keying, Quadrature Amplitude Modulation, trellis encoding, compression, forward error correction* and other techniques in order to maximize the bandwidth to shove as many bits as possible in a signal (be it audio, optical, or radio).

8. Conclusion

TODO

Appendix A. Plotting Data

Being able to plot equations and data is extremely helpful in developing and debugging a software modem. Python and matplotlib[1] [8] were used to visualize the data. The C++ code writes the data to a temporary file and invokes a python script to read and plot the data. This makes it easy to place hooks in the code to plot the data at different points in the code.

Gnuplot [4] was used to plot samples in realtime. Because matplotlib is implemented in python, it was assumed that piping data to gnuplot would outperform matplotlib, even though it was the only avenue attempted.

Appendix B. Hearing Data

Listening to the data is also a useful debugging technique. PCM data is generated and played to the sound hardware via a Linux command *aplay*. Hearing the PCM files can be played in order to listen to the generated tones to make sure they are generated correctly. It is obvious to the human ear if the sample format is not correctly represented on disk.

References

[1] Wikipedia. Asynchronous Serial Communication. URL *http://en.wikipedia.org/wiki/Asynchronous_serial_communication.*

[2] WikiPedia. AX.25. URL *http://en.wikipedia.org/wiki/AX.25*.

[3] Charles Jerse and Thomas A. Dodge. *Computer Music: Synthesis, Composition, and Performance.* Schirmer Books, 1997.

[4] Gnuplot Authors. Gnuplot. URL *http://www.gnuglot.info.*

[5] ITU. 300 Bits Per Second Duplex Modem Standardized for use in the General Switched Telephone Network. Tech. Rep. V.21 (1988), International Telecommunication Union. URL *http://www.itu.int/rec/T-REC-V.21-198811-I/en.*

[6] Sergey Chernenko. Fast Fourier Transform. URL *http://www.librow.com/articles/article-10*.

[7] Lyons, Richard G.. *Understanding Digi-*

[1] modelled after Matlab's plotting facilities

tal Signal Processing. Prentice Hall, 2011.

[8] Matplotlib Authors. Matplotlib. URL *http://matplotlib.sourceforge.net/*.

[9] WikiPedia. Nyquist-Shannon Sampling Theorem. URL *http://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem.*

[10] Smith, Steven W.. *Digital Signal Processing - A Practical Guide for Engineers and Scientists.* Newnes, 2003.