

18-551, Spring 2005
Group 3, Final Report

The Wireless Intercom System

Jesse Vizcaino	jvizcain@andrew.cmu.edu
Kermin Fleming	kfleming@andrew.cmu.edu
Muhammed Wahabsuhaili	mwahabsu@andrew.cmu.edu

The Wireless Intercom System

Table of Contents

Introduction	3
Previous Projects	4
System Architecture	4 - 5
System Block Diagram	6
Algorithms	7 - 20
Voice Compression: CS-ACELP	7 - 10
Encryption and Correction	11 - 15
Wireless Communication: CDMA	15 - 20
Available Software and Issues	21
Working with the EVM	22 - 24
Demonstration and test data	25 - 28
Conclusion	28
References	29

The Wireless Intercom: An Introduction

Consumers need communication technology in this information era. There is a great demand for reliable communication that is easy to use as well as secure. Email is one solution but will never be secure enough to stop someone from reading over your shoulder. Additionally, on a large company floor remembering several phone numbers is easily a hassle, especially when you only need to make a quick phone call down the hall. How can we make communication better or even easier than this for a simple, closed environment? Can it still be reliable if it is “better” or “easier”? Is there a need for security without complicating the system?

It is obvious then that as technology advances the demand for better communication solutions increases tremendously. Wires are quickly going out of style with the advancement of wireless technologies and industry. Because wireless communication is a surging technology that shows no signs of slowing down, it has the potential to answer to the needs of this demanding market. Consider one of the most common forms of wired communication, the landline telephone. A great amount of money has to be poured into a wired infrastructure, which even though is fairly common place, contains a hidden complexity. Wireless eliminates the need for additional hardware to connect lots of users for a simple environment.

A solution to these outlined problems then would be a wireless intercom system using encryption, where a group of people could communicate quickly and securely without the need for major installation projects or complicated hardware systems. First consider the situation where a business team, lets call them team A, needs to discuss an important account without having to continually hold meetings. Most people would agree that organizing meetings is not always time or cost effective. With the wireless intercom system, each member of the team can communicate with one another instantly through a reliable link on a secure channel. Security is a particularly important feature if you consider in the same example a rival business team, team B, on a lower floor of the building that would like to see what team A is working on, to possibly usurp their efforts. This system is ready to handle problems such as multiple access on top of common wireless problems such asynchronous communication and noise. Using a multiple access scheme such as CDMA, each user will be identified as unique by a code. What this means is that when one team member is talking to another team member, a third person won't interrupt the conversation but instead will be interpreted as noise. Since CDMA is structured around spread spectrum, we can spread out the power spectrum of our system such that we could transmit at or below the noise floor. Communication would then have encryption and a low probability of interception which would guarantee the users a safe mode of communication. Consider the intercom system divided into 3 major modules: the voice compression, the encryption/error correction, and the wireless communication.

Previous Projects

There have been two prior projects that involved basic components of our design. Group 1, in spring 2001 used linear predictive coding to do a voice morphing project; however, the coder used was a waveform encoder, which is designed to encode an audible signal in a slightly compressed format. As waveform encoders are intended for general audio applications, they do not achieve the compression possible when using a voice encoder. Human speech typically contains a variety of redundancies, as it is based on a fixed set of phonemes; thus, vocoders are able to search through fixed databases of simulated speech in order to choose filter coefficients that minimize the differences in the input signal and the synthesized signal. Additionally, vocoders typically produce better compression than the waveform encoders. The prior project and our project are similar in that they both use sound encoders, but in terms of application they are radically different.

In spring 2000, group 6 did a project similar to ours. The previous project sought to take voice input, use a vocoder to compress it and then transmit the encoded voice over a specialty modem using CDMA. They then intended to demodulate and decode the signal and play back the transmitted voice. Our project differs in several respects. First, we intend to use a different vocoder of slightly higher quality. Second we intend to demonstrate that our CDMA system can handle multiple users, whereas the previous project only had one user on the communication channel at any given point. The previous project also attempted to build their own modem, which they had great difficulty with, to the point that their project completely failed. Initially, we too wanted to either buy or build our own specialty communications hardware. Fortunately, we took a lesson from this project and decided to do a software simulation.

System Architecture

The overall system architecture consists of three major modules which include smaller support functions. User input will be simulated by 16 bit PCM audio files (sampled at 8 KHz with 16 bit quantization) obtained via microphone connected to the PC sound card. For each user desired there will be one audio file saved on the PC specifiable in the graphical user interface (GUI) to be sent for communication. Again, the saved files are simulating real time voice transmission since the entire system is too complex to support real time and in the interest of meeting our deadlines. More discussion on this topic is provided in the wireless algorithms portion.

The GUI will send each user file to the EVM to be compressed using our CS-ACELP vocoder using an HPI transfer to be stored on the EVM's 8MB SDRAM. Encoding one frame corresponds to 80 samples of speech signal or 10ms of speech, which is paged to on-chip RAM. After one frame is processed, the resulting 80 bit compressed speech is stored on a different section of the SDRAM. The process repeats until all the frames in each file are processed. After EVM processing is completed, the bit stream will be sent back to the PC using HPI transfer and saved as an intermediate binary file. This intermediate file will be sent to the encryption and ECC module next for processing.

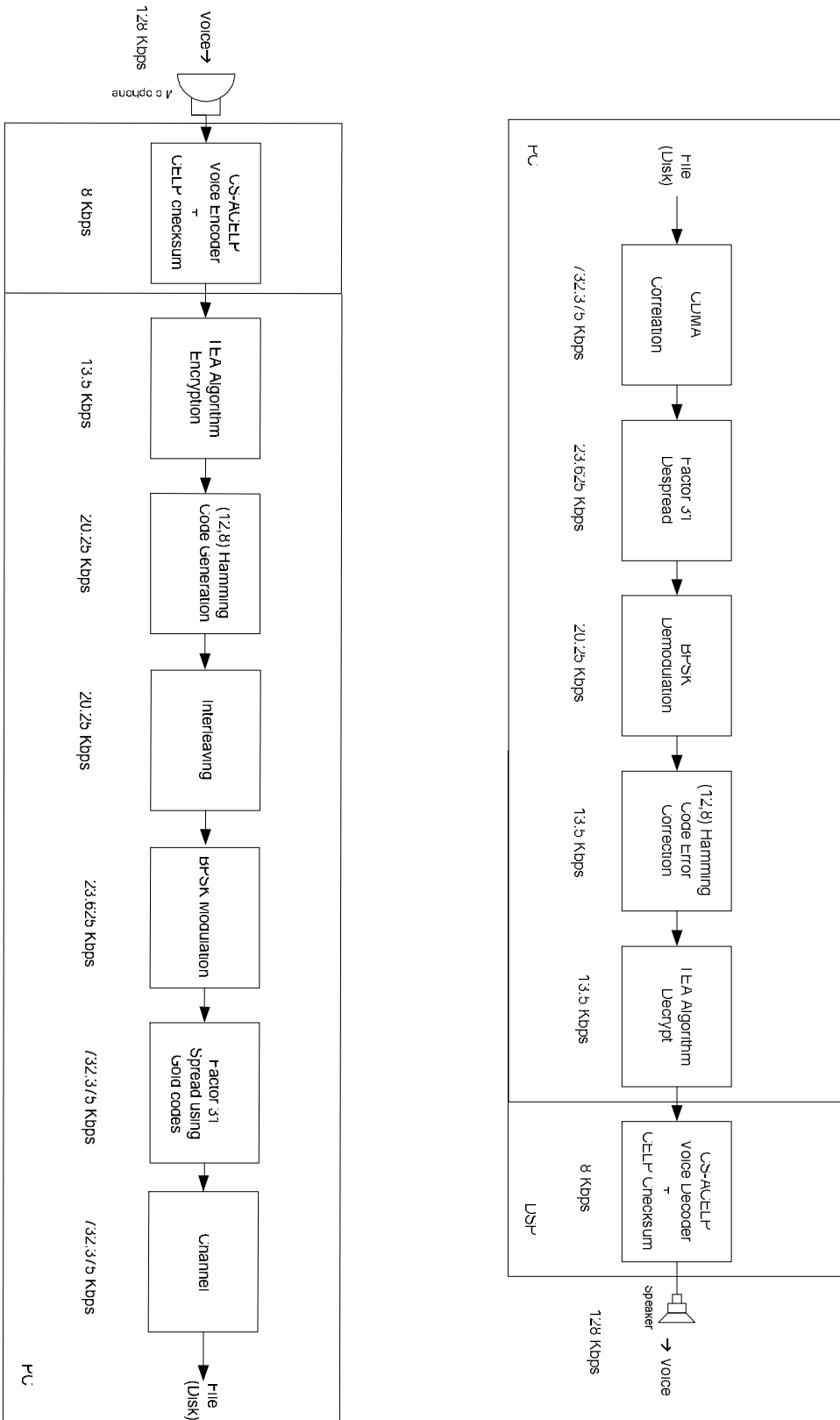
Upon entering this module, the data rate is 8 kilobits per second (kbps). This rate is immediately increased to 13.5 kbps by extending each 80 bit packet to 128 bits since our encryption algorithm requires 64 bit blocks of data. We add 16 garbage bits to the packet and then add a checksum, which will later be used to check for packet corruption. The checksum is calculated over the first three words in the packet and inserted as the fourth word of the packet, which gives us the full 128 bits we need for encryption. The extended packet is encrypted using the TEA algorithm. Next, the bits in the packet are word-wise interleaved. Note that neither of these steps increases the data rate. For the last step of this module, error correction codes are tabulated across each byte of the packet. Since we are creating four more bits for every existing eight using a (12, 8) Hamming code, the data rate is increased by 50% to 20.25 kbps, with a packet size of 192 bits. At this point, the packets are ready to be sent to the wireless component of the system for simulated transmission.

The CDMA module takes the data from the previous blocks where it first adds a 32 bit preamble to each packet making the new size 224 bits so that it can later be synchronized with the receiver. This step increases our bit rate to 23.625 kbps. Each file can be modulated using BPSK or QPSK. BPSK does not change the bit rate but QPSK multiplies by a factor of two for 47.25 kbps. Data is then converted to spread spectrum through a pseudorandom noise (PN) sequence used as CDMA codes to uniquely identify each user. The GUI lets the user choose between PNs of length 20 or 30 or enabling Gold codes, a special type of PN sequence which will be discussed in detail later, of length 31. The old bit rate is multiplied by the spread factor and can range from 472.5 kbps to 1.476 Mbps. The spread out data is sent over an additive white Gaussian noise (AWGN) channel model where each of the specified users will interfere, as they are all transmitted on the same channel. The receiver will then locate the correct user, also specifiable by the GUI, and despread as well as demodulate the received data to store it as a binary file for the decryption and correction module.

Once the binary file has been decrypted, it can be transferred back to the EVM for decompression using HPI transfer to be stored on the SDRAM. For decoding, the EVM will page in 80 bits from the bit stream. The decoding process yields 80 samples of reconstructed speech signal, yet again stored on a different portion of the SDRAM. The process repeats until all the bits in the stream have been processed.

After processing, the reconstructed speech file will be sent back to the PC using HPI transfer to be saved as a PCM file where the user then can playback the file through the PC's soundcard.

System Block Diagram – Transmission and Receiving



Algorithms: Voice Compression using CS-ACELP

Since we will be simulating a practical wireless communication system, there are certain restrictions that constrict our project in terms of resources, the transmission bandwidth and data storage. Since our system is multiple access, which is detailed later on, bandwidth takes priority. If we had all the resources in the world, we would have been able to do without compression at all; however, this is not the case. Our project is based on transmitting a person's voice over a wireless channel and resources are limited, so we have a requirement for voice compression.

Human Speech and LPC Modeling

Most low bit rate voice coders are based on Linear Predictive Coding (LPC) to a certain extent. LPC modeling takes advantage of the predictable elements of human speech to “predict” sample values. This achieves a huge factor of compression compared to Pulse Code Modulation (PCM) schemes where all samples are digitized regardless of whether the samples do not represent any excitation in the speech. It is best to examine the very nature of human speech and how LPC attempts to model this natural occurrence mathematically.

There are two types of speech sounds, voiced and unvoiced, that produce different sounds and spectra due to their differences in sound formation. With voiced speech, air pressure from the lungs forces normally closed vocal cords to open and vibrate. The vibration frequencies (pitch) vary from about 50 to 400 Hz, depending on the person's age and sex, and forms resonance in the vocal tract at odd harmonics. See figure 1.1 for a sample waveform of voiced speech.

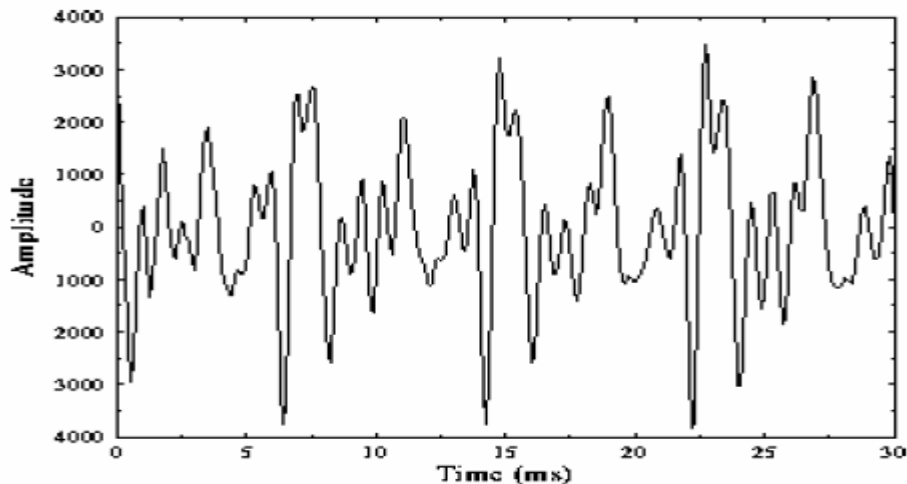


figure 1.1 Waveform of voiced speech

Unvoiced sounds, called fricatives (e.g., s, f, sh sounds) are formed by forcing air through an opening, hence the term, derived from the word “friction”. Fricatives do not vibrate the vocal cords and therefore do not produce as much periodicity as seen in the

formant structure in voiced speech. Unvoiced sounds appear more noise-like. As you can see from the waveform in figure 1.2, the time domain samples lose periodicity.

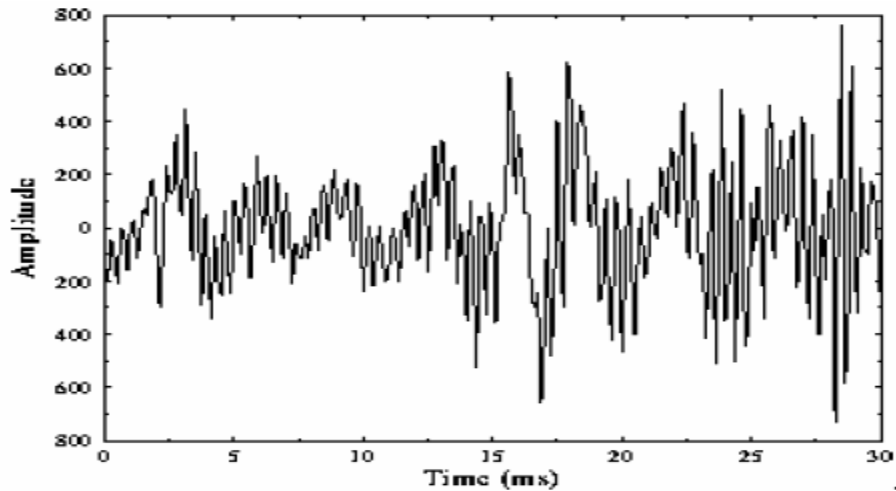


figure 1.2 Waveform of unvoiced speech

Linear Predictive Coding attempts to model the properties of speech. The mathematical model for LPC is given below in figure 1.3. Where:

- Vocal Tract is represented by $H(z)$ (LPC Filter)
- Air is represented by $u(n)$ (Innovations)
- Vocal Cord Vibration is represented by V (voiced)
- Vocal Cord Vibration Period is represented by T (pitch period)
- Fricatives and Plosives (s, f, sh sounds etc) is represented by UV (unvoiced) Air
- Volume is represented by G (gain)

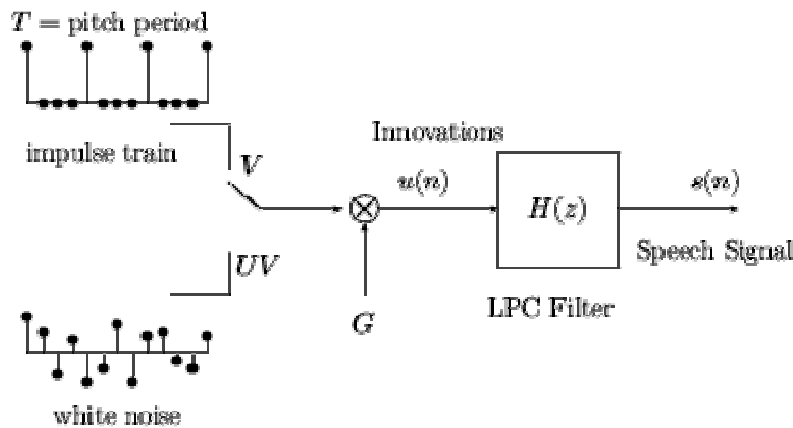


figure 1.3 Liner Predictive Coding

The equation for the LPC filter given by:

$$H(z) = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} + \dots + a_{10} z^{-10}}$$

The predicted speech will be of the form:

$$s(n) + \sum_{i=1}^{10} a_i s(n-i) = u(n)$$

$S = (s(0), s(1), \dots, s(n))$ where n is the number of samples in a frame

8Kbps CS-ACELP

We decided upon using the 8Kbps Conjugate Structured - Algebraic Codebook Excited Linear Prediction (CS-ACELP) algorithm to help conserve resources. The basic properties of CS-ACELP is it will have an output transmission rate of 8Kbps from the 128Kbps input rate (8 KHz * 16 bit) so we get a compression ratio of 16:1 which is tremendous. This compression is also achieved without any perceptible loss in sound quality thus making CS-ACELP a good choice for making our system as close to practical as possible. The system level diagram for a CELP encoder is given below in figure 1.4.

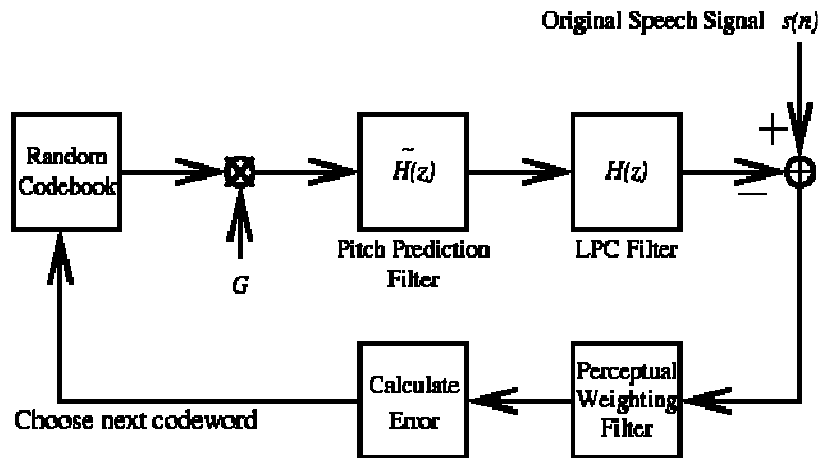


figure 1.4 CS-ACELP encoder

The 8kbps CS-ACELP compression scheme we have decided upon is the ITU recommended G.729 speech codec and is readily available on the ITU website in C. The coder accepts a 16 bit PCM linear digital signal of the incoming analog signal that was sampled at 8 KHz. The coder operates on 10ms frames of speech which corresponds to 80 samples at a sampling rate of 8000 samples per second. Every 10ms, the speech signal is analyzed to extract the CELP parameters to be transmitted through the channel. These parameters are the Linear-Prediction Filter coefficients, codebook indices and gains. The way these parameters are determined is by doing an exhaustive search of the codebook and determining the entries which minimize the error between the actual speech signal and modeled speech signal based on current and incoming frames. Instead of transferring the actual error value, only the indices of the codebook and coefficients are transmitted. Each 10ms frame is subdivided into two sub frames of 5ms each. This subdivision is done so that the spectral transition from one frame to the other is not abrupt, thus improving sound quality. The resulting output of the encoder after processing one speech

frame is the encoded parameters in the form of a bit stream containing 80 bits. Figure 1.5 shows the how the 80 bits are allocated:

Parameter	Codeword	Subframe1	Subframe 2	Total per frame
Line Spectrum pairs	$L0, L1, L2, L3$	–	–	18
Adaptive-Codebook Delay	$P1, P2$	8	5	13
Pitch-Delay Parity	$P0$	1	–	1
Fixed-Codebook Index	$C1, C2$	13	13	26
Fixed-Codebook Sign	$S1, S2$	4	4	8
Codebook gains (stage 1)	$GA1, GA2$	3	3	6
Codebook gains (stage 2)	$GB1, GB2$	4	4	8
Total				80

figure 1.5 packet allocation

At the decoder, the parameters are extracted from the bit stream corresponding to a 10ms frame. The following steps are repeated for every 5ms sub frame. First the excitation signal is constructed by adding the codebook vectors and scaling by their gains. This signal is then passed through the LP synthesis filter to reconstruct the speech. The reconstructed speech signal is then put through a high pass filter and scaled. The resulting speech signal at the output of the decoder is a 16 bit PCM linear digital signal sampled at 8 KHz. The system level-diagram for a Codebook Excited Linear Prediction decoder is shown below in figure 1.6:

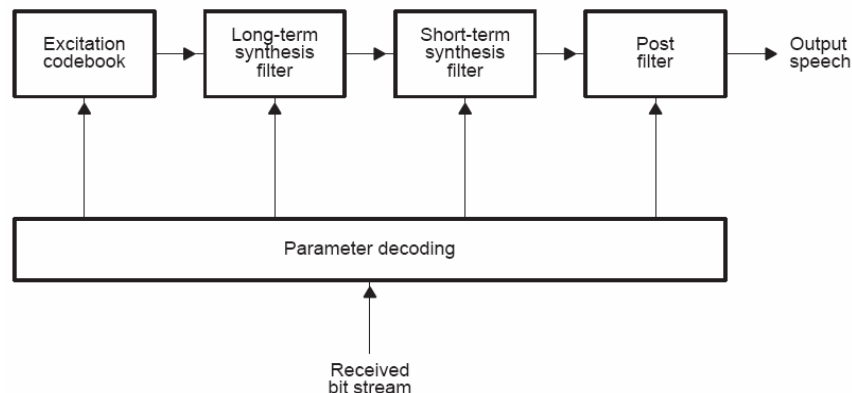


figure 1.6 CS-ACELP decoder

Algorithms: Encryption and Correction

Data Encryption

Security is a critical component of our system. If a malicious individual gains access to our voice transmissions, we like would to present him with garbled, unusable data, thereby giving an additional layer of protection to our user. The obvious solution to this problem is the use of data encryption. In general, there are two goals for any encryption algorithm: security and speed. The first goal of the encryption algorithm is obvious. We need an algorithm that can provide data protection, even if we have

intelligent cryptanalysts with large computing facilities and large amounts of time at their disposal. According to Claude Shannon, the quality of an algorithm depends on confusion – making the encryption algorithm complex and difficult to reproduce without heavy analysis, and diffusion – decreasing the statistical relationship between the bits of the input and output. For a good algorithm, changing a single bit of the input should alter each bit of the output by exactly 50%. To realize confusion, a technique known as Substitution-Box (S-Box) is used. When presented with an input, the S-Box substitutes the bits of the input for the bits of the output. To accomplish diffusion, a Permutation Box (P-Box) is used. The P-Box permutes the input bits to form the output bits. S-Box and P-Box are fundamental to many efficient encryption algorithms.

In addition to searching for a good encryption algorithm, we face additional constraints on the algorithm we choose. Our encryption algorithm is required to operate in what is essentially an embedded environment on the DSP board. The algorithm must consume limited memory resources (both data size and text size must be considered) and use only a small percentage of the processing power of the DSP chip. An ideal algorithm would provide relatively fast, secure encryption and be effectively described in a small amount of code. However, our resource constraints and requirement for good security is somewhat contrarian; implicit in strong encryption is a long, difficult to reproduce algorithm which may require either specialized hardware or significant computing power. Our limited resources will dictate which algorithms we can use.

During the design phase of the project, we explored two algorithms that were designed for embedded environments: Tiny Encryption Algorithm (TEA) and Turing. TEA is, in fact, tiny. Its implementation requires on the order of twenty lines of code for both encryption and decryption, which makes it quite attractive for implementation on the test-constrained DSP board. The basic premise of the TEA is to use add, xor, and shift operations to replace the traditional P-Box and S-Box. The usage of simple arithmetic operations makes it reasonably fast in terms of computations. Additionally, TEA is a block-based Feistel cipher and it can be effectively implemented in a single for loop. TEA operates on 64 bits of input at a time, and uses 32 rounds of calculation. DSP parallelism can be exploited by processing multiple blocks at the same time, as blocks are independent. Analysis of the sample code shows that for each block (64 bits) that we wish to encrypt, 1280 simple arithmetic (20 operations per round) operations are required. However, if we exploit system parallelism, we can complete a 64-bit encryption or decryption in 256 cycles. This computational requirement is negligible in terms of processor time.

TEA has few known weaknesses, although due to some redundancies, multiple keys are known to map to the same produced output. This effectively reduces the encryption strength of the algorithm by 2-3 bits, but for our application, TEA should provide adequate encryption. Additionally, since TEA is a block-encryption algorithm (see figure 2.1), it is less susceptible to data loss over the wireless transmission because the decryption technique is constant across all data. Since our transmitted data consists of well-defined packets, a block cipher makes a lot of sense, as the information transmitted by each packet is semantically distinct. A corrupted and, therefore dropped, packet is a highly localized error, and typically has no effect of previous or subsequent packets. Additionally, TEA does not extend the amount of data transmitted: the encrypted size of 64 bits is 64 bits. Ultimately, we chose to use the TEA algorithm as our means of

encrypting packets. TEA proved to have extremely low computational overhead and a straightforward implementation. Additionally, TEA gave us ‘atomic’ encryption, essentially allowing us to encrypt each packet separately and to view each packet as an entity independent of other data, which allowed use to achieve our goal of providing quality of service, while at the same time realizing that high interference will inevitably lead to data loss.

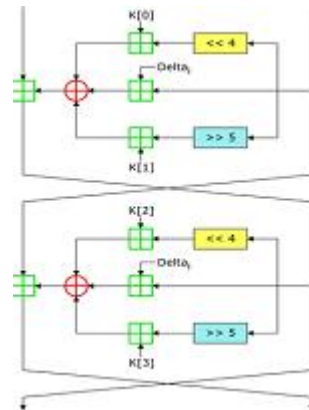


Figure 2.1 One block of Tiny Encryption

The Turing algorithm is a stream cipher, meaning that given an input, the encrypted output is dependent on not only, the input, but also the position of the data in the data stream. For example, if a stream algorithm encrypts two consecutive characters, then the two consecutive outputs will not be the same. This positional dependence is a key element of the stream ciphers since it enables them to encrypt small quantities of data without compromising security. The Turing algorithm has several additional features. The algorithm is based on a single pass through an S-box and P-box (a special P-Box called the Q-box is used) with the internals of the S and P boxes varying based on a fast, lookup table based version of a linear shift register. The advantage of the Turing algorithm is that the values used for the S and P boxes and the LSR can be calculated quickly during operation. Typically, these values must be stored in system RAM, which may prove infeasible on the DSP. However, since these values only take up approximately four kilobytes of RAM space, it may be possible to store them in the DSP RAM without incurring significant overhead.

To improve the diffusion offered by the Turing algorithm, its creators also included a four-word pad. That is, for each 32-bits of data transmitted, an additional 128 bits of padding is added. Whereas this data padding does improve security, it presents our system problems as we cannot afford this data dilution and still have our system be available to be implemented in hardware. However, our analysis of the algorithm suggests that this stage of the encryption may be removed. If the Turing S-Box and P-Box values are pre-computed and stored in RAM, Turing requires 134 cycles to encrypt a byte of data. However, if the coefficients are pre-computed, Turing requires just 34 cycles to encrypt a byte of data. These computational requirements will be acceptable for our system. One problem with the Turing algorithm is that it may run into difficulties if our transmission capabilities are poor, and ultimately, this was the reason we decided not to use it. Voice transmissions need to be relatively robust, and since ours is a multi-user

system, we must provide adequate recovery from high levels inter-user interference, which will surely be present in our system. Since Turing is a stream cipher, dropping or corrupting a cipher packet may result in the rest of the stream becoming unreadable. Our initial tests with Turing suggested that it was incapable of handling even small amounts of noise, such as would be present if two users were to broadcast simultaneously. Given its aptitude for failure in the face of our anticipated interference requirements, we decided not to use the Turing algorithm as our encryption mechanism.

Bit-wise Packet Interleave

In communications systems, interference frequently occurs in bursts of large magnitude, but short duration, creating a high probability that successive transmitted bits could be corrupted. Even if error correction is employed, there can be too many erroneous bits within the error correction region. For example, in our (12,8) error correction code, if two bits are corrupted out of any contiguous (well, nearly so, if the error spans two consecutive twelve bit fields, it may be corrected) twelve bit encoded field, then an entire packet is dropped. Thus, a short burst of interference could potentially destroy an entire packet, and if the interference was periodic in nature, it could potentially corrupt many packets. Additionally multi-bit symbol modulations, such as QPSK, can encounter a multi-bit error. If even a single symbol is corrupted, a multi-bit symbol misinterpretation could occur. A simple, and low cost solution to this problem is packet interleaving: the bitwise, systematic mixing of packet data prior to modulation. Interleaving solves the previously mentioned problem cases. If multiple packets are mixed together, then consecutive bit errors due to brief, high interference are spread across multiple packets and error correction regions, then the errors are likely to be corrected by the ECC.

Similarly, errors in multi-bit modulation schemes are also ameliorated as adjacent bits that are modulated into the same symbol are from different packets, and so badly misinterpreting a symbol has minor effects, rather than heavily corrupting a single packet. In our system, since we support only BPSK and QPSK, we interleave consecutive words in the encrypted packets (refer to the system block diagram to observe where interleaving occurs) at a time. We take two consecutive 32 bit words and shuffle them together bit by bit for two new 32 bit words, each comprised of exactly half of the parent packets. Thus, when we use QPSK for modulation, each bit of the two bit symbol comes from a different word. At the receiving end we simply undo the interleave operation and then attempt to decrypt the resulting packet.

In practice, packet interleave vastly decreases the amount of errors that we encounter when using QPSK modulation while broadcasting on a noisy channel. In fact, the use of packet interleaving masked a bug in our packet correlation. We had a small off by one bug in the correlation that effectively corrupted the last symbol of any transmitted packet. Our initial testing did not catch this bug because we were testing BPSK, and so the single bit error that was generated was fixed by the ECC. When we started testing QPSK modulation, we notice that even at low noise levels the last symbol was being corrupted enough so that a debilitating two bit error was resulting. We noticed the problem, and attributed it to bad luck, and introduced packet interleaving to solve it. Packet interleaving virtually removed the error, and it was not until several days later, when someone noticed a very obvious bug in the correlation that we realized that the

packet interleave was actually masking a bug in our own code, rather than fixing a bug created by the transmission process.

Error Correction and Checksum

Multi-user, single channel systems inherently have a large amount of interference, and indeed each user within the system can be viewed as interfering with every other user in the system. CDMA is intended to help avoid some of the user interference by spreading the transmitted data signal so that other users have a smaller probability of actually disrupting the data transmission. However, even CDMA has occasional bit errors, even if they are infrequent. Normally, low frequency errors don't cause a transmission problem: a single bit flip often has limited effect on the meaning of the transmitted data. In our case, the data transmitted is encrypted, and, as discussed above, a single bit error in the encrypted data has nearly a 50% probability of flipping each bit in the decrypted output when a good encryption algorithm is chosen. Thus, a single bit corruption during transmission will completely destroy a transmitted packet, rendering it unusable to recover encoded voice. Given that bit errors can occur frequently in wireless systems, and our packet size is 128 bits, the probability of single bit (assuming a bit error probability of .1%) is approximately 12%, meaning that roughly one out of eight packets is dropped. This drop rate is unacceptably high, so we employed a simple error correction scheme to help raise it.

In our system, we use (12,8) Hamming codes to provide error correction across packets. The (12, 8) Hamming code takes eight bits of data and adds a four bit error correcting code to the end of the byte. Hamming codes essentially number each bit of data by including the bit as a term in calculating the error correcting code corresponding to its numbering. For example, the sixth bit is included in the parity check for ECC bits one and two ($2^1 + 2^2 = 6$). Each bit of the ECC is essentially a parity check on the data bits that have the particular power of two in their numbering. When we go to correct errors, we simply recalculate the parity bits for the data and compare them to the ECC code. If the parity bits match, then no singleton error has occurred, but if some of the parity bits don't match, the combined indices of the failing parity bits indicate which bit of the data has been flipped, and we can flip this bit to correct the error. Hamming codes provide detection for single and double bit errors within the combined data and ECC, and they can correct a single bit error. In our case, this means that we encode eight bits of data as twelve bits, and we can tolerate a single bit-error across the twelve bit field. Nominally, across a 192-bit packet (the 128-bit packet extended to include ECC data) we can support up to sixteen single bit errors. Again assuming a .1% chance of a bit error, this reduces the probability of a dropped packet to .11%, which means that we drop only one of one-thousand transmitted packets, which is a huge increase in reliability over the non-error corrected transmission.

Error-correction by itself provides more than a ten-fold decrease in the number of dropped packets. However, in the presence of multiple users and high interference on the communication channel, packets will still occasionally be dropped. Hamming codes provide the ability to detect two-bit errors. However, depending on the amount of corruption within the transmitted, more than two single-bit errors could result. Thus, we employ a simple 32-bit checksum to discover whether decrypted packets are actually valid. Vocoder packets require only 80 bits of data, but our TEA cipher requires a 64-bit

input, and so each encrypted data packet is extended from 80 bits to 128 bits by the TEA algorithm. This effectively gives us 48 bits at the end of the original packet to use for whatever purpose we see fit. We chose to utilize this space for a simple 32-bit checksum. We calculate the sum of the first three 32-bit words in the packet and place the negation of their sum in the fourth word. To check for corruption, we simply add the four words. If their sum is zero, then the packet was not corrupted, but if their sum is non-zero, then we have encountered packet corruption and we drop the packet. In place of the dropped packet, we place a vocoder packet known to decode to silence.

Algorithms:

Wireless Communication using Code Division Multiple Access

Transmission takes place, once all the audio data has been ciphered and compressed, as the last module in our system. It should be noted that we chose to simulate the RF hardware on the PC instead of purchasing it for several reasons which will be detailed at the end of the discussion of our communication algorithm.

Modulation

To address the problems of reliable communication including multiple access, synchronization, and noise we implemented a Code Division Multiple Access (CDMA) system. First the incoming data has a 32 bit preamble added to the beginning of each packet, increasing its size to 224 bits. The preamble is simply 32 '1' bits for the purpose of synchronization and will be detailed shortly. The bits are modulated via M-ary Phase Shift keying (MPSK), a standard for CDMA. The reason being is that the process of spreading and despreading the signal causes major magnitude changes in the transmitted symbols. MPSK modulation divides the unit circle into M sectors called decision regions, where each contains a constellation or symbol. As an example, figure 3.1 illustrates the signal space for 8-PSK, where the circle is divided into 8 phase sectors. When a received constellation falls within a sector, no matter how near or far from the origin it is, it will still be mapped to the same symbol because of its phase.

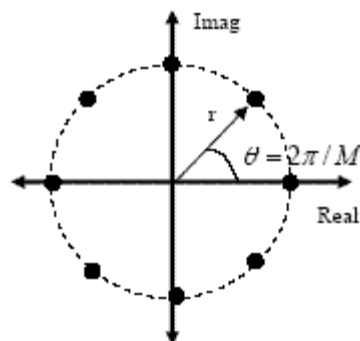


figure 3.1

The most commonly used PSK schemes in CDMA, which we provide, are Binary PSK (BPSK) and Quadrature PSK (QPSK). In BPSK each bit can be mapped to a real,

(positive or a negative value), usually 1, where as in QPSK each set of two bits is mapped to a point with both a real and imaginary component, as its name suggests. In our implementation we do not include an actual carrier wave to phase shift. The reason being is that, should this system be implemented for real use, a proper FCC band and bandwidth would need to be assigned. Since that topic is outside the purpose of the project we simply use the method mapping bits to points on the real-imaginary axis. Mathematically both are correct, so for our purpose, this is an acceptable substitution.

CDMA and Spread Spectrum

Once modulation has been completed we take our packet of symbols and perform the CDMA algorithm on it. CDMA, as the name suggests, is an algorithm to allow multiple users access to a communication channel with minimal losses due to interference between those users (co-channel interference). Our implementation utilizes Direct Sequence Spread Spectrum (DSSS) in order to achieve the CDMA, since CDMA is structured around spread spectrum systems. A PN sequence is generated using a preferred irreducible polynomial and a linear feedback shift register. The size of the register is the amount by which the signal is spread. In our test data, we generated codes from lengths 20 to 31 for comparison purposes to illustrate that the higher the value, the more users we can support. Each symbol is multiplied by a different PN sequence (also called a chip sequence) for each user. This being completed, we effectively have spread out the signals Power Spectral Density (PSD). Now instead of transmitting a few frequencies at high power we transmit several at a lower power. Figure 4.1 illustrates this effect and also shows that you could choose an amount to spread such that you could hide your signal below the noise floor.

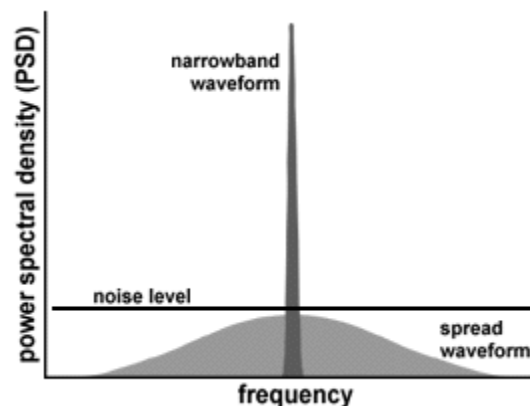


figure 4.1 An illustration of Spread Spectrum

To someone who does not have the code, the signal now looks like noise thanks to the pseudo-randomness of the code. The spread signal is then simulated as being transmitted over an AWGN channel where the signal to noise ratio (SNR) is specifiable. Noise is added to each user while all their data interferes (sums up). Without multiple access, the co-channel interference would be enough to corrupt each user's transmission. Rather than users waiting for a turn to transmit data (talk), they can do so whenever they please since users are unique by their codes. Once the data is received, we correlate for the preamble so that we can find where to begin decoding. With a 32 bit preamble,

BPSK, and a spread factor of 31, our maximum spreading, we receive 992 symbols as the encoding for the preamble. This maximum score should be more than high enough to find peaks when data is buried in noise and interference. Once the signal is synchronized with the receiver properly it can begin the decode process

Decoding is fairly simple for the AWGN channel. To implement it, we simply apply a matched filter to each spread factor sized set of symbols. The filter is matched to that users PN sequence, removing the code, despreading the signal, and attenuating noise. The matched filter given in figure 5.1 has a chip sequence C_k , a received signal Z_k , and a spreading factor of L .

$$\sum_{k=0}^{L-1} c_k z_k$$

figure 5.1 Our Matched Filter

Since the matched filter is simply an inner product, this computation is not as expensive for this channel as a receiver filter for a random channel could be. Once the data has been decoded it can be mapped back to bits through the decision device. The decision device makes a choice by assigning the received symbols to the closest actual constellation points through a minimum distance function. Clearly, if the noise power was enough, we could get an error in decision making. Once the data is mapped back to bits, it can be passed back to the other modules to be processed as well.

The other modules are dependent upon proper communications algorithms to give the best performance in terms of error so that they can produce meaningful data. This being said, choosing the best codes for use with the CDMA algorithm are important. Previously it was thought that orthogonal codes were the best for this application. If we have a code for one user and we received data for the other user when we attempt to match filter, the output will be zero since the inner product of two orthogonal vectors is zero (see figure 6.1). The problem becomes apparent when considering time skews. If the two given codes are even slightly delayed relative to one another, the dot product is no longer zero and thus the codes do not appear orthogonal. This implies then that the matched filter would essentially produce the incorrect data, useless to the next blocks. In our implementation after correlating for the first packet, we see if its checksum is valid since the probability of having a high score *and* having a valid checksum is extremely close to zero.

$$\sum c_k \tilde{c}_k = 0$$

figure 6.1 Orthogonal codes

The solution to the problems of orthogonal codes is PN sequences. Since they don't depend on orthogonality, PN sequences are more robust, implying they are better for CDMA use. A time skew is similar to having an error. If we have five zeros and then 25 symbols encoded by the PN sequence using a spread factor of 30, obviously that symbol can still be decoded because the spreading repeats one symbol 30 times, so we can still get a high score as the output of the matched filter, which is desirable. Since the codes for each user are unique, correlating two codes together should not return a high

score, meaning that the interference from other users can be removed without the codes having to be orthogonal; however, this isn't always guaranteed.

As we researched and ran tests we found that randomly chosen PN sequences were not the best for use in our application. The reason being is that randomly chosen PN sequences are just output from a shift register. There is no guarantee that the cross correlation between PNs will always be low all of the time for time shifted sequences. Although the time shifted autocorrelation is guaranteed to perform better over orthogonal codes, the cross correlation does not always give the best performance. Researching asynchronous communication lead us to find a better type of PN sequence, Gold codes, which have special properties such as provably low cross correlation peaks.

Gold codes can be created by generating two PNs using a generator polynomial in a shift register and then using modulo two additions (xor) to combine the two sequences together. Gold codes have cross correlations with peaks that are no larger than half the length of the code. They are better than orthogonal codes since they are built out of PN sequences which are proven to be better for time shifts but they are better than regular PNs in that they are built to have these specific properties that make them more reliable than normal PNs in terms of cross correlations. The randomness of where and how high the cross correlation peaks occur is removed, which is what makes Gold codes so useful. They are just as easy to generate as PNs but are much more resistance to time shifts.

Consider the following figures, figure 7.1 and 7.2. In figure 7.1 we have the cross correlation between two normal PN sequences used in our system. As you can see, there are very large peaks at random intervals in the cross correlation. What this would mean in terms of communication is that if two users were interfering and a time delay existed between them, the correlator and matched filter could have trouble distinguishing between the two if the time shift was enough to cause the matched filter to fall upon one of these peaks. In other words the receiver could receive corrupted data or if the cross correlation is high enough, the wrong user could be seen. In figure 7.2 we have the cross correlation of two gold codes. It is clear that the cross correlation peaks are low, as previously stated which means less co-channel interference will be present. In addition we should never enter the case where we decode the wrong user. For more performance analysis using Gold codes versus normal PN sequences, refer to the test results section.

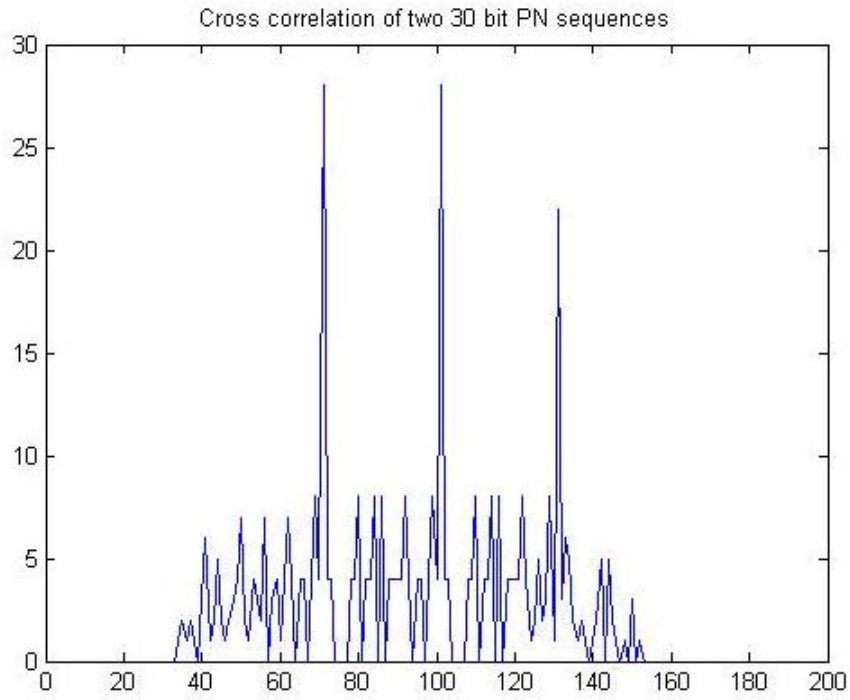


figure 7.1 PN sequence cross correlation

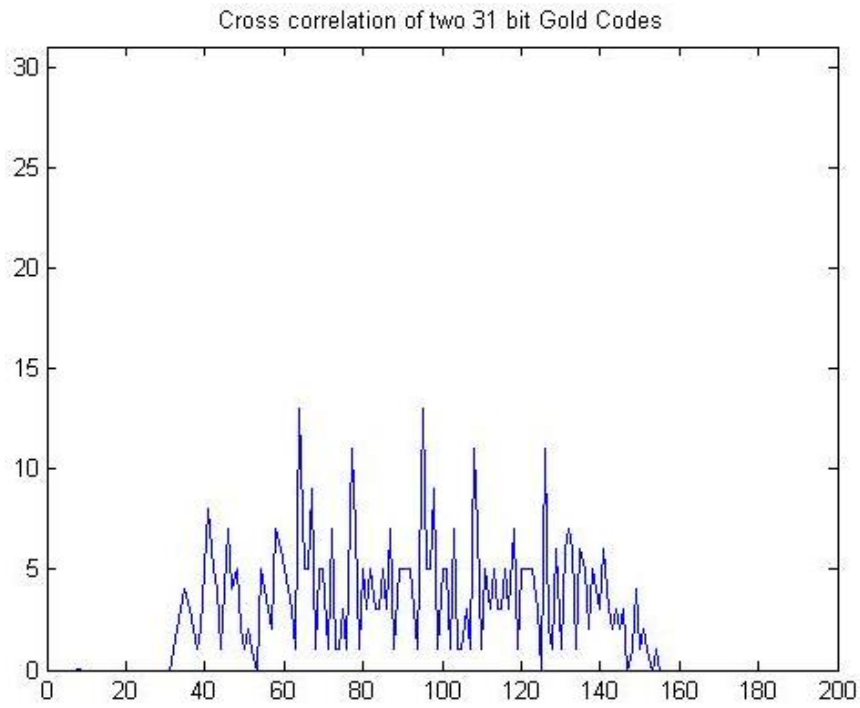


figure 7.2 Gold code cross correlation

Additional Notes

One of the downfalls of our wireless algorithm however is the channel assumption. In reality, the wireless channel would be modeled by a fading channel that has multipath propagation. The fading channel has a random impulse response so it's difficult to design a good receiver for it using simple matched filter techniques. As figure 8.1 illustrates, multipaths are created from the transmitted signal bouncing off environmental objects such as the ground, atmosphere, buildings, etc. This causes inter-symbol interference (ISI), which cannot be addressed without proper equalization. In the interest of time and given the already large complexity of the project, ISI could not be properly addressed within our system. The purpose of this system is to be a model of what an actual system might need to do in order to resolve the problem we identified previously.

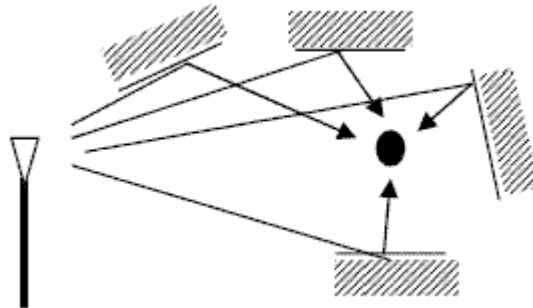


Figure 8.1 Multipaths

As for the reasoning behind transmission simulation, difficulty among other items could explain several issues. After much time searching for RF hardware to implement the wireless link we came to the realization that learning to use hardware would be a project in itself. In addition, all the boards we looked at would not give us access to the analog signals we wanted to process; as a result, we would have had to buy several cards to interfere on the same channel that already implemented a modulator, spread spectrum, and CDMA. If there was an existing board with all these specifications that we could buy for consumer use, then we would have had the communications portion of our project completely finished, invalidating the purpose of the project. We also referenced a previous CDMA paper that used hardware in their project for communication and noted that they were unable to complete theirs because of the nightmare debugging the hardware became. It was in the best interest then of our project in terms of time, money, and success that we instead simulate the wireless transmission in software.

Available Software and Issues

There are several readily available implementations of the G.729 vocoder. One of the first that we found was the VoiceAge G.729 Open Initiative, which is based on the ITU recommended G.729 vocoder. VoiceAge's implementation works well in the sense that we are able to achieve the desired 16:1 compression ratio. In a test case, we were able to compress a 256Kb PCM file down to about 16Kb. The code takes in a PCM file as its input and outputs a bit stream file. The problem with this implementation is that it has to open and read the input file from a disk and thus changes have to be made if we are going to port it to the EVM. The other problem with VoiceAge's implementation is that they obfuscate their main codes by pre-compiling them as Windows libraries. The only codes that are literally available in C are the main driver files where functions are called. The inner workings of the functions are hidden. Since we foresaw that the Windows libraries could pose some problems when ported to the EVM, we then decided to discard VoiceAge's implementation and use something else.

The next implementation we tinkered with was ITU's own G.729 vocoder, specifically G.729 Annex A which is a reduced complexity implementation of the original G.729. The main issue with this implementation is that the output format is not the one desired. In ITU's G.729A, 1 output bit is represented by 2 bytes, thus instead of getting a 16:1 compression ratio, we were getting something on the order of 1:1. After realizing this fact, we changed the bit stream formatting so that 1 output bit is represented by 1 bit accordingly. The other issue we realized is that G.729A is a fixed-point implementation of the algorithm but to fully utilize the floating point capabilities of the EVM, we later decided to use ITU's G.729 Annex C floating point implementation instead.

Once we finalized the voice compression scheme to use, we proceed to make changes necessary so that the code would work on the EVM. Since we can't open a file to read data from the EVM, we needed to feed the data from the PC. Thus we implemented two components to making the EVM code work. First is the PC side which contains mainly the HPI transfer calls (from Lab 3) and some code that extracts data from file and also writes data to file. The second is the EVM side which contains the vocoder and some function calls to initiate HPI transfer (from lab 3).

Once the vocoder code was on the EVM instead of the PC, it no longer made function calls to reading and writing to files. Instead all the data the vocoder needed was available on the SDRAM after HPI transfers are done.

In addition to the vocoder, we also used an AWGN channel model provided on a public web page (see references) and added some changes to make it more applicable for our purposes. This portion of provided C code worked without any major issues.

Working with the EVM

The biggest hurdle while working with the EVM was error messages that we didn't really understand. The biggest mistake we made while working with the EVM was to create our own project file in Code Composer instead of using a template project from one of the labs. In doing so, we encountered 'Referencing Errors' in CCS and CCS indicated that standard functions and libraries like 'printf' and 'math.h' were giving out the errors. Once we used the project template from lab 3 and moved the vocoder codes into the template, the problem was resolved. The problem was caused probably by some settings and linking that weren't done properly as in the labs. It would have saved us a lot of time if we had realized it sooner or somewhat addressed during the course of the labs.

Memory Needs and Speed

The following are the memory requirements of the EVM:

- Input Speech: 80KB (128Kbps with an average length of 5 seconds)

We allocated nearly 4MB on the SDRAM so we can practically support up to 4 minutes of speech

- Bit Stream: 5KB (assuming we always achieve a 16:1 compression ratio)

We allocated nearly 200KB on the SDRAM

- Output Speech: 80KB

We allocated nearly 4MB on the SDRAM so we can practically support up to 4 minutes of reconstructed speech

- Code: 257KB

Before doing any kind of optimization, the runtime for encoding 1 frame of speech was roughly 167ms which was too slow for any type of real-time processing (which requires about 15ms). Decoding naturally takes less time (nearly ½ the encoding time) because all that needs to be done is to match up the parameters to extract the speech whereas in encoding, most time is spent on determining which parameters will minimize the error between predicted speech and actual speech.

EVM Optimizations

We did several levels of optimizations to further speed up the processes on the EVM. To document the speed improvements we achieved after the different levels of optimizations, we profiled the EVM running on several configurations.

- Firstly there was no optimization at all; no paging was done, and since the vocoder code is too big to fit on the ONCHIP_PROG, all the codes reside in the slower SBSRAM_PROG.
- Secondly, we did paging from the SDRAM but the codes still reside in the slower SBSRAM_PROG.
- Finally we tinkered with the linker and filled the ONCHIP_PROG with most of the code and the remaining codes will be put on the SBSRAM_PROG. We did this by profiling the vocoder code using 'gprof' on UNIX and by doing so we could determine which functions are called most often. Thus, we know which functions to move to the faster on-chip RAM. So even though the functions are

called often, now that they're on faster on-chip memory, the whole processing time will significantly be reduced.

In addition to the above steps, we also utilized the built in optimization options in CCS. The optimization levels we employed were:

- Local (-o1)
- Local (-o1) with 'printf's removed from EVM code. These 'printf's calls were on the EVM mainly for debugging purposes and as an indicator that the program is running. These 'printf's were called every time a frame is processed. So it can be seen that for a five second file, there will be 500 frames to be processed and 500 times the 'printf' is called. Eliminating the 'printf's prove to be very cost effective.
- File (-o3) with loop unrolling (as done in lab 3) and removed printf's

Optimization Results

7.81KB speech file (50 frames, 0.5 second)

Optimization Level	<ul style="list-style-type: none"> • Code in slow RAM • No paging • Local (-o1) 	<ul style="list-style-type: none"> • Code in slow RAM • With paging from SDRAM • Local (-o1) 	<ul style="list-style-type: none"> • Code in fast RAM • With paging from SDRAM • Local (-o1) • Printf's removed 	<ul style="list-style-type: none"> • Code in fast RAM • With paging from SDRAM • Printf's removed • Loop unrolling • File (-o3)
# of Cycles	Encode: 203,545,336 Decode: 55,678,055	Encode: 202,881,966 Decode: 55,676,378	Encode: 102,285,466 Decode: 26,544,863	Encode: 70,052,720 Decode: 16,789,101
Time (seconds)	Encode: 8.142s Decode: 2.227s	Encode: 8.115s Decode: 2.227s	Encode: 4.091s Decode: 1.062s	Encode: 2.802s Decode: 0.672s
Frame Processing Rate (frames/s)	Encode: ~ 6 frames/s Decode: ~ 22 frames/s	Encode: ~ 6 frames/s Decode: ~ 22 frames/s	Encode: ~ 12 frames/s Decode: ~ 47 frames/s	Encode: ~ 17 frames/s Decode: ~ 74 frames/s

27.2 KB speech file (172 frames, 1.74s)

Optimization Level	<ul style="list-style-type: none"> Code in slow RAM With paging from SDRAM Local (-o1) 	<ul style="list-style-type: none"> Code in fast RAM With paging from SDRAM <ul style="list-style-type: none"> Local (-o1) Printfs removed 	<ul style="list-style-type: none"> Code in fast RAM With paging from SDRAM <ul style="list-style-type: none"> Printfs removed Loop unrolling <ul style="list-style-type: none"> File (-o3)
# of Cycles	Encode: 707,814,175 Decode: 194,134,264	Encode: 361,601,969 Decode: 98,981,571	Encode: 244,450,121 Decode: 59,356,437
Time (seconds)	Encode: 28.313s Decode: 7.765s	Encode: 14.464s Decode: 3.959s	Encode: 9.778s Decode: 2.374s
Frame Processing Rate (frames/s)	Encode: ~ 6 frames/s Decode: ~ 22 frames/s	Encode: ~ 12 frames/s Decode: ~ 43 frames/s	Encode: ~ 17 frames/s Decode: ~ 73 frames/s

EVM Speed Improvement

As you can see from the above tables,

- ***moving from SBSRAM_PROG to ONCHIP_PROG***
 - Encode: 6 frames/s to 12 frames/s
 - 2X speed improvement
 - Decode: 22 frames/s to 45 frames/s
 - 2X speed improvement
- ***-o1 to -o3 with loop unrolling***
 - Encode: 12 frames/s to 17 frames/s
 - ~1.5X speed improvement
 - Decode: 45 frames/s to 73 frames/s
 - ~1.6X speed improvement
- ***Overall improvement***
 - Encode: ~2.8X
 - Decode: ~3.3X

Meeting Vocoder Real-Time Requirement

The real time requirement for the vocoder is roughly 10 to 15ms to encode a frame of speech (decoding takes about 5ms per frame). As can be seen from the table above, the time it takes to encode a frame before any optimization methods is roughly 167ms. In the end after all the optimizations have been done, the encoding time for a frame is reduced significantly to about 60ms. Therefore there is a 100ms improvement, yielding roughly a 64% speed improvement from the initial setup on the EVM.

Demonstration and test data

The demonstration system consists of three somewhat distinct portions: the graphical user interface, the EVM side speech vocoder and error correction processing, and the PC side wireless simulation. The GUI was written in using Microsoft Foundation Class in Visual C++, and serves mainly as ‘glue’ to connect the primary system components together. The GUI additionally manages the HPI transfers that move audio data to and from the PC, as well as serving as a driver program for the PC side wireless simulation. The GUI presents the user with a variety of specifiable options: the number of users in the system, the user whose transmission is to be listened for, transmission delays for broadcasting users, the amount of noise to be added to the communication channel, the type or spread spectrum encoding to be used, and the type of modulation. Once the user has selected the desired options and specified appropriate input (16 bit pulse code modulation encoded audio file) they simply press the “communicate” button and the PC and EVM cards handle the rest.

In general the demo simulates the full functionality of our system, and is intended as a test-bench for users who want to determine which type of CDMA communication is best for their potential application. The demo gives the user the ability to specify a valid input vector, run it through a simulated sub-optimal (with the degree of sub-optimality controlled by said user) communications medium and then listen to what our system would have played back had the transmission actually occurred. In addition, the user is given the ability to specify multiple parameters, which should accurately model the operation environment of their system.

The wireless simulation provided by the GUI consists of several phases. First, the GUI determines the various parameters selected by the user and prepares to process the voice transmission accordingly. Next, each user specified input file is run through the EVM encoder, which runs the CS-ACELP encoder, encrypts the encoded data, and adds error correction. The transmittable data is then passed back to the PC, where it is modulated, spread and then written to disk. If the user has specified any noise or delay, they are added to the transmittable files at this point. Then, the files are “interfered” (really the data contained within the files is simply added together). At this point, the encoded signal represents the actual transmission that a receiver would see. The signal is then despread and demodulated according to the listener that the user specified. We then decrypt the data and run the checksum, replacing bad packets and then pass the data to the EVM. We finally decode the CS-ACELP packets. When all packets are processed, we return the decoded PCM audio to the PC via an HPI transfer, where it is recorded to disk.

We analyzed several segments of test data after running them through our system and confirmed it’s validity. The following is a small segment of test cases we ran for a PN sequences of size 20, 30, and Gold codes of size 31 using both BPSK and QPSK (note: noise tests ranged from -8dB to 10dB):

- One to four users with no noise and no delay
- One to four users with noise and no delay
- One to four users with no noise and synchronous delay
- One to four users with noise and synchronous delay
- One to four users with no noise and asynchronous delay
- One to four users with noise and asynchronous delay

What we expect to happen when thinking about the communication theory outlined earlier is that as we add more users to the channel, Gaussian noise is negligible since the interference becomes more powerful. In addition, because the PN sequences were not able to handle asynchronous delays as well as the Gold codes, the cross talk should show up more between users, which would degrade the quality of each individual user. Figure 9.1 shows two input waveforms that were sent through the intercom system. They were transmitted asynchronously with 10dB of transmission power using both Gold codes and length 30 PN sequences. Figure 9.2 shows the received waveforms when we attempted to decode the first file, user 1. Compare the waveforms in figure 9.2 with those in 9.1. Notice that the attempt that used Gold codes (first waveform in 9.2) is very similar to the original user 1 data. Now refer to the second waveform in figure 9.2. It's clear that the waveform appears to be severely degraded in comparison with the original. In addition you can see that parts of user 2's data have shown up in our decoded user 1 file (after the 8000 sample mark). This further illustrates how PN sequences are not always ideal for asynchronous communication in comparison to Gold codes because they can potentially have high cross correlations that allow cross talk to appear in decoded data.

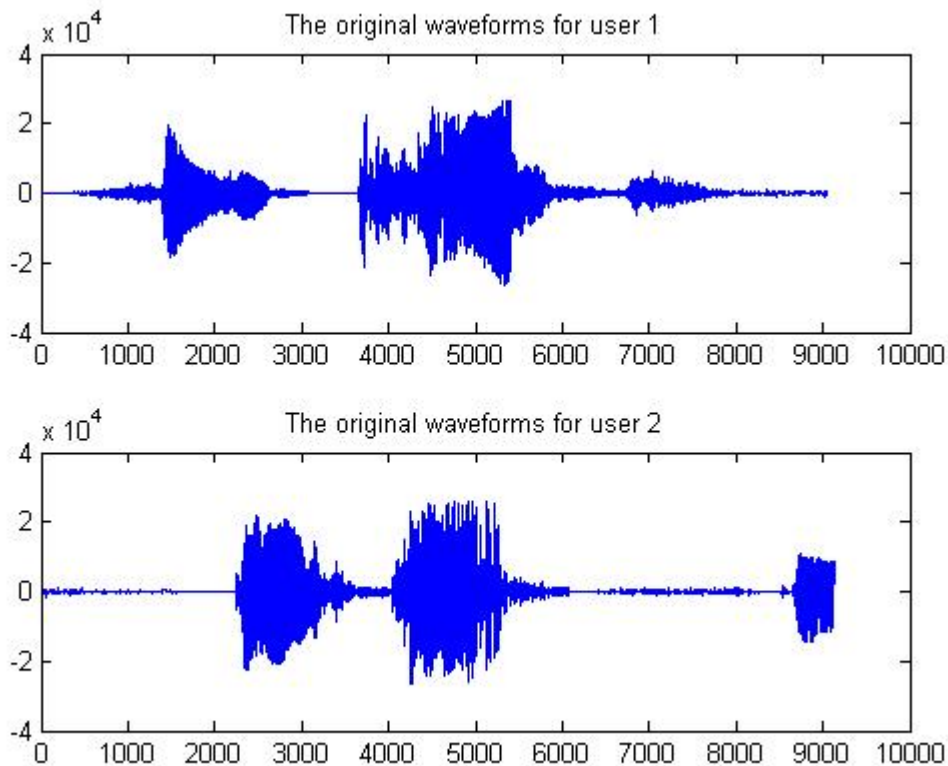


figure 9.1 The original waveforms

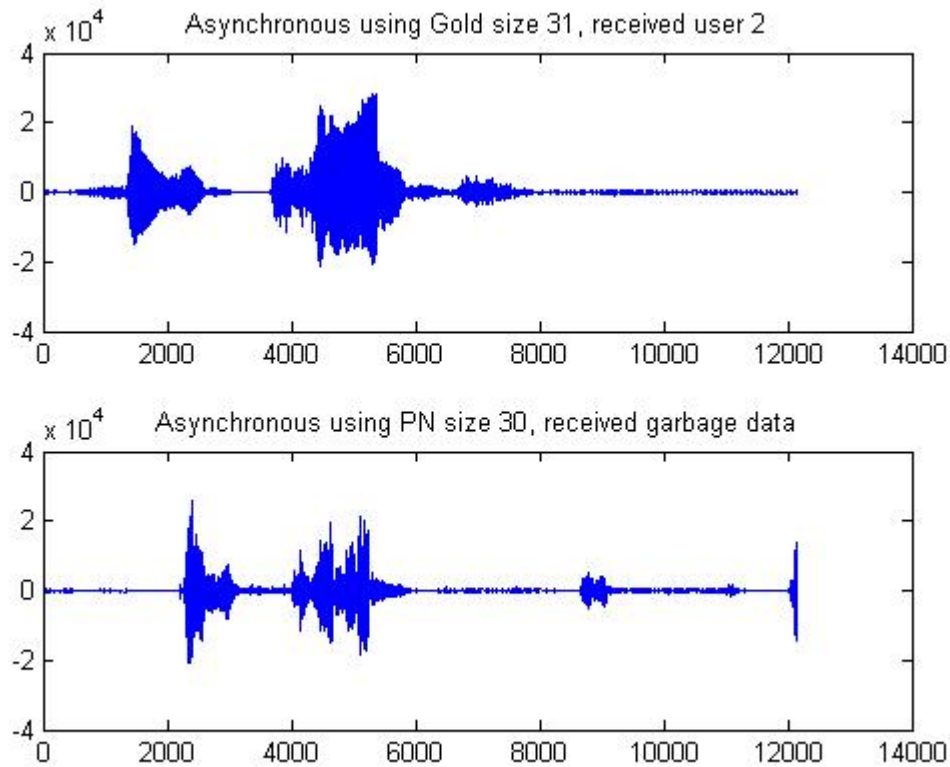


figure 9.2 Using Gold codes (top) and normal PNs (bottom)

We were able to support all four users transmitting asynchronously in our test data given the following constraints:

- PN length 20 using BPSK at a minimum of 3dB transmission power
- PN length 20 using QPSK at a minimum of 6dB transmission power
- PN length 30 using BPSK at a minimum of 4dB transmission power
- PN length 30 using QPSK at a minimum of 6dB transmission power
- Gold length 31 using BPSK at a minimum of -0.1 dB transmission power
- Gold length 31 using QPSK at a minimum of 5dB transmission power

Clearly, we need the least amount of power using Gold codes because they perform the best in the face of time shifts and use a large spreading factor. Using Gold codes with a large spreading factor allowed us to achieve our original project goal of transmission below the noise floor but with the added caveat of doing so asynchronously. Originally we had only planned on being able to support synchronized users reliably at or below the noise floor. Adding Gold codes gave us the extra boost needed to surpass our previous project goals. If the system requires more users to be implemented, the spreading factor would have to be increased in order to continue operation at the noise level.

In order to implement our system in hardware the following minimum bit rates would be required:

- Using 20 bit PN sequences, BPSK: 0.473 Mbps; QPSK: 0.945 Mbps
- Using 30 bit PN sequences, BPSK: 0.709 Mbps; QPSK: 1.417 Mbps
- Using 31 bit Gold codes, BPSK: 0.732 Mbps; QPSK: 1.464 Mbps

Through all this testing, our model shows the optimum solution to our problem of secure, reliable communication would be implementing voice compression, error correction (reliability), encryption (security), and CDMA (multi-user) where the transmitter implements 31 bit Gold codes(reliability) using BPSK modulation. The simple specs of our system would then be antenna gains of approximately 0 dB and a transmission rate of 732 kbps.

Our Conclusion: It's a success!

In our project, we successfully demonstrated reasonable asynchronous audio transmission of a single user in the presence of multiple other asynchronously transmitting users and high channel noise, to the point of broadcasting at the noise floor, which was far lower than we thought that we could achieve. Thus we accomplished all of the goals set forth by our amended project proposal. Indeed, we were actually quite pleased with the outcome of the project in terms of both its robustness and its customizability; however, it is important to note that, while we have realistically modeled a real system, we have not actually implemented a true broadcast system. A future goal for our project would be to take the EVM processing stack, which is composed of the vocoder, checksum, error correcting codes, and packet interleave, and integrate it with a wireless backend to produce an operational intercom working in real time. Primarily, this would consist of two phases: first, we would need to develop suitable wireless hardware, and second we would need to integrate the EVM card with the wireless hardware, probably by using the extra McBSP ports. If we had another semester and RF circuit experience, this would be the logical next step in building our product.

Several facets of our project went well, while we, quite honestly, failed in some respects. Our obvious success is that our project was able to handle virtually every test case that we tried, beyond of course, broadcasting far beneath the noise floor with several users. A byproduct of the project, which ought to be considered a major success, is that we learned a great deal about real world communication and embedded DSP programming. However, the entire project was not free of problems. We had a lot of issues with working as a team, and there were several shouting matches and some pretty long nights of work. Additionally, we had some trouble initially selecting a project and then, after we had selected a suitable project, actually getting started on it. However, we were able to pull together in the end and complete a difficult yet interesting project. To improve our cohesiveness, we should have been more accepting of each other's needs and skills. We recommend the following to future 551 students: determine your project early (making sure that it is within reason), start working early, and work together.

References

- [1] <http://www.palowireless.com/bluetooth/docs/BDouglas.pdf>
- [2] <http://www.data-compression.com/speech.shtml>
- [3] http://akhisar.sdsu.edu/abut/EE658/CHAP10_2004.pdf
- General working principles and description of common voice encoding schemes
- [4] http://www.eetkorea.com/ARTICLES/2001MAY/2001MAY22_DSP_AN1.PDF
- Specific description of the ITU G.729 vocoder implementation on a TI TMS320C54x EVM including ASM optimizations
- [5] <http://www.itu.int/rec/recommendation.asp?type=items&lang=E&parent=T-REC-G.729-199809-I!AnnC>
- C Source code for ITU G.729 Annex C vocoder
- [5] Claude E. Shannon, "Communication Theory of Secrecy Systems", Bell System Technical Journal, vol.28-4, page 656--715, 1949. – a primer on general encryption
- Hamming codes:
- [6] http://www.nuvisionmiami.com/books/asm/workbook/error_correcting.htm
- [7] Turing: a fast stream cipher. Greg Rose, Philip Hawkes. Qualcomm Australia, November 2002. eprint.iacr.org/2002/185.pdf
- [8] 18-450 Digital Wireless Communication course notes
Handouts 11,20,29,43
- modulation, DSSS, CDMA, matched filter techniques, SNR/BER/CDMA equations
- [9] Digital Communications: Fundamentals and Applications, second addition by Sklar
- modulation, DSSS, CDMA
- [10] R. L. Pickholtz, D. L. Schilling and L. B. Milstein "Theory of Spread Spectrum Communications -- A tutorial", IEEE Transactions on Communications, COM 30, No. 5, May 1982, pp. 855-884. – a spread spectrum primer
- spread spectrum and Gold codes
- [11] R. Gold, "Optimal binary sequences for spread spectrum multiplexing," IEEE Trans. Inform. Theory, vol. IT-13, pp. 619-621, 1967. – original Gold code paper
- Gold code creation and properties
- [12] home.netcom.com/~chip.f/Viterbi.html
- AWGN C code