

Appendix E - UNIVERSITY HONORS PROGRAM
SENIOR PROJECT - APPROVAL

Name: Jacob DePriest

College: Engineering Department: Electrical & Computer

Faculty Mentor: Dr. Paul Grilly

PROJECT TITLE: A Wireless Voice
Transmission System

I have reviewed this completed senior honors thesis with this student and certify that it is a project commensurate with honors level undergraduate research in this field.

Signed: Paul R. Grilly Faculty Mentor

Date: 5/3/04

General Assessment - please provide a short paragraph that highlights the most significant features of the project.

Comments (Optional):

Mr. DePriest designed a wireless transmitter/receiver that incorporated the following features:

- 100% 200K
- Project team
- PSC

He conducted some tests of PSC using Hamming, convolution codes and a 1/2 rate code. He also conducted some tests of a 1/2 rate code using a 1/2 rate code. He also conducted some tests of a 1/2 rate code using a 1/2 rate code. He also conducted some tests of a 1/2 rate code using a 1/2 rate code.

A Wireless Voice Transmission System

Honors Senior Design Project

Jacob A. DePriest

5/3/04

Contents

Abstract.....	2
1 Background	3
2 Introduction.....	4
Figure 2.1 – Overview of design	5
3 ADC – Analog to Digital Conversion	6
Figure 3.1 – Sampling an analog signal	6
Figure 3.2 – Microphone Setup	7
4 DAC – Digital to Analog Conversion	8
Figure 4.1 – Speaker Setup	9
5 Data Transmission	9
5.1 Transmitting Bits on the TRF6901	10
5.2 Increasing Transmission Speed.....	10
5.3 Bandwidth Considerations	11
5.4 Channel Speed	12
5.5 Word Transmission.....	12
5.6 Effective Packet Transmission	13
5.7 Packet Radio	14
5.8 Storage / Memory	15
5.9 What We Implemented - Short Recorded Tones.....	15
5.10 Future Transmission Enhancements - Reduction of Data Rate.....	16
6 Error Correction.....	17
6.1 Convolutional coding.....	18
6.2 ARQ (automatic-repeat-request) system.....	18
6.3 Block coding (Hamming codes in particular)	19
Figure 6.3.1 - Probabilities of word errors vs. probability of bit error.....	21
Figure 6.3.2 - Coding gain vs. initial probability of bit error.....	22
7 Results	23
8 Conclusion	24
9 References.....	25
Appendix A – TI MSP-TRF6901-DEMO Schematic.....	26
Appendix B – Analog Devices DAC8512	27
B.1 – Pin Configuration	27
B.2 – Control Logic Truth Table.....	27
B.3 – Timing Diagram.....	27
Appendix C – Budget.....	28
Appendix D – Pictures.....	29
D.1 – Board Layout	29
D.2 – Project Hardware	30
Appendix E – Digital Communication Code	31

Abstract

There are three main components in digital, wireless, voice communication systems: taking an analog signal of varying values and changing that into a stream of ones and zeros (Analog to Digital Conversion-ADC), processing and sending the new digital signal, and then recreating the original analog signal on the other end (Digital to Analog Conversion-DAC). In this project we used a TI MSP-TRF6901-DEMO kit and an Analog Devices DAC8512 digital to analog converter to implement a digital, wireless communication system with forward error correction. Hardware limitations prevented the system from transmitting and receiving satisfactory voice data. The system is capable of reproducing simple tones under 1.5 KHz. The communication system operates in the 900 MHz ISM band and transmits packet data using FSK modulation.

1 Background

(Jacob DePriest)

As the demand for mobile technology has increased rapidly over last several years, the need for reliable wireless voice communication has become one of the main focuses of the technological world. From cell phones to digital radio, wireless voice data is becoming an integral part of our society and will continue to affect many areas of our lives.

Certain technologies such as two way radios, many home telephones, and standard FM and AM radio still rely on analog communication to transmit voice wirelessly. However, this method does not ensure the reliable transfer of data and leads to less than clear voice communication. Digital communication alleviates many of these issues, but brings with it a whole new set of problems.

Digital voice communication is composed of three main areas: taking an analog signal of varying values and changing that into a stream of ones and zeros (Analog to Digital Conversion-ADC), processing and sending the new digital signal, and then recreating the original analog signal on the other end (Digital to Analog Conversion-DAC). The ADC and DAC processes must be adjusted to the environment that they are being used. In voice communication, the ADC step must take a sufficient number of samples per second in order to be able to recreate the signal on the other end. It must also take the samples with some amount of precision for the same reason. The DAC step must be able to interface with some output, usually a speaker of some kind, by taking the received digital signal and converting it into an analog signal that can drive the speakers.

The majority of variation among different wireless digital communication schemes is located in the second step of the overall process: processing and sending the

digital signal. Digital communication provides the opportunity for error detection and correction of transmitted data and allows very reliable data transfer because of these prospects. By using error correction algorithms, the receiver knows when the data it received is bad and can correct it in certain circumstances to produce the exact transmitted signal even though it was not received correctly. There are many ways to add information to the digital signal before it is sent in order to correct and detect errors in the received signal.

There are many issues to be overcome in each of the three main steps of wireless digital communication. This report will focus on the analog to digital conversion of voice and the digital to analog conversion of the received signal back into an audible voice signal.

2 Introduction

(Jacob DePriest)

We set out to implement a full duplex, digital, wireless voice communication system. This project incorporated all three of the digital voice communication areas described above and required us to exercise our knowledge of digital communication learned in Communication System Design and previous communication classes.

Our main platform for development was the TI MSP-TRF6901-DEMO kit which incorporates the MSP430F449 microcontroller and the TRF6901 transceiver. The MSP430F449 has a built in 12-bit ADC that can sample at over 200 thousand samples per second; the external microphone was connected to the ADC. We used the Analog Devices DAC8512 as our external DAC which the speakers are connected to as seen in

Figure 2.1 below. The MSP430F449 was programmed using basic C under the IAR Embedded Workbench software application (www.iar.com).

The TRF6901 operates in the unlicensed 868-MHz and 915-MHz bands which are used for scientific, industrial, and medical purposes. It transmits data using FSK (Frequency Shift Keying) or OOK (On-Off Keying). The TI MSP-TRF6901-DEMO kit comes preconfigured to operate between 902 MHz and 917 MHz based on a user-configurable dip switch and is set to use FSK modulation. The output transmission rate of the demo boards is preset to 38.4 kbps with a maximum achievable rate of 64 kbps. The boards are powered by 2 AAA batteries each and have 4 programmable buttons. The boards can be seen in Appendix D.

This report details the challenges we faced in implementing a full duplex, digital, wireless voice communication system, the solutions we came up with to address some of these challenges, and the final results of our endeavors. We will also discuss possible solutions to problems that were not solved during this process. Each of the components of the system will be discussed in detail and we will clarify our results.

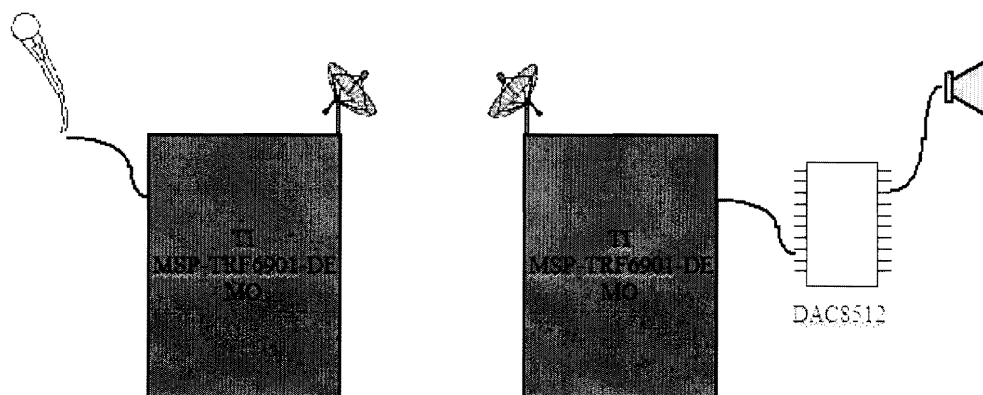


Figure 2.1 – Overview of design

3 ADC – Analog to Digital Conversion

(Jacob DePriest)

As stated above, the MSP430F449 contains a built-in 12-bit ADC which was used in the conversion of the analog voice signal received from the microphone and converted into a digital signal to be sent over the wireless link. The main objective of this step was to take enough voice samples per second to accurately recreate the voice on the receiver side, but not to take so many samples that the amount of information would slow down the sending and receiving process. An example of this process can be seen in Figure 3.1.

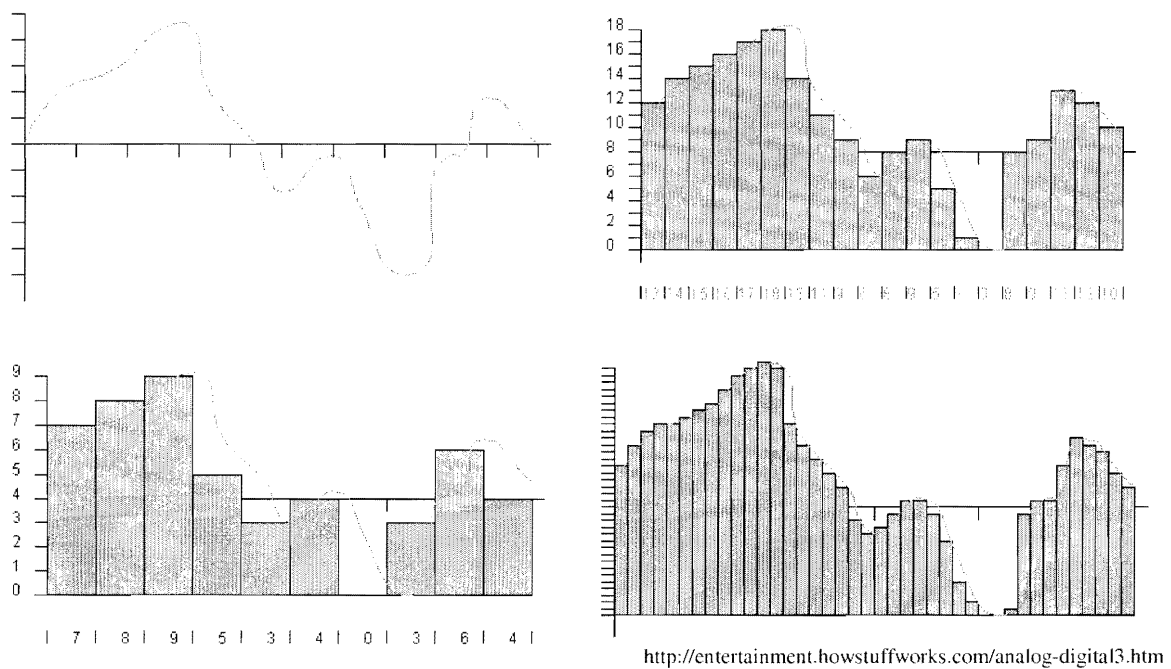


Figure 3.1 – Sampling an analog signal

The first step in the analog to digital conversion process involves connecting the microphone to the TI MSP-TRF6901-DEMO board. The MSP430F449 has 8

individually configurable inputs/outputs to the ADC, and these are referenced as A0 through A7 on the schematic of the board seen in Appendix A. We chose to take input from the microphone on pin A4. The microphone was connected to the board through the use of a three way conductor socket, GC Electronics 30-574, so that any microphone could be used. The setup of the microphone circuit is shown below in Figure 3.2.

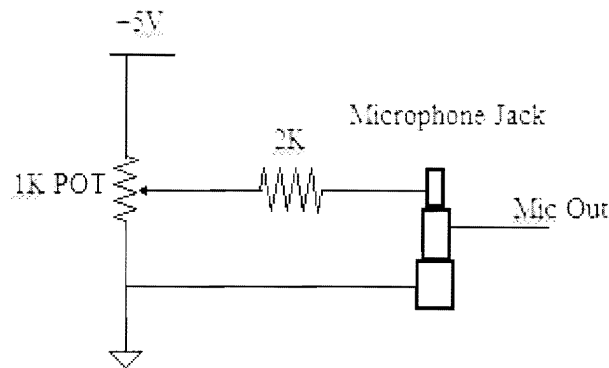


Figure 3.2 – Microphone Setup

Once the microphone is connected properly, the ADC must be configured in the software. It must first be turned on and the reference voltage set, then the appropriate input pin(s) must be enabled (in this case: A4). Finally, conversions must be enabled, and then it becomes a matter of taking samples from the microphone at the appropriate speed. Once a sample is taken, the 12 bit value is stored in a register on the MSP430F449 and can subsequently be processed and sent to the receiver before taking another sample.

The standard bandwidth for voice is considered to be 3 kHz, so according to Nyquist's theorem one must sample a voice signal at greater than twice the highest frequency in order to recreate that signal accurately, so we had to sample our signal at 6 kHz. To configure the ADC to sample at 6 kHz, we added timing loops to the sampling process that wait a specific amount of clock cycles before taking another sample. We

also had to account for the transmission time of each sample in the timing of the sampling loop.

4 DAC – Digital to Analog Conversion

(Jacob DePriest)

Since the MSP430F449 does not have a built in DAC, we were forced to purchase a separate IC to do the job. We chose the 12-bit Analog Devices DAC8512 mainly because it takes a serial input, which simplifies the process of output data from the MSP430 to the DAC, and because of its convenient package. The pin configuration, timing diagram, and control logic truth table for the 8-DIP package can be found in Appendix B. There were two basic parts of the DAC to configure: the DAC8512 output to the speakers and the MSP430 output to the DAC8512.

As seen in the truth table for the DAC8512 in Appendix B.2, the IC is controlled by three main inputs: the SDI pin (Serial Data Input), the LD pin (load input low), and the CS or CLK pin (these can be used interchangeably – we chose the CLK pin). The basic operation of the DAC is as follows: on every rising edge of the CLK input the shift register data in the DAC is advanced one bit, with the LSB being the current bit value of the SDI pin input. On the falling edge of the LD input pin, the main DAC register is updated with the current contents of the input shift register.

Based on the operation of the DAC8512, we configured the MSP430 to have 3 output pins to interface directly with the IC. We chose I/O node P4 for the connection and specifically used P4.2 as the SDI interface, P4.3 as the CLK interface, and P4.4 as the LD interface. These pins were configured and controlled in the software which can be seen in Appendix E.

The algorithm for outputting a 12 bit sample involves taking a sample and stepping through each bit of that sample determining if that bit is a 1 or a 0. The SDI pin is set high or low accordingly and then the CLK pin is shifted from low to high creating a rising edge and inserting the new bit value into the DAC shift register. After this is done for all 12 bits, the LD pin is shifted from high to low creating a falling edge which updates the main DAC register with the new contents of the shift register.

The output of the DAC8512 is fed directly into another three way conductor socket connected to a speaker. This circuit can be seen in figure 4.1 below. The speaker can be powered externally, but the change in the input voice signal is enough to drive an unpowered 8 Ohm speaker. The speed of the DAC is only limited by the speed at which the MSP430 can give it information in the current setup.

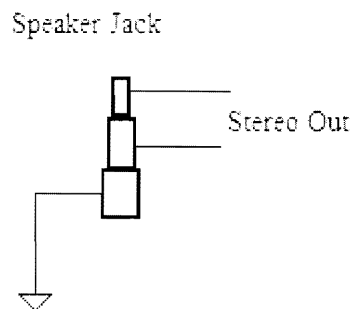


Figure 4.1 – Speaker Setup

5 Data Transmission

(Jason Markley)

Once the input has been sampled by the ADC, it needs to be transmitted to the other board wirelessly. The TRF6901 is capable of using FSK and OOK modulation, and can transmit at a rate of up to 64kbps. The demo boards came pre-configured for FSK operation at 38.4kbps and had several hardware limitations that prevented us from being able to transmit voice across the wireless link.

5.1 Transmitting Bits on the TRF6901

Transmitting data to the TRF6901 is rather simple. There are two registers that hold configuration information for two different modes of operation. In our case, one register was setup for transmission, and one for reception. In order to transmit data, first one needs to set the *mode* line to the logic '0' or '1' that corresponds to which configuration settings to use. Then the *tx_data* line needs to be set to the logic '0' or '1' that is desired to be transmitted. Finally, the TRF6901's $\sim$ *STDBY* line is taken to a logic '1' which will take the transceiver out of standby mode and instruct it to transmit the logic level that is on the *tx_data* line. Thus, pseudo code to transmit a '1' and then a '0' would look like the following:

1. Set mode to tx_mode
2. Set *tx_data* to logic '1'
3. Set $\sim$ *STDBY* to logic '1'
4. Wait for duration of the '1' bit (1 symbol time)
5. Set *tx_data* to logic '0'
6. Wait for duration of the '0' bit (1 symbol time)
7. Set $\sim$ *STDBY* to logic '0'

5.2 Increasing Transmission Speed

The demo code that came with the boards includes a *send_RF* routine which is responsible for sending data to the TRF6901. Below is an excerpt of the code that generates the training sequence for each packet transmission. A training sequence is an alternating stream of '1's and '0's that is sent at the beginning of each packet. The purpose of the training sequence for this application is to properly charge the receiver's

capacitor so that it will be able to accurately discriminate between logic '1' and '0'. See section 5.3.1 of [Trf03] for more detail.

```
1. send_RF_toggle
2. BIS          #CPUOFF+GIE,SR  ; CPU off
3. ;----- Start of the trainings sequence -----
4. XOR.B        #tx,&P2OUT      ; toggle TXDATA(P2.2)
5. DEC          tr_counter      ; decrement counter for the
    training sequence
6. JNZ          send_RF_toggle  ; length of the loop exactly
    26,04μsec
```

It is evident from this code that the transmission speed of the TRF6901 is directly controlled by the speed with which the data and control lines are toggled. Line 2 puts the MSP430 into a low power state. The MSP430 waits until a watchdog timer interrupt is received before it resumes the code loop. In this case, the training sequence loop is described as being 26.04μs in length, which corresponds to a 38.4kbps transmission rate, which is the preconfigured transmission rate described in the documentation. The documentation does state however, that the TRF6901 can achieve reliable transmission at rates of up to 64kbps, so if we could effectively increase the code execution rate on the MSP30, we could get the TRF6901 to transmit faster. The preconfigured clock rate of the MSP430 is approximately 2.4 MHz. If 2.4 MHz corresponds to a transmission rate of 38.4kbps, then we need to increase the clock rate by a factor of $64 / 38.4 = 1.67$. This corresponds to a clock rate of $2.4 \text{ MHz} * 1.67 = 4 \text{ Mhz}$. We were able to issue the following commands to reconfigure the master clock on the MSP430, resulting in the desired clock rate increase.

```
SCFIO |= FN_4;          //set nominal frequency (1.3-12.1 MHz)
SCFQCTL = 63;           //(63+1) x 32768 Hz x 2 = 4 MHz
```

5.3 Bandwidth Considerations

In order to transmit our data effectively, we needed to do so with enough speed that we could decode the data, and send it to the DAC at a fast enough rate so that audible speech would be heard on the speaker. Here is where we ran into some rather significant hardware limitations of the MSP-TRF6901-DEMO board. The most notable limitations were required channel speed, having to transmit multiples of 2 words at a time, the receiver's capacitor for logic '1' and '0' discrimination, and storage space available on the MSP430.

5.4 Channel Speed

As stated above, the board came preconfigured to operate at 38.4kbps. Trying to send 12 bit samples over a 38.4kbps channel would result in a maximum sample transmission rate of $38.4\text{kbps} / 12\text{bits} = 3\text{kHz}$. According to Nyquist's sampling theorem, this means we would only be able to reliably reconstruct frequencies up to $3\text{kHz} / 2 = 1.5\text{kHz}$, certainly not enough for typical voice (which requires approximately 3 kHz bandwidth). We would need to speed up the TRF6901's transmission rate to the maximum specified rate of 64kbps in order to achieve acceptable voice recovery. We were in fact able to speed up the transmission rate to 64kbps, but this was not our only problem. We also had to consider the overhead in the system which among others includes word transmission limitations.

5.5 Word Transmission

In the current configuration for the TRF6901, it is only capable of sending words in multiples of two. Our data samples are 12 bits long, and the word size of the MSP430

is 2 bytes or 16 bits. Thus, when we send one data sample, we are required to send 16 total bits, not just the 12 that contain our data sample. This means that our maximum transmission rate is now $64\text{kbps} / 16\text{bits} = 4\text{ kHz}$ and a corresponding maximum recoverable frequency of 2 kHz.

5.6 Effective Packet Transmission

Word transmission was not the only overhead in the system. The above limitation on speed and word transmission assumes that there is constant data transmission or the continuous streaming of data. The MSP-TRF6901-DEMO board designed for packet transmission. There is a capacitor that is used during reception that needs to remain charged in order to distinguish between a logic '1' and '0'. The number of bits able to be received consecutively is limited to the ability of the capacitor to remain charged to an acceptable level. The TRF6901 documentation states that this time is approximately several seconds. This, however, prevents us from using a circuit type of transmission (streaming data) and forces us to use a packet based transmission.

In packet based transmission, a preamble is send before each packet for synchronization of the words. In the TRF6901's case, this also allows for the charging of the capacitor so that it is able to properly distinguish between logic 1 and 0. The MSP-TRF6901-DEMO documentation (section 5.3) describes a preamble of 100 bits, with start, stop and word sync bits. The documentation states that a shorter preamble may be used for more time sensitive transmission, but does not state how short a preamble will provide reliable bit extraction.

Packet based transmission greatly reduces our ability to send recoverable voice data. Instead of 16bits per sample (because of the even word limitation), now we are being forced to send a minimum of *preamble_length* + 16 + 3 bits for each sample. We found some evidence online that the TRF6901 cannot recover data with a preamble of less than 32bits. This came from another group that did a project with the TRF6901 who posted their report online [BTS03]. Even with a preamble of only 8 bits however, this reduces our maximum transmission rate to $64 \text{ kbps} / 27 \text{ bits} = 2.43 \text{ kHz}$, corresponding to a maximum recoverable frequency of $2.43 \text{ kHz} / 2 = 1.21 \text{ kHz}$.

5.7 Packet Radio

Given the above hardware limitations, we attempted to implement a packet radio solution. This type of solution would seek to record many data samples at once, and then transmit them all in one packet, getting around the overhead of needing a preamble for every data packet. We needed to keep in mind that we would not be able to record more than a few seconds worth of data because of the limitation on the capacitor used in reception. We would need a preamble to recharge the capacitor every few seconds. If we limit the reception of bits to 2 seconds, and assume we are transmitting at a rate of 64kbps, this means we can only send 131,072 bits (128KB) before we need another preamble to recharge the capacitor. With 16 bits per data sample, this equates to $131,072 / 16 = 8,192$ samples. Sampling at twice the Nyquist rate of 3 kHz, this would yield $8,192 / 6,000 = 1.36$ seconds of data per packet. We would not even be able to do this in real time because by assumption, it would take us 2 seconds to transmit these 1.36 seconds of data.

5.8 Storage / Memory

The above solution of packet radio is a good one considering the hardware limitations of the TRF6901 transceiver. However, there is another hardware limitation that prevented us from implementing even the 1.36 second recording option: the memory available for sample storage in the MSP430.

The memory specifications of the MSP430F449 [Msp03] state that there is 60KB of FLASH for code and 2048 bytes of RAM. The amount of storage required for 8,192 samples is $8,192 \text{ samples} * 2 \text{ bytes/sample} = 16\text{KB}$. We discovered that because of space limitations on the MSP430, we were not able to store more than 1000 samples (2,000B). This would mean that sampling at a rate of 6 kHz (twice the Nyquist rate), would only yield 167ms worth of recorded data.

5.9 What We Implemented - Short Recorded Tones

Given all the hardware limitations described above, we decided to demonstrate the transmission of short tones over the wireless link. The 167ms of available recording time (see above section on Storage / Memory) was not deemed enough time to demonstrate actual voice transmission, however the only way to increase this time would be to decrease the sampling rate, thus preventing voice recovery. Demonstrating transmission of tones was the only viable alternative given that our data was comprised of 12bits/sample. We decided to cut the data sampling rate down to 3 kHz, which would allow us to capture 333ms of data in our 1000 samples, while still allowing us to demonstrate the transmission of tones of up to 1.5 kHz in frequency. Because we were

pre-recording our data, there was no longer a need to speed up the MSP430's clock rate; we could simply sample our data at the desired rate of 3 kHz, transmit that data at the original 38.4 kbps over the wireless link, and then pump the data back out to the DAC at a similar rate of 3 kHz, reproducing the original data.

5.10 Future Transmission Enhancements - Reduction of Data Rate

The main limitation we encountered was the transmission rate of the TRF6901 being limited to 64kbps. Because of overhead and sample size, we were unable to transmit voice at this channel rate. However if we were able to reduce our required data rate, then we would surpass this limitation.

One way to achieve a reduced data rate is to compress our sample data so that we are effectively sending less bits/sample than the current 12 (16 including FEC). Digital Cellular phones employ various forms of voice compression in order to reduce the required data rate. GSM's voice CODEC compresses 160 samples down to 33bytes of data, resulting in a data rate of approximately 8kbps. Other voice CODECs used in cellular phones today have data rates of as little as 4-6kbps.

For the MSP-TRF6901-DEMO board, the amount of space required to implement a voice codec capable of such low data rates would be too great. The only way to implement such a CODEC would be to do so in hardware (a GSM coding chip for example) or to increase the memory available to the MSP430.

6 Error Correction

(Augustin Manolache)

A simple method to reduce the probability of error in digital communication is to employ error detection and correction. By adding few extra bits to the message (sometimes referred to as *check bits*) using dedicated algorithms, the probability of error can be lowered dramatically. A simple example from every day life is the necessity to repeat words when communicating with someone in a noisy room. There are two classes of methods for detecting and correcting errors:

- The sender includes enough information for the receiver to detect the errors, and the receiver sends back a NACK (not acknowledged) signal every time there is an error, asking for retransmission of the message
- The sender includes enough information with the message to make error recovery possible at the receiver. This method is commonly referred to as Forward Error Correction (FEC).

In our project we tried to implement such error correction methods, as a real life application of the concepts covered in the course, but also to lower the probability of error, enable communication at faster rates (which proved to be the biggest challenge), and to make good use of the bandwidth.

The output of the ADC will give 12-bit samples. The board is able to transmit only groups of 2 bytes, which is equivalent to 16 bits. Four bits will always be left unused, which is a waste of bandwidth (because they will be transmitted regardless). The logical thing to do is to employ some method of coding that would decrease the

probability of error, which can lead to a speed increase (by increasing the bit rate of the transmitter). The following coding methods were considered:

6.1 Convolutional coding

At first sight this method is somewhat easy to implement using the MSP430 microcontroller, because it does not require lookup tables, just a shift register at the sender (easy to implement with a FIFO list) and a state matrix at the receiver. A problem arises from the fact that the transmission rate is a punctured code rate ($3/4$) with a minimum number of 64 states. For a 64-state encoder, a convolutional code becomes feasible only for codes with lengths above 42 bits. This rapidly increases the complexity of transmit and receive subroutines, since multiple samples need to be concatenated to produce a usable length of code. The main limitation is the memory requirement (because of the 64 possible states) and the processing time. This solution can be implemented only by using additional hardware.

6.2 ARQ (automatic-repeat-request) system

This is the simplest system to implement. Since there are four bits available, it can be implemented in such a way to detect 1 and 2 bit errors (and all the odd combinations). The software requirements are very small, but the decrease in the transmission speed is unacceptable. The speed decrease is due to two reasons:

- A NACK requires the receiver to switch to a transmitter and the sender to switch to a receiver, requiring a preamble (which proved to be a big speed bump in our project)

- The time required for NACK and retransmission will create a pause at the receiver speaker, which causes distortions in the audio signal (since the DAC needs constant rate of data to recreate the audio signal).

6.3 Block coding (Hamming codes in particular)

This kind of coding can provide good gain even at smaller size blocks, with average complexity in the software and small memory requirements. A Hamming code (and for this purpose, any block code that can correct for at least one error) has the following requirement for the minimum number of check bits to be added to the code:

$$n \leq 2^q - 1 \text{ where } k = n - q$$

For 12 message bits we need $q = 5$, with $n = 17$ total bits. The system needs to stay inside the 16 bits that can be transmitted at a time. The solution is to encode only 11 bits using a (15, 11) Hamming code, and transmit the least significant bit unchanged.

After careful consideration of all three possibilities, we have chosen the Hamming block code to be the most suitable for this kind of operation. The following tables and diagrams present in detail the code being used ($n = 15$, $k = 11$, $q = 4$).

The Generator matrix and P sub matrix are:

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \quad P = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

We have the following equations for the check bits:

$$\begin{aligned}
 c_1 &= m_5 \oplus m_6 \oplus m_7 \oplus m_8 \oplus m_9 \oplus m_{10} \oplus m_{11} \\
 c_2 &= m_2 \oplus m_3 \oplus m_4 \oplus m_8 \oplus m_9 \oplus m_{10} \oplus m_{11} \\
 c_3 &= m_1 \oplus m_3 \oplus m_4 \oplus m_6 \oplus m_7 \oplus m_{10} \oplus m_{11} \\
 c_4 &= m_1 \oplus m_2 \oplus m_4 \oplus m_5 \oplus m_7 \oplus m_9 \oplus m_{11}
 \end{aligned}$$

The H^T matrix is: $H^T \equiv \begin{bmatrix} P \\ I_q \end{bmatrix}$

In order to determine the errors we need to generate the syndrome look-up table.

The table can be created using various input vectors of length 15, containing a maximum of one 1.

	S				Error														
	s_1	s_2	s_3	s_4															
0 0 1 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0 1 0 1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0 1 1 0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0 1 1 1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1 0 0 1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1 0 1 0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1 0 1 1	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1 1 0 0	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1 1 0 1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1 1 1 0	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
1 1 1 1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1 0 0 0	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0 1 0 0	1	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0 0 1 0	1	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0 0 0 1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0

We have the following equations for the syndrome bits:

$$\begin{aligned}
 s_1 &= y_5 \oplus y_6 \oplus y_7 \oplus y_8 \oplus y_9 \oplus y_{10} \oplus y_{11} \oplus y_{12} \\
 s_2 &= y_2 \oplus y_3 \oplus y_4 \oplus y_8 \oplus y_9 \oplus y_{10} \oplus y_{11} \oplus y_{13} \\
 s_3 &= y_1 \oplus y_3 \oplus y_4 \oplus y_6 \oplus y_7 \oplus y_{10} \oplus y_{11} \oplus y_{14} \\
 s_4 &= y_1 \oplus y_2 \oplus y_4 \oplus y_5 \oplus y_7 \oplus y_9 \oplus y_{11} \oplus y_{15}
 \end{aligned}$$

The FSK modulation used in TRF6901 results in a probability of bit error

of $P_{be} = \frac{1}{2} e^{-\gamma_b/2}$. For coding purposes we can take write this as α . For 12 consecutive bits

we have a probability of error of: $P_{lowe} = 1 - P(0, n) = 1 - \binom{12}{0} (1 - \alpha)^{12} = 1 - (1 - \alpha)^{12}$

The probability of more than 1 error in the 15 bits of the Hamming code is:

$$P_{Hwe} = 1 - P(0, n) - P(1, n) = 1 - \binom{15}{0} (1 - \alpha)^{15} - \binom{15}{1} \alpha (1 - \alpha)^{14} = 1 - (1 - \alpha)^{15} - 15\alpha (1 - \alpha)^{14}$$

Added to this is the probability of error in the least significant bit (which was left uncoded). The total probability of error for the result of the coding is:

$$P_{we} = 1 - (1 - \alpha)^{15} - 15\alpha (1 - \alpha)^{14} + \alpha.$$

A graphical representation of the probabilities of error for coded and uncoded transmission, relative to the probability of bit error on the channel, is given in the following chart:

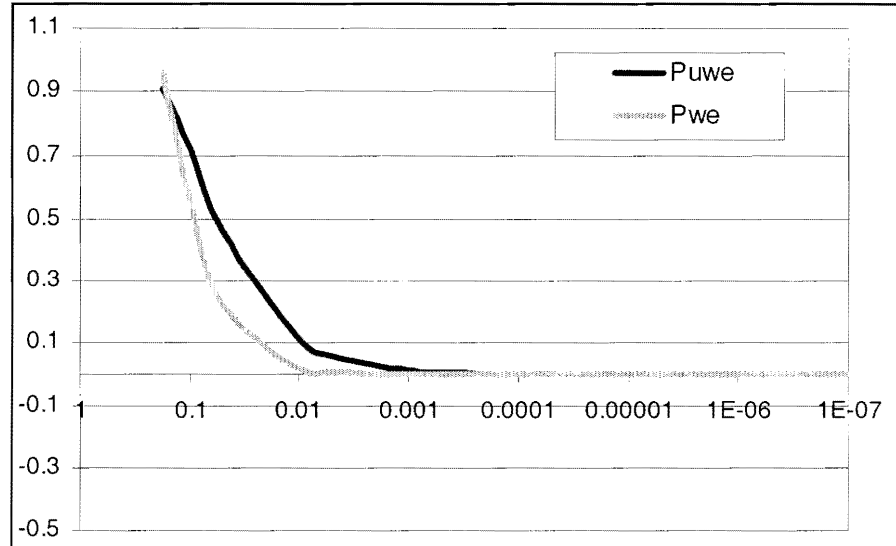


Figure 6.3.1 - Probabilities of word errors vs. probability of bit error

Another important chart is the coding gain versus bit probability of error, and it is given in the next chart. It can be noted from the chart that the best gain is obtained for bit probabilities of error of at least 10^{-3} . For anything over this value, this type of coding becomes inefficient.

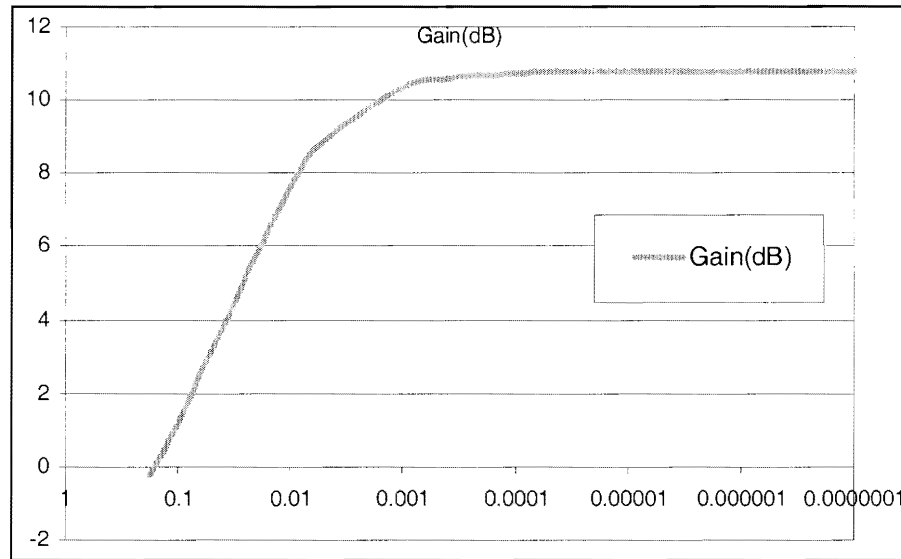


Figure 6.3.2 - Coding gain vs. initial probability of bit error

The implementation strategy is:

- a) At the sender the encoder calculates the four check bits, then adds them to the message
- b) At the receiver the decoder calculates the four syndrome bits, and looks up the result in the syndrome-error table to determine what bit needs to be changed if the syndrome is not 0. The check bits are cleared and the resulting message is returned.

The full code can be seen in Appendix E.

7 Results

(Jason Markley and Jacob DePriest)

Our original proposal involved full-duplex CDMA communication, but this was realized early on to be impossible due to the sample and hold capacitor used for receiving in the TRF6901. Since streaming data was no longer a possibility, full-duplex operation via CDMA was also no longer possible. Therefore, full-duplex communication became a non-trivial task – implemented by simply selecting two separate frequency channels for the links. Because of this change of direction in our original design, we decided to implement a Forward Error Correction (FEC) coding scheme to decrease the error probability.

Because of the hardware limitations of the transceiver discussed in previous sections, we were not able to accurately reconstruct the voice signal sent from the transmitter to the receiver. However, we were able to demonstrate the effectiveness of the ADC and DAC steps on a single TI MSP-TRF6901-DEMO board by removing the wireless transmission and reception from the process. By setting up both the ADC and DAC on the same board, we were successfully able to input a voice signal over the microphone and recreate the signal and output it out of the speaker.

Even though we were not able to reproduce a voice signal over the wireless link, we successfully transmitted various tones produced by varying the input to the microphone. We successfully implemented each individual part of the digital, wireless voice communication link, but we were not able to execute the transmission stage at a fast enough rate to demonstrate a complete working voice communication system.

8 Conclusion

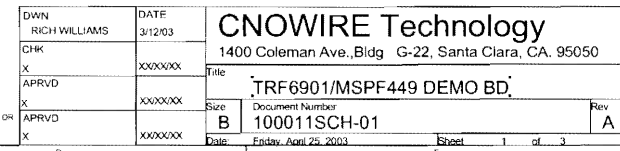
(Jason Markley and Jacob DePriest)

This particular design project has allowed us to gain insight into the complexities that go into the electronics that drive our lives. There are many things that affect the digital voice communication process such as timing, hardware, and software. The efficiency of the system must also be a consideration. We have been able to take a deeper look into the realm of digital, wireless communication and have gained a deeper understanding of communications engineering through this process.

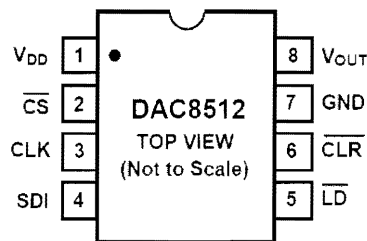
Overall, we succeeded in implementing a digital, wireless communication system, and if we had not encountered the hardware limitations outlined in this report our original goal of wireless voice transmission would have been realized.

9 References

- [AD96] Analog Devices: *DAC8512: 5V, Serial Input Complete 12-Bit DAC Data Sheet (Rev. A. 11/96)*
http://www.analog.com/UploadedFiles/Data_Sheets/42749849DAC8512a.pdf November 1996.
- [BTS03] Boureslie, Fawaz; Lai, Gerald; McCredie, Matt; Sullivan, Brian: *Technical Manual, Bus Tracking System*.
<http://classes.engr.oregonstate.edu/eecs/spring2003/cce44x/groups/g2/Tech.pdf> 2003.
- [Msp03] Texas Instruments: *Mixed Signal Controller MSP430F449 Data Sheet (Rev. C)*
<http://www-s.ti.com/sc/ds/msp430f449.pdf> March 2003.
- [Msp02] Texas Instruments: *MSP430x4xx Family User's Guide (Rev. D)*
<http://focus.ti.com/lit/ug/slau056d/slau056d.pdf> April 2002
- [Trf03] Texas Instruments: *TRF6901 with MSP430 Demonstration and Development Evaluation Kit User's Guide (Rev. A)*
<http://focus.ti.com/lit/ug/slwu008a/slwu008a.pdf> August 2003.



B.1 – Pin Configuration



B.2 – Control Logic Truth Table

CS ²	CLK ²	CLR	LD	Serial Shift Register Function	DAC Register Function
H	X	H	H	No Effect	Latched
L	L	H	H	No Effect	Latched
L	H	H	H	No Effect	Latched
L	↑+	H	H	Shift-Register-Data Advanced One Bit	Latched
↑+	L	H	H	Shift-Register-Data Advanced One Bit	Latched
H	X	H	↓-	No Effect	Updated with Current Shift Register Contents
H	X	H	L	No Effect	Transparent
H	X	L	X	No Effect	Loaded with All Zeros
H	X	↑+	H	No Effect	Latched All Zeros

NOTES

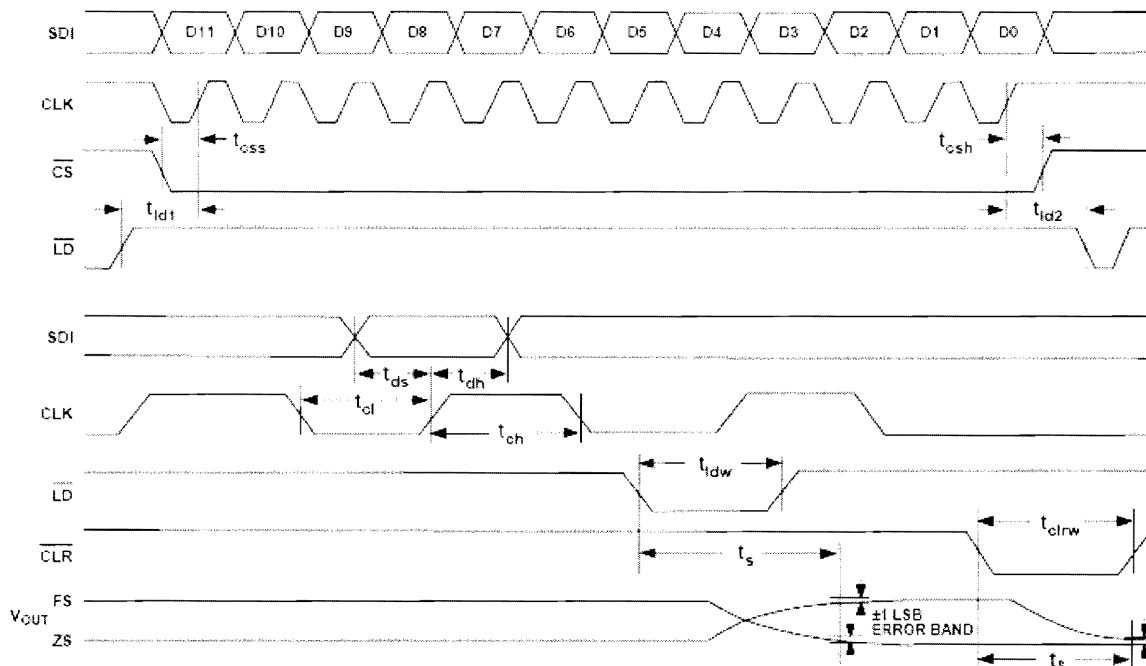
¹+ positive logic transition, - negative logic transition; X = Don't Care.

²CS and CLK are interchangeable.

⁵Returning $\overline{CS}$ HIGH avoids an additional "false clock" of serial data input.

⁴Do not clock in serial data while $\overline{\text{LD}}$ is LOW.

B.3 – Timing Diagram

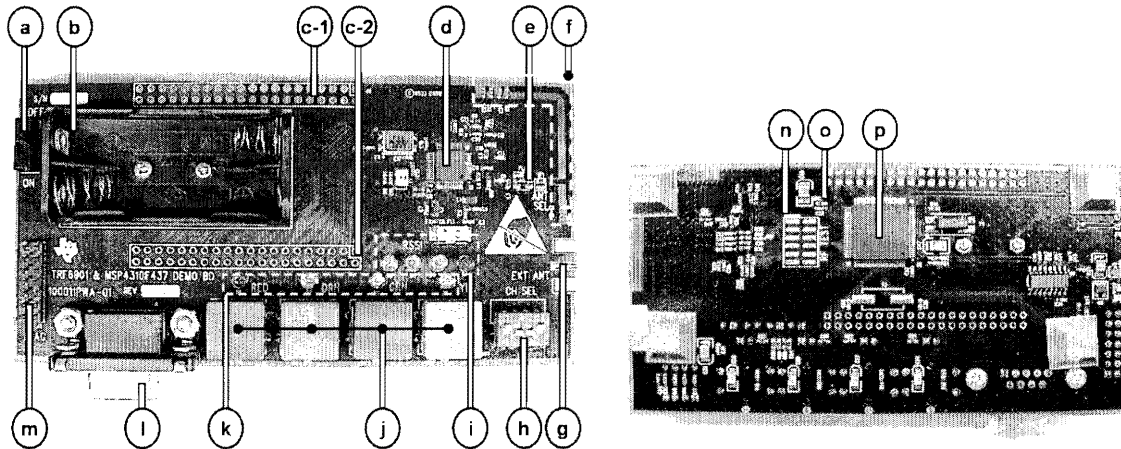


Appendix C – Budget

(Augustin Manolache)

Item Number	Item Name	Item Group	Qty	Cost	Extended Cost
30-543-1	Phone Jack	Misc	1	\$2.66	\$2.66
30-574	Phone Jack	Misc	2	\$2.65	\$5.30
30-546	Phone Jack	Misc	1	\$2.66	\$2.66
MSP-TRF-6901-DEMO	TRF6901 Demo board	Misc	2	\$101.47	\$202.94
MSP-FET430P440EVM	JTAG tool	Misc	1	\$59.50	\$59.50
Microphone	Microphone	Misc	1	\$3.00	\$3.00
Sony SRS-7	Speakers	Misc	1	\$3.00	\$3.00
1Kohm	Resistor	Misc	2	\$0.20	\$0.40
DAC8512	Digital Analog Convert	Misc	2	\$16.37	\$32.73
TOTAL			13		\$312.19

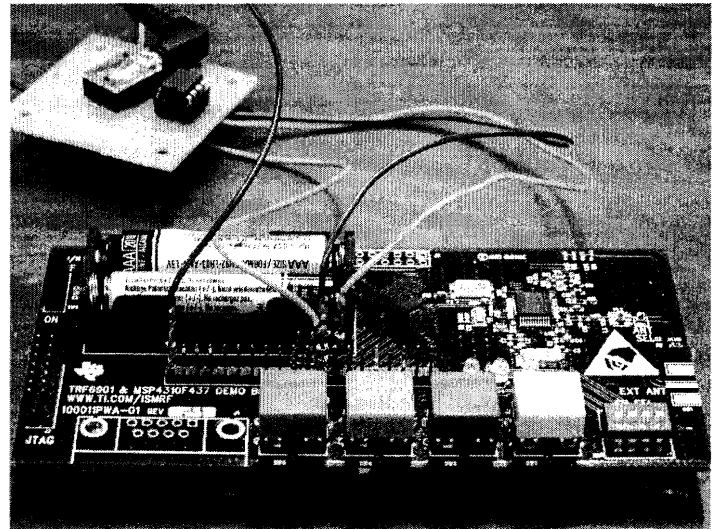
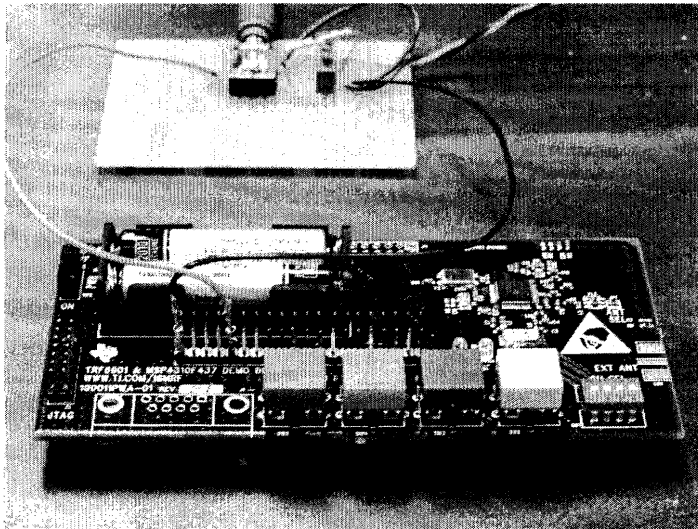
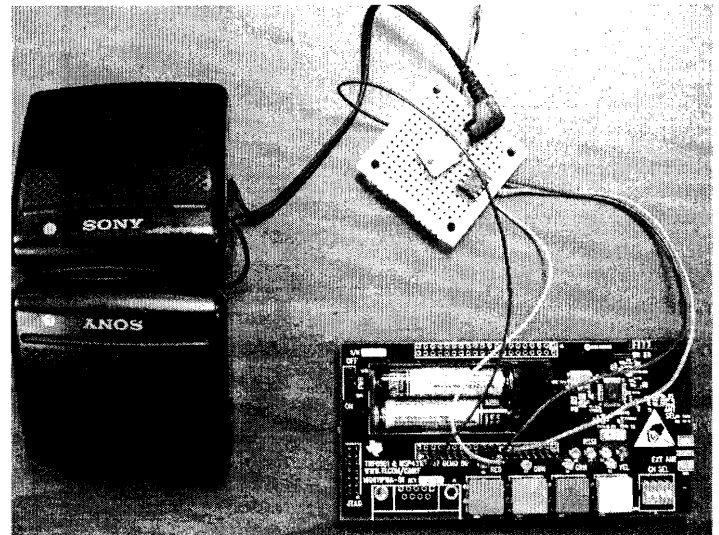
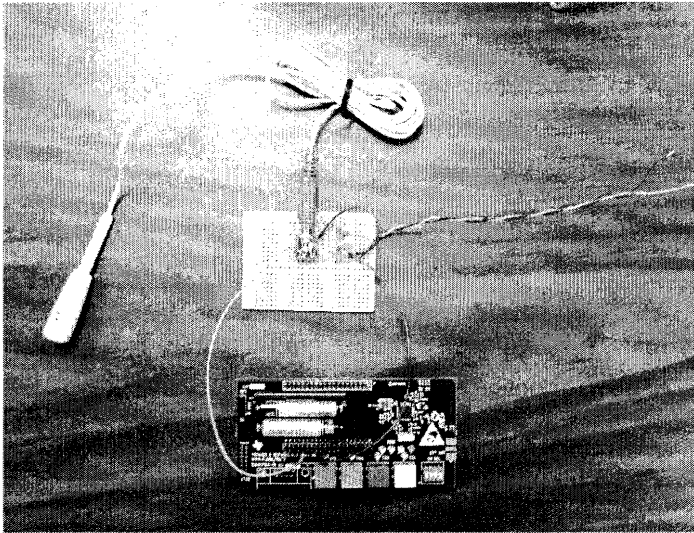
D.1 – Board Layout



- a) On/Off switch
- b) Battery holder: stores two AAA batteries in series (batteries are not included)
- c) MSP430 headers: used to access the I/O ports of the MSP430F449
- d) TRF6901 component
- e) Antenna selection jumper used to select either the SMA connector or PCB antenna.
- f) On board PCB antenna (default configuration)
- g) SMA footprint connector used to add an external antenna (SMA connector not included)
- h) 4-bit DIP switch used in the demonstration to change the frequency configuration. The MSP430 I/O ports P2.3, P2.6, P2.7, and P3.0 connect to the 4-bit DIP switch.
- i) Receive signal strength indicator (RSSI) LEDs – used as the RSSI for the demonstration. The LEDs are tied to the MSP430 P6.0 to P6.3 I/O ports.
- j) Four buttons used for the demonstration application; connected to the MSP430 P1.0 to P1.3 I/O ports.
- k) Four LEDs used for the demonstration application; connected to the MSP430 P1.3 to P1.7 I/O ports.
- l) RS-232C connector footprint (RS-232C circuitry is not included in the kit)
- m) JTAG connector
- n) TRF6901 I/O header used to control or monitor the TRF6901 component externally.
- o) MSP430 disable jumper used to disable the MSP430F449 component
- p) MSP430F449 component

[Trf03]

D.2 – Project Hardware



Appendix E – Digital Communication Code

E:\Design\ECE442\2004-05-02_working_1000_samples_tones\2004-04-21_b\trf6901_demo.c 1

```
1  /***** TRF6901_demo.c *****/
2  *
3  *   DESCRIPTION: TRF6901 CODE
4  *
5  *   Jason Markley, Jacob DePriest, Augustin Manolache
6  *
7  *   Date: 5/2/04
8  *
9  *****/
10
11 /** include files */
12 #include "msp430x44x.h"
13 #include "in430.h"
14 #include "stdio.h"
15 #include "f_map.h"
16
17
18 /** local definitions */
19 //TRF6901 control signals
20 #define TRF_STROBE 0x2
21 #define TRF_DATA 1
22 #define TRF_MODE 0x80
23 #define TRF_CLK 0x40
24 #define TRF_STANDBY 0x20
25 #define TRF_LH 0x10
26 #define tx 0x4          //; P2.2 TXDATA transmit data to the TRF6901
27 #define rx 0x8          //; P1.3 RXDATA receive data from the TRF6901
28 #define FALSE 0
29
30 #define LED_YELLOW 0x80
31 #define LED_GREEN 0x40
32 #define LED_ORANGE 0x20
33 #define LED_RED 0x10
34 #define LEDALL 0xf0
35 #define FSEL4 1
36 #define FSEL3 0x80
37 #define FSEL2 0x40
38 #define FSEL1 0x8
39 #define NONE 0
40 #define RED 1
41 #define ORANGE 2
42 #define GREEN 4
43 #define YELLOW 8
44 #define MAXWORD 3          //size in word
45 #define MAXBYTE MAXWORD*2  //size in byte
46 #define FALSE 0
47 #define TRUE 1
48 #define ACK_CODE 0x0606
49 #define COM_SUCCESSFUL 6
50 #define LOCAL_BUTTON_PUSHED 0x8          //equivalent to uart_R_buf_full
51
52 #define rf_X_buf_full 0x4          //if xmit buffer has new data
53 #define RF_timedout_flag 0x80
54 #define rf_R_buf_full 0x10
55 #define F_change_flag 0x100
56 #define MAX_RETRAN 3
57
58 #define SEC5 50          /* default settings */
59 #define SEC10 100
60 #define RESPONSE_TIMEOUT SEC10
61 #define SAMPLEPERIOD 0x7800;
62 #define XTAL_OFFSET 0
63 #define DEBOUNCE 7690          //25ms
64 #define US 0x40
65 #define EU 0
66
```

```

67 /* our definitions */
68 #define DAC_Out_High (0x04) //Output HIGH to DAC - Serial Output Pin 3 of an 8 pin output ✓
69 #define DAC_Out_Low (0xFB) //Output LOW to DAC - Serial Output Pin 3 of an 8 pin output ✓
70 #define DAC_CS_High (0x08) //Chip select HIGH to DAC Pin 4 of an 8 pin output ✓
71 #define DAC_CS_Low (0xF7) //Chip select LOW to DAC Pin 4 of an 8 pin output ✓
72 #define DAC_LD_High (0x10) //Load strobe HIGH to DAC Pin 5 of an 8 pin output ✓
73 #define DAC_LD_Low (0xEF) //Load strobe LOW to DAC Pin 5 of an 8 pin output ✓
74 #define DATA_SIZE 1000
75 #define SAMPLE_DELAY 0x53
76 #define SAMPLE_DELAY2 0x20
77
78 #define SENDER
79 //#define RECEIVER
80
81 //out functions
82 void toDAC(unsigned int data);
83 int bit_select(unsigned int num, unsigned int index);
84 unsigned int Ham_decode(unsigned int num);
85 unsigned int Ham_encode(unsigned int num);
86 unsigned int bit_toggle(unsigned int num, unsigned int index);
87 unsigned int bit_set(unsigned int num, unsigned int index, unsigned int vbit);
88 unsigned int bit_get(unsigned int num, unsigned int index);
89
90 /** external functions */
91 extern void Set_Clk(void);
92 extern void Set_DCO(int);
93 extern void program_TRF69(unsigned int, unsigned int); //trf6901.s43
94 extern int receive_RF(unsigned char, unsigned int *);
95 extern void send_RF(unsigned int, unsigned int *);
96
97 /** public data */
98
99 int f_sel, fsel_update;
100 int ComFlag;
101
102 /** public functions */
103 void InitIo(void);
104 int GetFsel(void);
105
106 void configure_trf6901(void);
107 void program_TX_Mode_0(void); //
108 void program_RX_Mode_1(void); //
109 void program_WORD_C(void); //
110 void program_WORD_D(unsigned long); //
111 void program_Tx_C(void);
112 void program_Rx_C(void);
113
114 struct trf69_control{
115     unsigned int a_word_l, a_word_h, b_word_l, b_word_h;
116     unsigned int c_word_l, c_word_h, d_word_l, d_word_h;
117 }trf6901;
118
119 int ms100_cnt; //5 sec delay
120 unsigned int Btime;
121 unsigned int Btime_5;
122 unsigned char REGION;
123 unsigned char retry_count;
124 unsigned char failed;
125
126 //unsigned int data; //single samples

```

```

127 unsigned int data2[DATA_SIZE];
128
129 /* ***** */
130 * main program function
131 * ***** */
132 void main(void)
133 {
134     int i,j;
135     unsigned char temp_s = 0;
136     unsigned char temp;
137     int error1;
138
139     #ifdef SENDER
140     #ifndef RECEIVER
141     unsigned int data;
142     #endif
143     #endif
144
145     WDTCTL = WDTPW+WDTHOLD; // Stop watchdog timer
146
147     #ifndef RECEIVER
148     /* supposedly primes node 4 to be an output node */
149     P4OUT |= DAC_Out_High+DAC_CS_High+DAC_LD_High;
150     P4DIR |= DAC_Out_High+DAC_CS_High+DAC_LD_High;
151     #endif
152
153     #ifdef SENDER
154     /* testing of the output speed */
155     P4OUT |= DAC_LD_High;
156     P4DIR |= DAC_LD_High;
157
158     /* our code start */
159     P6SEL |= 0x10; // Enable A/D channel A4
160     ADC12CTL0 = ADC12ON+SHT0_2+REFON; // Turn on and set up ADC12
161     ADC12CTL1 = SHP; // Use sampling timer
162     ADC12MCTL0 = INCH_4 + SREF_1; // Vr+=Vref+
163
164     for ( j=0; j<0x3600; j++) // Delay for reference start-up
165     {
166     }
167
168     ADC12CTL0 |= ENC; // Enable conversions
169     /* our code end */
170     #endif
171
172     InitIo();
173     Btime=64;
174     REGION=P6IN&US; //check the region of operation
175     Btime1_5=60; //new 1.5ms total timing
176     f_sel = GetFsel();
177     for(i=0;i<10;i++) //retry multiple times to make sure the RF chip
        programmed during power up
178         configure_trf6901();
179
180     /* main program loop */
181     while(1)
182     {
183         #ifdef SENDER
184         #ifndef RECEIVER
185         while(1)
186         {
187             for(i = 0;i<DATA_SIZE;i++)
188             {
189                 ADC12CTL0 |= ADC12SC; // Start conversion
190                 while ((ADC12IFG & BIT0)==0);
191                 data2[i] = ADC12MEM0;

```

```

192         for(j = 0;j < SAMPLE_DELAY;j++);
193
194         temp_s = ~temp_s;           //attempt to indicate how fast we're sending
195         if(temp_s)
196         {
197             j |= DAC_LD_High;
198         }
199         else
200         {
201             j &= DAC_LD_Low;
202         }
203
204     }
205
206     for(i = 0;i<DATA_SIZE;i++)
207     {
208         toDAC(data2[i]); /*sends data to DAC */
209         for(j = 0;j<SAMPLE_DELAY2;j++);
210     }
211 }
212 #endif
213 #endif
214
215 #ifdef RECEIVER
216 ComFlag&=~rf_R_buf_full;
217 error1 = receive_RF(sizeof(data2),data2);
218
219 //data = data ^ 0xFFFF;
220
221 for(i =0;i<DATA_SIZE;i++)
222 {
223     toDAC(data2[i]); /*sends data to DAC */
224     for(j = 0;j<SAMPLE_DELAY2;j++);
225 }
226 #endif
227
228 #ifdef SENDER
229
230 data2[DATA_SIZE-1] = 0;
231 temp_s = 0;
232
233 for(i = 0;i<DATA_SIZE;i++)
234 {
235     ADC12CTL0 |= ADC12SC;           // Start conversion
236     while ((ADC12IFG & BIT0)==0);
237     data2[i] = ADC12MEM0;
238     //data2[i] = data2[(i-1)%DATA_SIZE] ^ 0xFFFF;
239     for(j = 0;j<SAMPLE_DELAY;j++);
240     //data = ADC12MEM0;             //;ust for one sample
241
242     //attempt to indicate how fast we're sending
243     if(temp_s ^ 0xFFFF)
244     {
245         P4OUT|=DAC_LD_High;
246     }
247     else
248     {
249         P4OUT&=DAC_LD_Low;
250     }
251 }
252
253 ComFlag|=rf_X_buf_full;           //indicate rf xmit buff full
254 send_RF(sizeof(data2),data2);
255
256 #endif
257

```

```

258     } //end while(1)
259
260 } //end main
261
262 /***** ***** *****/
263 Function: toDAC
264 Description: sends our data to the DAC
265 *****/
266 void toDAC(unsigned int data)
267 {
268     static unsigned int temp = 0;
269
270     P4OUT|=DAC_LD_High; /*primes (set to HIGH, the LD input to the DAC for the
271                          falling
272                          edge after contents are loaded */
273     /* outputs the 12bit input from the ADC to the DAC as it receives it */
274
275     P4OUT&=DAC_CS_Low; /*sets the Chip Select to zero before bit input */
276     temp = bit_select(data,11); /* checks to see if each bit is high */
277     if(temp)
278         P4OUT|=DAC_Out_High;
279     else
280         P4OUT&=DAC_Out_Low;
281
282     P4OUT|=DAC_CS_High; /*flips the Chip Select to one for SR data advance one bit */
283     /
284
285     P4OUT&=DAC_CS_Low; /*sets the Chip Select to zero before bit input */
286     temp = bit_select(data,10); /* checks to see if each bit is high */
287     if(temp)
288         P4OUT|=DAC_Out_High;
289     else
290         P4OUT&=DAC_Out_Low;
291
292     P4OUT|=DAC_CS_High; /*flips the Chip Select to one for SR data advance one bit */
293     /
294
295     P4OUT&=DAC_CS_Low; /*sets the Chip Select to zero before bit input */
296     temp = bit_select(data,9); /* checks to see if each bit is high */
297     if(temp)
298         P4OUT|=DAC_Out_High;
299     else
300         P4OUT&=DAC_Out_Low;
301
302     P4OUT|=DAC_CS_High; /*flips the Chip Select to one for SR data advance one bit */
303     /
304
305     P4OUT&=DAC_CS_Low; /*sets the Chip Select to zero before bit input */
306     temp = bit_select(data,8); /* checks to see if each bit is high */
307     if(temp)
308         P4OUT|=DAC_Out_High;
309     else
310         P4OUT&=DAC_Out_Low;
311
312     P4OUT|=DAC_CS_High; /*flips the Chip Select to one for SR data advance one bit */
313     /
314
315     P4OUT&=DAC_CS_Low; /*sets the Chip Select to zero before bit input */
316     temp = bit_select(data,7); /* checks to see if each bit is high */
317     if(temp)
318         P4OUT|=DAC_Out_High;
319     else
320         P4OUT&=DAC_Out_Low;
321
322     P4OUT|=DAC_CS_High; /*flips the Chip Select to one for SR data advance one bit */

```

```

319
320 P4OUT&=DAC_CS_Low; /*sets the Chip Select to zero before bit input */
321 temp = bit_select(data,6); /* checks to see if each bit is high */
322 if(temp)
323     P4OUT|=DAC_Out_High;
324 else
325     P4OUT&=DAC_Out_Low;
326
327 P4OUT|=DAC_CS_High; /*flips the Chip Select to one for SR data advance one bit */
328 /
329 P4OUT&=DAC_CS_Low; /*sets the Chip Select to zero before bit input */
330 temp = bit_select(data,5); /* checks to see if each bit is high */
331 if(temp)
332     P4OUT|=DAC_Out_High;
333 else
334     P4OUT&=DAC_Out_Low;
335
336 P4OUT|=DAC_CS_High; /*flips the Chip Select to one for SR data advance one bit */
337 /
338 P4OUT&=DAC_CS_Low; /*sets the Chip Select to zero before bit input */
339 temp = bit_select(data,4); /* checks to see if each bit is high */
340 if(temp)
341     P4OUT|=DAC_Out_High;
342 else
343     P4OUT&=DAC_Out_Low;
344
345 P4OUT|=DAC_CS_High; /*flips the Chip Select to one for SR data advance one bit */
346 /
347 P4OUT&=DAC_CS_Low; /*sets the Chip Select to zero before bit input */
348 temp = bit_select(data,3); /* checks to see if each bit is high */
349 if(temp)
350     P4OUT|=DAC_Out_High;
351 else
352     P4OUT&=DAC_Out_Low;
353
354 P4OUT|=DAC_CS_High; /*flips the Chip Select to one for SR data advance one bit */
355 /
356 P4OUT&=DAC_CS_Low; /*sets the Chip Select to zero before bit input */
357 temp = bit_select(data,2); /* checks to see if each bit is high */
358 if(temp)
359     P4OUT|=DAC_Out_High;
360 else
361     P4OUT&=DAC_Out_Low;
362
363 P4OUT|=DAC_CS_High; /*flips the Chip Select to one for SR data advance one bit */
364 /
365 P4OUT&=DAC_CS_Low; /*sets the Chip Select to zero before bit input */
366 temp = bit_select(data,1); /* checks to see if each bit is high */
367 if(temp)
368     P4OUT|=DAC_Out_High;
369 else
370     P4OUT&=DAC_Out_Low;
371
372 P4OUT|=DAC_CS_High; /*flips the Chip Select to one for SR data advance one bit */
373 /
374 P4OUT&=DAC_CS_Low; /*sets the Chip Select to zero before bit input */
375 temp = bit_select(data,0); /* checks to see if each bit is high */
376 if(temp)
377     P4OUT|=DAC_Out_High;

```

```

378     else
379         P4OUT&=DAC_Out_Low;
380
381     P4OUT|=DAC_CS_High; /*flips the Chip Select to one for SR data advance one bit */
382     /
383
384     P4OUT&=DAC_LD_Low; /* moves the contents of the SR to the output register on the
385     DAC */
386 }
387
388 Function: bit_select
389 Description: takes the ADC 12bit number and checks to see if bit 'index'
390             is high or low and returns 1 or 0 accordingly
391
392 int bit_select(unsigned int num, unsigned int index)
393 {
394     switch(index)
395     {
396     case 0:    if((0x01 & num) > 0)
397                 return 1;
398                 else
399                 return 0;
400
401     case 1:    if((0x02 & num) > 0)
402                 return 1;
403                 else
404                 return 0;
405
406     case 2:    if((0x04 & num) > 0)
407                 return 1;
408                 else
409                 return 0;
410
411     case 3:    if((0x08 & num) > 0)
412                 return 1;
413                 else
414                 return 0;
415
416     case 4:    if((0x10 & num) > 0)
417                 return 1;
418                 else
419                 return 0;
420
421     case 5:    if((0x20 & num) > 0)
422                 return 1;
423                 else
424                 return 0;
425
426     case 6:    if((0x40 & num) > 0)
427                 return 1;
428                 else
429                 return 0;
430
431     case 7:    if((0x80 & num) > 0)
432                 return 1;
433                 else
434                 return 0;
435
436     case 8:    if((0x100 & num) > 0)
437                 return 1;
438                 else
439                 return 0;
440
441     case 9:    if((0x200 & num) > 0)

```

```

442         return 1;
443     else
444         return 0;
445
446     case 10:    if((0x400 & num) > 0)
447                 return 1;
448     else
449                 return 0;
450
451     case 11:    if((0x800 & num) > 0)
452                 return 1;
453     else
454                 return 0;
455
456     default:    return 1;    /* just output high if something screws up */
457 }
458 }
459
460 /*****
461 Function: Ham_decode
462 Description: The bits from 1-15 are going to be decoded. Bit 0 is left alone.
463 The result will have only bits 1-11 set, the upper bits should be 0
464 *****/
465 unsigned int Ham_decode(unsigned int num)
466 {
467     unsigned int s, c;
468     // Before error correction we need to calculate the syndrome
469     s = 0;
470     // First syndrome bit s1 = y5 + y6 + y7 + y8 + y9 + y10 + y11 + y12
471     c = bit_get(num,5)+bit_get(num,6)+bit_get(num,7)+bit_get(num,8)+bit_get(num,9)+
472         bit_get(num,10)+bit_get(num,11)+bit_get(num,12);
473     c &= 1;
474     s = bit_set(s, 3, c);
475     // Second syndrome bit s2 = y2 + y3 + y4 + y5 + y8 + y10 + y11 + y13
476     c = bit_get(num,2)+bit_get(num,3)+bit_get(num,4)+bit_get(num,8)+bit_get(num,9)+
477         bit_get(num,10)+bit_get(num,11)+bit_get(num,13);
478     c &= 1;
479     s = bit_set(s, 2, c);
480     // Third syndrome bit s3 = y1 + y3 + y4 + y5 + y7 + y10 + y11 + y14
481     c = bit_get(num,1)+bit_get(num,3)+bit_get(num,4)+bit_get(num,6)+bit_get(num,7)+
482         bit_get(num,10)+bit_get(num,11)+bit_get(num,14);
483     c &= 1;
484     s = bit_set(s, 1, c);
485     // Fourth syndrome bit s4 = y1 + y2 + y4 + y5 + y7 + y9 + y11 + y15
486     c = bit_get(num,1)+bit_get(num,2)+bit_get(num,4)+bit_get(num,5)+bit_get(num,7)+
487         bit_get(num,9)+bit_get(num,11)+bit_get(num,15);
488     c &= 1;
489     s = bit_set(s, 0, c);
490
491     //Now we zero out the 4 check bits, they are not needed anymore
492     num &= 0x0FFF;
493
494     // Based on the syndrome we can correct errors. We have 16 states with three
495     // possibilities
496     // 1) no error - we should do nothing in this case
497     // 2) error in the check bits - we do not care since we will clear them out anyway
498     // 3) error in the message bits - we need to switch the affected bit
499     switch(s)
500     {
501     case 3:    {
502                 num = bit_toggle(num, 1);
503                 return num;
504             }
505     case 5:    {
506                 num = bit_toggle(num, 2);
507                 return num;
508             }
509     }
510 }

```

```

503     }
504     case 6: {
505         num = bit_toggle(num, 3);
506         return num;
507     }
508     case 7: {
509         num = bit_toggle(num, 4);
510         return num;
511     }
512     case 9: {
513         num = bit_toggle(num, 5);
514         return num;
515     }
516     case 10: {
517         num = bit_toggle(num, 6);
518         return num;
519     }
520     case 11: {
521         num = bit_toggle(num, 7);
522         return num;
523     }
524     case 12: {
525         num = bit_toggle(num, 8);
526         return num;
527     }
528     case 13: {
529         num = bit_toggle(num, 9);
530         return num;
531     }
532     case 14: {
533         num = bit_toggle(num, 10);
534         return num;
535     }
536     case 15: {
537         num = bit_toggle(num, 11);
538         return num;
539     }
540     default: return num;
541 } // end switch
542 }
543
544 /* ***** */
545 Function: Ham_encode
546 Description: The bits from 1-11 are going to be encoded, with the check
547 bits set in 12, 13, 14 and 15. Bit 0 is left alone.
548 /* ***** */
549 unsigned int Ham_encode(unsigned int num)
550 {
551     unsigned int c;
552     // First check bit c1 = m5 + m6 + m7 + m8 + m9 + m10 + m11
553     c = bit_get(num,5)+bit_get(num,6)+bit_get(num,7)+bit_get(num,8)+bit_get(num,9)+
554         bit_get(num,10)+bit_get(num,11);
555     c &= 1;
556     num = bit_set(num, 12, c);
557     // Second check bit c2 = m2 + m3 + m4 + m6 + m8 + m10 + m11
558     c = bit_get(num,2)+bit_get(num,3)+bit_get(num,4)+bit_get(num,6)+bit_get(num,8)+
559         bit_get(num,10)+bit_get(num,11);
560     c &= 1;
561     num = bit_set(num, 13, c);
562     // Third check bit c3 = m1 + m3 + m4 + m6 + m7 + m10 + m11
563     c = bit_get(num,1)+bit_get(num,3)+bit_get(num,4)+bit_get(num,6)+bit_get(num,7)+
564         bit_get(num,10)+bit_get(num,11);
565     c &= 1;
566     num = bit_set(num, 14, c);
567     // Fourth check bit c4 = m1 + m2 + m4 + m5 + m7 + m9 + m11
568     c = bit_get(num,1)+bit_get(num,2)+bit_get(num,4)+bit_get(num,5)+bit_get(num,7)+

```

```

        bit_get(num,9)+bit_get(num,11);
566     c &= 1;
567     num = bit_set(num, 15, c);
568     return num;
569 }
570
571 /*****
572 Function: bit_toggle
573 Description: In the word 'num' toggles the bit specified by 'index'
574 *****/
575 unsigned int bit_toggle(unsigned int num, unsigned int index)
576 {
577     unsigned int i, op;
578     i = 0;
579     op = 1;
580     while (i<index)
581     {
582         op *= 2;
583         i++;
584     }
585     num ^= op;
586     return num;
587 }
588
589 /*****
590 Function: bit_set
591 Description: In the word 'num' sets the bit specified by 'index'
592 'vbit' represents the value to be set, either 0 or anyother number
593 *****/
594 unsigned int bit_set(unsigned int num, unsigned int index, unsigned int vbit)
595 {
596     unsigned int i, op;
597     i = 0;
598     op = 1;
599     while (i<index)
600     {
601         op *= 2;
602         i++;
603     }
604     if (vbit==0)
605         num &= (0xFFFF - op);
606     else
607         num |= op;
608     return num;
609 }
610
611 /*****
612 Function: bit_get
613 Description: Gets the bit specified by 'index' from the word 'num' and returns
614 it as a word value.
615 *****/
616 unsigned int bit_get(unsigned int num, unsigned int index)
617 {
618     unsigned int i, op;
619     i = 0;
620     op = 1;
621     while (i<index)
622     {
623         op *= 2;
624         i++;
625     }
626     num &= op;
627     if (num == 0)
628         return 0;
629     else
630         return 1;

```

```

631 }
632
633 void InitIo(void)
634 {
635     int i;
636
637     P6OUT|=0xf;           //All RSSI led off
638     P6DIR |= 0xf;        //RSSI LED driver configured as output
639     P6SEL|=0x80;         //RSSI is AD function
640
641     P4DIR|= TRF_STROBE + TRF_DATA;           //P4.0, P4.1 are output
642     P3DIR|= TRF_CLK + TRF_STANDBY + TRF_MODE + TRF_LH; //P3 OUTPUT
643
644     P2DIR|= tx;
645
646     P1DIR|= LEDALL;
647     P1OUT&= ~LEDALL;
648     P1IES |= 0xf;        //all int on high to low trans
649     P2SEL=0x33;          //select UART, select TimerA capture
650     Set_DCO(30000);       //set dco for 32Khz clock
651     //initialize UART port. Default Baud =93.4K
652     UTCTL0=0x20;
653     URCTL0=0x8;
654     UBR00=0x40;
655     UBR10=0;
656     UMCTL0=0x0;
657     UMCTL0=0;
658     UCTL0=0x11;
659     UCTL0&=0xfe;
660     ME1|=URXE0;          //enable receiver
661     //blink All LED to indicate power up condition
662     for(i=0;i<30000;i++){
663         P1OUT |= LEDALL;
664         P6OUT&=0xf0;
665     }
666     P1OUT&=~(LEDALL);
667     for(i=0;i<30000;i++); //arbitrary wait until TRF6901 warms up
668
669     P6OUT|=0xf;
670     P5DIR|=1;            //use this for test flag
671     P5OUT&=0xfe;         //start off low
672 }
673
674 void configure_trf6901(void)
675 {
676     unsigned long long_reg;
677     P4OUT&=0xfe;          //start with 0 on data
678     program_TX_Mode_0();  //
679     program_RX_Mode_1();  //
680     program_WORD_C();     //
681     long_reg = (unsigned long)XTAL_OFFSET;
682     long_reg<=<=16;
683     long_reg|= TRF69_Dword;
684     program_WORD_D(long_reg);
685 }
686
687 /* ***** Programming the TRF6900 ***** */
688
689 // purpose: programs the Register A for FSK transmission(changed from original TI
690 // config)
691 // (Mode 0, A-word)
692 // *****
693 */

```

```

694 void program_TX_Mode_0(void)
695 {
696     unsigned long trf69;
697     int pointer;
698     pointer=(int)f_sel;
699     trf69 = TRegA[pointer];
700     trf6901.a_word_h = (unsigned int)(trf69>>16);
701     trf6901.a_word_l = ((unsigned int)trf69);
702     program_TRF69(trf6901.a_word_h,trf6901.a_word_l);
703 }
704
705 /*                                     program_RX_mode_1
                                     */
706                                     /* used to be */
707 /* **** program_DDCl_send ****
                                     */
708 ; purpose: programs B register for reception (see further settings below)
709 ; (Mode 1, B-word)
710 ;
711 ; ****
                                     */
712 void program_RX_Mode_1(void)
713 {
714     unsigned long trf69;
715     int pointer;
716     pointer=(int)f_sel;
717     trf69 = RRegB[pointer];
718     trf6901.b_word_h = (unsigned int)(trf69>>16);
719     trf6901.b_word_l = ((unsigned int)trf69);
720     program_TRF69(trf6901.b_word_h,trf6901.b_word_l);
721 }
722
723 /* **** program_WORD_C ****
                                     */
724 /* purpose: programs word C (see further settings below)
                                     */
725
726 void program_WORD_C(void)
727 {
728     unsigned long trf6901_word;
729     trf6901_word = TRF69_Cword;
730     trf6901.c_word_h = trf6901_word>>16;
731     trf6901.c_word_l = trf6901_word;
732     program_TRF69(trf6901.c_word_h,trf6901.c_word_l);
733 }
734
735 /* **** program_WORD_D ****
                                     */
736 /* ****
                                     */
737
738 void program_WORD_D(unsigned long trf69)
739 {
740
741     trf6901.d_word_h = (unsigned int)(trf69>>16);
742     trf6901.d_word_l = (unsigned int)trf69;
743     program_TRF69(trf6901.d_word_h,trf6901.d_word_l);
744 }
745
746
747 int GetFsel(void)
748 {
749     int in0, in1,in2;
750     in0 = P3IN&FSEL4;
751     in0<=3;
752     in1 = P2IN & (FSEL3+FSEL2+FSEL1); // Get remaining select

```

```

753     in0|=(in1>>5);                // align FSEL3 and FSEL2
754     in1 = P2IN & FSEL1;          // align FSEL1
755     in0 |= in1>>3;
756     in0=~in0;                    //reverse polarity
757     in1=in0 & 0xf;
758     in0 = (in1>>3)&0x1;
759     in2 = (in1>>1)&0x2;
760     in0 |=in2;
761     in2 = (in1<<1)&0x4;
762     in0 |= in2;
763     in2 = (in1<<3)&0x8;
764     in0 = in0 |in2;
765     return in0;
766 }
767
768 interrupt [TIMERB0_VECTOR] void TimerB0(void)
769 {
770 }
771 /*****
772  Timer_B1 LED and push button timer
773 *****/
774 interrupt [TIMERB1_VECTOR] void Timer_B(void)
775 {
776     switch( TBIV )
777     {
778         case 2:                    /* COR1 */
779             TBCCR1=(TBR + 0x7800) & 0xffff; //turn on the interrupt
780             if(ms100_cnt--==0){
781                 TBCCTL1&=0xffed; // timed out turn off further INT
782                 ComFlag|=RF_timeout_flag; //enable function as RF timeout
783             }
784             break;
785         case 4:
786             TBCCTL2&=0xffed; //round trip ACK timer. If ACK is not received
787             within this time, ACK
788             ComFlag|=RF_timeout_flag; // monitor is terminated.
789             //P4CUTA-->stdb_time900; //disable the chip to get rid of noise
790             break;
791         case 6:                    /* COR3-Sampling period timer */
792             break;
793     }
794
795 void DisableButtonInt(void)
796 {
797     P1IE &=0xf0;
798 }

```

Appendix E - UNIVERSITY HONORS PROGRAM
SENIOR PROJECT - APPROVAL

Name: _____

College: _____ Department: _____

Faculty Mentor: _____

PROJECT TITLE: _____

I have reviewed this completed senior honors thesis with this student and certify that it is a project commensurate with honors level undergraduate research in this field.

Signed: _____, Faculty Mentor

Date: _____

General Assessment - please provide a short paragraph that highlights the most significant features of the project.

Comments (Optional):