# TABLE OF CONTENTS

Abstract .......................................................................................................................... 1

Introduction .................................................................................................................... 1

Software Radio .............................................................................................................. 2

  Software Radio Reception .......................................................................................... 3

  Software Radio Transmission ...................................................................................... 3

RFID Tags ....................................................................................................................... 3

  ISO 15693 .................................................................................................................... 5

The USRP Hardware ...................................................................................................... 6

  DAC and ADC ............................................................................................................ 6

  Altera FPGA ............................................................................................................... 7

  Daughterboards ........................................................................................................... 8

Design ............................................................................................................................. 8

Software ......................................................................................................................... 8

  Signal Processing Blocks ........................................................................................... 9

  Our Signal Processing Blocks - Transmission ............................................................ 10

  Layout Code - Python ................................................................................................ 15

Active Filters ................................................................................................................ 15

  Determining the Necessary Order ............................................................................. 15

  Determining the Poles ............................................................................................... 17

  Constructing the Filter ............................................................................................... 18

Reception ....................................................................................................................... 18

Antenna ......................................................................................................................... 19

Testing and Verification ............................................................................................... 21

  Verification of Transmission Software ....................................................................... 23

  Filter Construction and Testing ............................................................................... 26

  Antenna Tuning and Verification .............................................................................. 28

Conclusion ..................................................................................................................... 30

References .................................................................................................................... 32
ABSTRACT

This paper describes our senior design project in Electrical Engineering. The objective of this project was to design and implement an RFID (radio frequency identification) reader using an Ettus Research USRP (Universal Software Radio Peripheral). We used the ISO 15693 RFID protocol operating at 13.56MHz. In this we were required to work in multiple facets including: signal processing software, data control software, and hardware for antennas and an anti-aliasing filter. The following paper describes these details both textually and graphically.

INTRODUCTION

Radio Frequency Identification (RFID) is a means of using radio waves for the purpose of identifying goods, animals, and even people. It involves the use of devices, commonly called “tags,” which respond to radio waves with unique identifying signatures which can be used to identify devices across a wide number of fields in countless specific applications. The technological precursors to RFID date back to World War II, with the use of radio frequency to transmit data to and from transponders for the purpose of identifying whether or not an aircraft was a friend or a foe. The technology progressed to a point where in 1973, the first passive radio transponder was patented, the true precursor to the increasingly ubiquitous RFID tags we see today. There is already a broad range of applications for modern RFID tags, in fields from sports to medicine to farming, with technological improvements and price decreases making them more common every day.

The primary goal of our project was to implement an RFID reader which uses the ISO 15693 protocol. This protocol operates at 13.56MHz and uses Amplitude Shift Keying (ASK) in coordination with Pulse Position Modulation (PPM). The further details of the protocol will be explained in a later section.

Although our original intention was to devote most of our time to the software development of modulation and demodulation of the transmission to and from the RFID cards, we quickly realized that our project would entail a fair amount of hardware design as well. In order to accomplish what was needed we had to build two antennas that would resonate at the desired carrier frequency of transmission, and design and build a front-end filter. These ideas will be expressed more fully in subsequent sections.
Furthermore, in the end we realized that in order to accomplish transmission, although we verified that our transmission coding meets the timing constraints, we would need a power amplifier to strengthen the magnetic field being transmitted to power the RFID cards. The following sections describe the design, implementation, and verification process of our antennas, filter, and software as well as general comments on RFID, ISO 15693, and the USRP.

SOFTWARE RADIO

A software defined radio is a fairly new concept, though one which is quickly growing in popularity and importance. At its core, software defined radios work on the idea that the typical procedures performed by a radio in hardware, including filtering, modulation, detection, and more could just as easily be implemented in software as they are in hardware. Furthermore, this would have the advantage of developing a system which could be simply reprogrammed to allow for a new form of transmission and reception, whether the use of a new form of modulation, or an update to a manner of encoding, or whatever else a given situation might call for.

Progress in and application of software radio (or software defined radio) systems has increased very greatly over the last decade. The versatility and lower costs that software radio can afford are appealing to many areas of research and application. The ability to use one piece of hardware for many different applications by simply changing the software has long had strong appeal with groups such as the military, where the ability to simply reprogram a radio to work on an entirely different modulation or encoding frequency has immediate and obvious value, though use of software radios appeals more and more to a continually growing community of users.

In its most simplistic form, a software radio could consist of a simple Analog to Digital Converter/Digital to Analog Converter (ADC/DAC) combination, some kind of processor, and a radio frequency (RF) front-end that will appropriately filter and amplify the analog signals before digitizing them. This could be done, in other words, by using an RF front-end interfaced with a sound card and your personal computer. Since basic RF components are relatively inexpensive, and nearly everyone owns a computer with a sound card, the basic software radio can be constructed fairly inexpensively. Obviously, however, this simplicity will limit the use of the radio.
A more versatile and useful software radio will consist of the parts previously mentioned simply in a more robust form, specifically with regard to the ADC and DAC. Whereas the ADC on computer sound-cards typically sample at only 44.1 kHz, an ADC for a software radio would typically have a much wider bandwidth. In addition, special hardware and software are used which are designed specifically for signal processing and communication modulation and demodulation. The hardware and software that were used in our research and project are described in detail in their respective sections in this paper.

SOFTWARE RADIO RECEPTION

Since processing signals at large frequencies can require narrowly specialized and often expensive hardware, software radios make use of superheterodyne reception. That is, a local oscillator (LO) is used to transfer the signal to an intermediate frequency (IF). Following this, a digital down-converter (DDC) is used to convert the signals to the baseband complex envelope I(t) and Q(t) signals by multiplying the IF signal with local oscillator samples of sine and cosine waves. These wave samples are stored in memory in the software radio. These signals are typically put through a decimation process before being handed over to the user controllable software. This provides the digital signal data in a form which the signal processing software can easily handle.

SOFTWARE RADIO TRANSMISSION

Similar to the reception in software radios, digital converters are also used in transmission; in this case, a digital up-converter (DUC) is used. The DUC is used to convert baseband complex envelope I(t) and Q(t) signals to the desired transmission frequency. The specifics of both the reception and transmission paths are explained in more detail in their respective sub-sections in the section regarding the USRP hardware.

RFID TAGS

There are a broad number of different kinds of RFID systems, for a wide variety of purposes. The primary separation between types of RFID tags is whether the tags are active or passive. An active RFID tag is one which has a chip and antenna which are powered by a discrete power source, whether a battery embedded in the device,
or as part of a container or vehicle with some other form of power supply. Passive devices, on the other hand, are RFID tags which receive the power necessary for operation solely through the current induced in their antenna when they are exposed to the electromagnetic field of the RFID tag reader. Beyond the distinction between active and passive tags, RFID tags work in a number of different frequency bands, with their functions depending on the bands in which they operate. For example, Low Frequency RFID tags operate in the range between 30kHz and 300kHz, with typical standards placing the operating frequency at 125kHz. These cards are traditionally for very short range readings, generally a few centimeters, though compared to other cards they are very useful for tracking items with high water or metal content, as they are less susceptible to interference than the higher frequency RFID devices.

High Frequency RFID tags, however, are far more common, operating between 3MHz and 30MHz, with a typical operating frequency of 13.56MHz. These tags are by far the most common in modern use, with read ranges and data throughput which greatly outperforms that of the Low Frequency devices. Read ranges with High Frequency tags can extend farther than a meter with the newer tags, though the older HF-RFID tags suffer from the same short read distances that the LF tags do.

![Figure 1](image.png)

*Figure 1 – A typical example of an RFID tag, with attached antenna*

Ultra-High Frequency RFID tags, operating between 300MHz and 3GHz, tend to be limited to specialized applications, and are almost always active tags. They have seen use in standardized systems such as automatic toll readers for toll roads, trucking lanes, or taxis heading in and out of areas like airports. Their read ranges, particularly with strong active devices, can be longer than the other types of tags, making them far more suited than other tags for these kinds of tasks.
For this project, we selected to use High Frequency RFID tags which are designed according to ISO (International Organization for Standardization) standard 15693, which is a specification for “Vicinity Cards.” Vicinity cards are a kind of HF-RFID tag so called because they are designed to operate at a longer read distance, from 1-1.5 meters, than were previous generations of RFID tags called “Proximity Cards.” ISO 15693 cards operate at the 13.56MHz frequency that most HF cards share, but they do not require the same strength magnetic field that other cards need, only requiring 0.15A/m where Proximity cards require at least 1.5A/m. It is this lower necessary field strength which affords ISO 15693 cards their longer read ranges.

The communication scheme specified by ISO 15693 as the method of communicating with the card is a form of Amplitude Shift Keying (ASK) using Pulse Position Modulation (PPM). What this means is that messages are specified according to the position of pulses in a data stream, and those pulses are represented as shifts in the amplitude of the carrier signal. There are two data coding schemes accepted in the Vicinity Card specifications. The first is a “1 out of 4” data coding scheme, which means that out of every 4 pulse periods, a pulse period being 18.88µs, the placement of one 9.44µs pause is used to signify the data. The second is a “1 out of 256” data coding scheme, which means respectively that the placement of that same 9.44µs pause is counted once for every 256 pulse periods. This means that the “1 out of 4” scheme results in a much faster transfer of data between card and reader, but also that there are more pauses in the carrier wave, resulting in less overall power being delivered to the card.

The ISO specification also denotes two possible ways of having the card communicate back with the reader. The most common in implementation is an ASK modulation on a 423.75kHz subcarrier, where a logic zero is represented as the transition in a modulation period from high to low, and a logic one is represented as the transition in a modulation period from low to high. This is commonly called a Manchester encoding scheme. The ISO specification also allows for communication from card to reader to use Frequency Shift Keying, or FSK. The encoding scheme is the same, but instead of modulation being either high or low, it is either high frequency or low
frequency, with the low frequency being 423.75kHz, and the high frequency being 484.25kHz. An example of the Manchester encoding scheme can be seen below.

![Manchester Encoding](image)

**Figure 2 - Manchester Encoding**

### THE USRP HARDWARE

The following section provides a brief overview of some of the primary hardware contained within the USRP device that we used in our project.

### DAC AND ADC

The USRP contains two Analog Devices AD9862-MixedSignal Front-end Processors. Figure 3 below shows the basic functional diagram of the AD9862 – taken from the Analog Devices data-sheet. As seen in the figure, the device contains two analog-to-digital converters, and two digital-to-analog converters. Both the receiver and transmitter paths in the device have PGAs (Programmable Gain Amplifiers). These PGAs can be set from the GNU Radio software.

The ADC of the AD9862 outputs a 12-bit word and runs at 64 MS/s. The dual-channel feature of the ADC allows for it to be used to digitize a single channel real signal, dual channel real signals, or a signal channel complex signal with I and Q baseband components. The DAC takes in a 14-bit word and runs at 128 MS/s. Interpolation to meet the 128 MS/s run-speed requirements of the DAC is taken care of in both the AD9862 and in the FPGA. The
AD9862 is set to interpolate by a factor of four. Therefore, the FPGA on the USRP also contains an interpolator which interpolates the data rate to a rate of $128/4 = 32$ MS/s.

**FUNCTIONAL BLOCK DIAGRAM**

![Functional Block Diagram](image)

Figure 3 - Analog Devices - AD9862-MixedSignal Front-end Processors

**ALTERA FPGA**

Although we did not directly use it, the USRP has an Altera FPGA which can be user programmed for specific signal processing applications. The FPGA interfaces to the USB controller on the USRP; therefore, it controls the data-flow to the four daughterboard ports. Also, the FPGA performs the digital down-conversion in the reception path. The digital up-conversion is handled in the AD9862 device. The USB interface has a bandwidth of 32 MB/s.
DAUGHTERBOARDS

The USRP interfaces to a number of different daughterboards. For our project we used the BasicRX and BasicTX daughterboards from Ettus Research. These have a bandwidth from 1 MHz to 250 MHz. They are, just as their name implies, very basic. They do interface, via pins, with the FPGA digital I/O, SPI, and I2C busses. The BasicTX and BasicRX can be seen below.

![Figure 4 - The BasicTX and BasicRX Daughterboards](image)

DESIGN

SOFTWARE

GNU Radio is an open source project that provides software signal processing capabilities. Although GNU Radio is not exclusively used with the USRP, it does come with a plethora of libraries that work with the USRP. Also, it has been interfaced to the USRP to make using it fairly easy.

We used GNU Radio for all of the software development in our project. GNU Radio comes in several different packages for several different open source Linux distributions. For our project we installed a Cygwin
distribution over our Windows XP operating system. This proved to work well once we overcame all the small issues that typically accompany open source installations.

Much of this project consisted solely of the great amount of research necessary to get GNU Radio to communicate with the USRP. We developed both new signal processing blocks and the code to accomplish wiring those, and other, signal processing blocks together. In the following sections GNU Radio is examined more by investigating, at a high-level, how things are actually implemented; in other words, how signal processing blocks are written and wired together. We will show how this is actually done by looking at the code that we wrote for our own project.

**SIGNAL PROCESSING BLOCKS**

C++ is used to implement the signal processing blocks in GNU Radio. The signal processing blocks contain a few functions which are required for use as a traditional input/output block. In addition to those functions, you may write your own to perform the functions that are necessary for the block to work as you want it to. Each signal processing block contains a “work” function which is called by the scheduler when more data is needed. Also, the block can contain a “forecast” function which is called by the scheduler in order to determine the optimum timing for the system. The “work” function takes a number of parameters, including the number of output items desired, a pointer to the input items, and a pointer to the output items. When the scheduler calls this function, the signal processing block takes the input items, performs the desired processing on them, and produces the desired number of output items as specified by the scheduler. Then, the output items pointer is returned.

GNU Radio uses an interface known as SWIG to interface the PYTHON code, used for the abstract wiring, to the C++. Consequently, in addition to the normal configuration and make files that we had to wade through, we also had to provide a file which told GNU Radio how to interface between the PYTHON and C++. After a significant amount of time spent in the GNU Radio tutorials, we worked these peripheral files out.
**OUR SIGNAL PROCESSING BLOCKS - TRANSMISSION**

For our transmitter, we divided the signal processing into two main blocks. We developed one block that would contain the bytes of data that we wanted to send, and another that would upsample the data appropriately to match the DAC interpolation rate to output at 128 MS/s. The upsample block also takes care of the modulation index of the transmitted signal. In the design of our blocks we borrowed the basic structure from an unrelated open-source GNU Radio project, and altered the code to match our desired needs.

In the design of our transmitter, we decided to use “1 out of 256” encoding instead “1 out of 4” encoding, both of which are provided in the ISO 15693 protocol. Although “1 out of 256” encoding results in a much slower data-rate, 1.66 kb/s compared to 24.68 kb/s, it provides more consistent power to the card because the pauses occur far-less frequently.

*Data Source*

One of the obstacles that we ran into when developing the transmission software, was making the USRP output a constant signal in order to power the cards, while intermittently sending data. We obtained a significant amount of help from the GNU Radio message forum. In order to accomplish this, we use a FIFO to hold our output data before it is sent. Then, when more data is needed, the data at the end of the FIFO is transmitted. Whenever data is required (to keep a constant signal), and there is no actual data in the FIFO, the symbols corresponding to a full signal are stuffed into the FIFO so that it will output at full amplitude. Hence, unless true data is being sent, the reader outputs a full amplitude signal constantly. Below, in Figure 5, our “work” function is shown which performs this functionality.
int
gr_hdlc_router_source_b::work (int noutput_items,
  gr_vector_const_void_star &input_items,
  gr_vector_void_star &output_items)
{
  unsigned char * outbuf = (unsigned char *) output_items[0];
  int i;

  for(i=0; (i<noutput_items) && (i<1000); i++)
  {
    if(d_fifo.empty()) //Stuff with ones
    {
      push_flag();
    }
    // Output next data symbol
    outbuf[i] = d_fifo.pop();
  }

  return i;
}

Figure 5 - Work Function in our Data Source Block

The push_flag() function is used to stuff additional ones into the FIFO if it is empty (there is no real data to be sent). If, however, the FIFO is not empty, then the next value is popped off and put in the output buffer.

We defined, in our code, the three symbols that are necessary to send simply as 0, 1, and 2. They are decoded and transcribed into actual samples in our upsample block. The full source of this block is included at the end of the paper, but the figure below shows a sample of how we defined our data by looking at the functions in our block which are used for the SOF (start-of-frame) and EOF (end-of-frame).
As can be seen, in both functions an array of ones is originally created. Then, based on what the protocol calls for the SOF and EOF, the symbols corresponding to a 2 and 0 are appropriately placed in the correct positions of the array. These arrays are then sent on to be stuffed into the FIFO.

In addition to these functions already mentioned, we implemented a CRC (cyclical redundancy check) function which took as a parameter the data to be sent, not including the SOF or EOF. It performed the protocol defined CRC and returned a two-byte value to be sent with the data. The function was given in the ISO 15693 protocol, and we verified that our implementation worked by using one of their examples and verifying that we got the same values. The function implementation is shown below in Figure 7.
Upsample and Symbol Define Block

In the block which not only upsamples our symbols but also defines our pauses, we implemented most of the code directly in the “work” function. The constructor to this block takes two parameters. One of the parameters specifies the input rate, and the other parameter specifies the desired output rate. As will be seen, the desired output rate varies depending on what the DAC interpolation rate is set to. Therefore, both the DAC interpolation rate, and the output rate of our upsampler are, effectively, functions of one another. In addition to all of this, based on the desired modulation index this signal processing block adjusts the amplitudes of the symbols being sent out appropriately. Also, in this block the timing of the symbol pauses are defined. So, it is in this function that we adjust the timing of the pauses in order to make sure that we are meeting ISO 15693 timing specifications.

The block upsamples by taking one input symbol, and outputting the correct number of samples for that symbol in order to meet the DAC output requirements and the timing requirements of the symbol. As an example, in our PYTHON we pass to the upsampler constructor an input frequency that is equal to our symbol rate
since we are simply passing the individual symbols from our data source to the upsampler. So, since our symbol period is 18.88us we pass 1/18.88us = 52.966 kHz to the constructor. Also, if we have the DAC interpolation rate set at 32, in order to obtain the required DAC output of 128 Ms/s, we need our upsampler to output at 128M/32 = 4 Ms/s. In order to accomplish this, the upsampler takes in one symbol and outputs the appropriate amplitude samples for that symbol while counting the time elapsed. Once the time elapsed has reached the period of a symbol, the next symbol is brought in. Figure 8 below shows part of our code which defines the pause of a symbol.

```
input_symbol = in[i];
if(input_symbol == 0) {
    if(elapsed_time <= (0.55*d_input_sample_time)) { //In order to meet ISO 15693 timing
        out[c] = 100; //Full Amplitude
    } else {
        if(MOD_INDEX == 100) {
            out[c] = zero; //Modulated Amplitude
        } else {
            out[c] = 59;
        }
    }
} else if(input_symbol == 2) {
    if(elapsed_time <= (0.45*d_input_sample_time)) { //In order to meet ISO 15693 timing
        if(MOD_INDEX == 100) {
            out[c] = zero; //Modulated Amplitude
        } else {
            out[c] = 59;
        }
    } else {
        out[c] = 100;
    }
} else {
    out[c] = 100;
}
```

**Figure 8 - Code Which Defines Modulation and Timing**

As seen in the code, the pauses are located in the back or front half of the symbols depending on whether the symbol is a 0 or a 2 respectively. Also, if the symbol is a 1 then the output signal is held at full amplitude for the entire 18.88us period. Furthermore, depending on whether the modulation index desired is 100% or 26% the signal is either modulated to 0 or the appropriate amplitude for the other index.
The PYTHON in GNU Radio can be thought of as the “wiring code” and is, in effect, the glue that holds the signal processing blocks together. Using PYTHON, graphs are created in which the signal processing blocks are the graph nodes, and the PYTHON creates the vertices between the blocks. Also, the main programs are run from PYTHON main functions. The PYTHON is used similar to any other object oriented language in that signal processing blocks are created as objects which can take in any number of specified parameters for their constructors. In addition, GNU Radio supplies relatively easy to use PYTHON function calls which simplify using the USRP for basic operations.

Figure 9 shows the entire block diagram that our PYTHON code defines. The data type conversions are performed in order to match the data types that our blocks use to the other GNU Radio blocks; GNU Radio also supplies blocks to perform these data conversions, which we used.

```python
def self.dst = usrp.sink_c(interp_rate=32), creates an object which handles the transmission of the USRP. One of the parameters that it can take is the interpolation rate for the DAC; we set this to 32 based on what our upsampler was outputting in order to meet the overall DAC requirement of 128 MS/s. Once all of the blocks have been made, they are wired together using the “connect” function. Along with this, there are several
```
functions used for setting up the USRP appropriately. One of these functions is “tune” which sets the center
frequency on the USRP that we are interested in. In our case, the center frequency was 13.56MHz.

ACTIVE FILTERS

One of the strengths of software radios is that many of the functions normally implemented in hardware
can be implemented through the use of software, including the modulation and filtering of the relevant signals.
However, as these functions are performed digitally, a layer of analog signal filtering is required in order to ensure
that the digital signals can be processed in the USRP. This is necessary to avoid complications with aliasing and
other analog signal effects which are a consequence of digitization. The more specialized USRP daughterboards,
the ones designed to work over very focused frequency ranges, come with built in filters and amplifiers for this
reason; this is part of the reason why the specialized daughterboards are more expensive. However, the Basic TX
and Basic RX daughterboards being used with the USRP in this project do not come with filters, and as a
consequence we had to design some to ensure proper reception of signals.

In order to design the filter, we first determined the general requirements of a filter for this project. We
know that the highest frequency signal we are interested in receiving is at 13.56MHz, +423kHz for the sideband
width, so our passband should extend through 14MHz. Furthermore, it was determined that we should achieve a
reasonable degree of attenuation, determined in this case to be approximately -60dB, by twice this frequency.

DETERMINING THE NECESSARY ORDER

We first had to determine the necessary type, and order, of filter to use in this project. To this end, we did
the following:

\[ \omega_p = \text{Passband edge (14MHz)}, \omega_s = \text{Stopband point (28MHz)}, \]

\[ \varepsilon = \sqrt{\frac{10^{\text{1dB (allowable ripple)/10}}}{} - 1} = 0.5088 \]
Epsilon determines the maximum variation in the passband, based on our desired allowable passband ripple of 1dB. Knowing this, we can compute the necessary order of filter to achieve the desired results.

Using the equation for the attenuation in a Butterworth filter, with the variables at their previously specified value, we find that we can solve for the order \( N \):

\[
-60 dB \leq \text{Attenuation}(N) = -20 \log_{10}\left(\frac{1}{1 + e^{2(\frac{\omega_p}{\omega_s})^N}}\right)
\]

This results in a found necessary order of 11 to achieve an attenuation of -60.3583dB by the determined stopband point. Unfortunately, this is an order which would be rather difficult to implement, and this particular application would not necessarily benefit from the flat passband response that a Butterworth filter would provide.

A Chebyshev filter, by contrast, has attenuation which can be determined through the following:

\[
-60 dB \leq \text{Attenuation}(N) = -10 \log_{10}\left(1 + e^{2 \cosh^2\left(N \cosh^{-1}\left(\frac{\omega_s}{\omega_p}\right)\right)}\right)
\]

This results in a found necessary order of 6 to achieve an attenuation of -60.2839 dB by the determined stopband point, which is far more achievable in practice than the Butterworth filter results.

**DETERMINING THE POLES**

The equation to determine the poles of a Chebyshev filter is as follows:

\[
Pole \ p_k = -\omega_p \sin\left(\frac{2k - 1}{N} \pi\right) \sinh\left(\frac{1}{N} \sinh^{-1}\frac{1}{\epsilon}\right) + j\omega_p \cos\left(\frac{2k - 1}{N} \pi\right) \cosh\left(\frac{1}{N} \cosh^{-1}\frac{1}{\epsilon}\right)
\]

For \( k = 1,2,\ldots,N \)

We expected this to result in 3 pairs of complex conjugate poles, as the filter is of an even order and consequently all of the poles will be complex conjugates. The final poles for our system were:

\[
p_1, p_6 = -(4.132 \pm j86.355) \times 10^6
\]

\[
p_2, p_5 = -(1.1289 \pm j6.3216) \times 10^7
\]

\[
p_3, p_4 = -(1.5421 \pm j2.3139) \times 10^7
\]
Each of these pairs of poles is to be used in a 2\textsuperscript{nd} order filter, to be cascaded with the others to give a 6\textsuperscript{th} order filter with the desired response. To translate the pairs of poles into filter stages, we must determine 2 attributes from each pair: the pole frequency $\omega_o$ and the pole quality factor $Q$. The pole frequency can be interpreted from the complex plane as the distance from the origin to the pole, where the quality factor can be interpreted as the distance along the real axis. This can be seen below.

![Diagram of poles and complex plane](image)

Figure 10 - How the pole frequency and pole quality factor can be derived from the poles themselves

**CONSTRUCTING THE FILTER**

After finding the properties needed to construct the circuit, we had to decide on a filter topology. We decided to use the Antoniou inductance-simulation circuit. We chose this topology because it tends to be far more tolerant of non-ideal opamp and circuit construction properties than many of the more traditional layouts, such as the Sallen Key and Biquad topologies. The theory behind the inductance-simulation circuit, invented by Andreas Antoniou, is the creation of a circuit which simulates the effects of an inductor through the use of resistors, capacitors, and opamps. This can then be used, in conjunction with a resistor and capacitor, to duplicate the effects of a standard, second order LCR resonator filter in a more versatile active filter format.
The Antoniou inductance-simulation circuit can be seen below, and is implemented as follows:

\[
Z_{in} \equiv \frac{V_1}{I_1} = \frac{1}{sC_4R_1R_3R_5/R_2}
\]

**Figure 11 - The Antoniou inductance-simulation circuit.**

When this “inductor” is placed in parallel with a resistor and capacitor, the pole frequency and quality factor can be used to calculate the resistance and capacitance values needed to create a filter with the desired parameters. Placed in parallel with a resistor R6 and a capacitor C6, we find the following:

\[
\omega_0 = \frac{1}{\sqrt{LC_6}} = \frac{1}{\sqrt{C_4C_6R_1R_3R_5/R_2}}
\]

\[
Q = \omega_0 C_6 R_6 = R_6 \sqrt{\frac{C_6}{C_4}} \frac{R_2}{R_1R_3R_5}
\]

These equations were used with the pole frequencies and quality factors which were earlier derived from the pole locations of our filter to determine the necessary circuit for our 6-stage active filter.

**RECEPTION**

When first beginning the project, we thought it best to make sure our grasp of the transmission principles used was solid; to this end, we used LabVIEW to simulate the transmission and reception of ASK signals from the perspective of the card. To this end we decided to simulate the card response set to one sub-carrier frequency in
LabVIEW in order to understand some of the basic things that needed to be accomplished in order to demodulate the card response. This section describes the LabVIEW code.

In order to generate a simulation of the card type of coding that is used in the card response, we used a for-loop which generated one hundred random bits. A conditional statement within the larger for-loop determined what type of signal to generate based on the ISO 15693 card response protocol for sending a 1 or a 0. The protocol uses a Manchester encoding which uses a modulated time and unmodulated time in each bit sent. The following figure shows the protocol, and our simulation of sending a bit of value 1. As can be seen, it is an unmodulated period followed by a modulated period. The sampling rate used in LabVIEW was 50Ms/s.

![Figure 12 - (Top) LabVIEW simulation of logic 1 (Fs = 50Ms/s), (Bottom) ISO 15693 protocol for logic 1](image)

On the receiver side, we run the incoming signal into a series of envelope detectors. The first envelope detector removes the 13.56MHz carrier signal, and the second envelope detector removes the 423.75 kHz sub-carrier frequency. Figure 13 shows the original bits graphed with the final envelope output.
Since we know the bit rate of the ISO 15693 protocol, because we actually set it from the reader, and we know the sampling rate, then we know how many samples per bit there are. Given this, we can easily take the appropriate samples from each bit in order to determine what bits were sent. We perform this in LabVIEW in a for-loop which simply pulls out the appropriate samples and compares them with a threshold level; based on the threshold we determine whether the value is a logic 1 or 0. Although in GNU Radio there would undoubtedly be other issues that need to be resolved, and although we may not be able to perform the demodulation in this exact way, this does give a good illustration of one possible way of demodulating the incoming signals from the card.

**ANTENNA**

In order to maximize our transmission power and reception capabilities we needed two antennas: one for transmission, and one for reception. The main design for these antennas was borrowed from a Texas Instruments antenna design cookbook for RFID applications. Furthermore, we received very significant help from Dr. Barry Speilman of Washington University. Also, because the department lacked the specialized equipment needed to properly tune and match our antenna, we had to use a spectrum analyzer to get as close as possible. The following section describes our process of designing our antenna for the appropriate application.
Because the frequency that we wanted to use our RFID reader at is 13.56MHz, it has a wavelength of 
\[(3 \times 10^8)/(13.56 \times 10^6) = 22.12 \text{ m}.\] This makes it nearly impossible to construct an antenna that is even a quarter 
wavelength, much less half or a full wavelength. Consequently, in most practical RFID applications, antennas are 
used that are less than a quarter wavelength. This means that the antenna components do not have to be treated 
as transmission lines, but can just be modeled as lumped components.

The antenna design that we used is made from \(\frac{1}{2}\)” copper pipe. It is a square, 500mm X 500mm. The 
sides are soldered together using copper elbows. The antenna uses a matching method known as gamma arm 
matching. Our gamma arm is constructed from \(\frac{1}{4}\)” copper pipe. The gamma arm, as seen in Figure 14, creates 
another smaller loop which also produces a certain amount of induction. A variable capacitance and a resistance 
are then put in parallel at the open ends of the antenna for tuning purposes and adjusting the quality factor of the 
antenna. The tuning process is given in the “Testing and Verification” section.

![Figure 14 – The generalized design for our antenna](image)
TESTING AND VERIFICATION

VERIFICATION OF TRANSMISSION SOFTWARE

We tested that our software worked, in that it met timing and modulation specifications, primarily by using an oscilloscope. This allowed us to easily measure the timing and modulation of the pause. The following image, Figure 15, shows one of our pauses at a 100% modulation index. Also, Figure 16 shows the ISO 15693 timing requirements for a 100% modulation index.

Figure 15 - 100% Modulation Index
Testing the timing of our pause on the oscilloscope produced the following results: \( t_1 = 8.54\text{us}, t_2 = 8.16\text{us}, t_3 = 376\text{ns}, \) and \( t_4 = 216\text{ns}. \) Observing the timing constraints given in the above table, it can be seen that we have easily met these restraints. Furthermore, it is very easy to alter the timing of our pause by simply changing the conditionals in the “work” function of our upsampler.

Figure 17 shows a pause of our modulation index of 26%.
Another verification that we performed was ensuring that the time between pauses over the course of an entire data transmission was correct. We also did this using the oscilloscope. Since we were using “1 out of 256” coding, we were able to accurately predict the time between pauses based on the value of bytes. All of our measurements matched our predictions. For example, in our inventory request from the reader, the first two bytes that are sent are 0x36, and 0x01. Since this means, based on “1 out of 256” encoding, that these two bytes will consists of $512 - 18.88\text{us}$ periods together, we could determine the time that we expected to see between the pause for 0x36 and the pause of 0x01. The pause of the first byte would be found in the $37^{th}$ time slot, and the pause of the second would be in the $2^{nd}$ time slot. So, we expected a time between the pauses of $[(256 - 37) + 2] \times 18.88\text{us} = 4.17248\text{ms}$; and using the cursors on the oscilloscope to measure the time between pauses we measured 4.17272ms. The difference of 240ns can be attributed to the inaccuracies of measurement due to cursor use at these time periods on the oscilloscope. Figure 18 shows the two pauses associated with the first two bytes being transmitted.

Figure 18 - Pauses Associated with the First Two Bytes of Transmission
FILTER CONSTRUCTION AND TESTING

With the math having been performed for the development of the filters, it then became a matter of verification and testing of the design. The first step was to check the response of the transfer function in Matlab. This is shown below, in this instance with 20dB of gain.

![Figure 19 - The Matlab transfer function for the active filter.](image)

The filter was then implemented in Multisim to verify that the general topology of the filter was correct. It was known that the implementation in Multisim would vary from the Matlab transfer function due to the use of actual resistor values to which we have access in the Multisim layout. However, despite a few small changes such as an increase in the passband ripple, the filter simulation matched the Matlab transfer function fairly accurately. The Multisim layout and Bode plot can be seen below.

![Figure 20 - The Multisim layout of the active filter.](image)
For actual construction of the filter, selection of opamps was a very important concern; all of the opamps kept in regular stock in the labs have a gain-bandwidth product of around 1MHz, which is approximately 150x lower than what would be needed for implementation of these filters with a reasonable range of gain values. Because of this, we purchased TSH95 opamps, with a gain-bandwidth product of 150MHz. The filters were then built on a breadboard for an initial test of the circuit. The circuit at first performed oddly, providing distortion to the output signal due to what appeared to be harmonic interference, but this was due to its being more current intensive than expected; when supplied enough current from the power supplies, it performed reasonably well, though with again more ripple than the Multisim simulation suggested there would be.
Finally, we built the filter on protoboard, using proper jacks and binding posts for our inputs and outputs, as well as refining the design to use a pentiometer to allow for adjustable output gain on the final stage of the filter.

**ANTENNA TUNING AND VERIFICATION**

In order to cause a circuit to resonate at a particular frequency, the complex component of the impedance must be canceled out at that frequency. We used a couple different methods to tune the antenna to the appropriate frequency: 13.56MHz. Since we didn’t have a meter that would measure the impedance of the antenna, or even simply the inductance at 13.56MHz, we used different methods to approximate the inductance. First, we measured it using an LCR meter. Even though the LCR meter only made measurements as high as 1KHz, it still provided a ball-park range approximation for the inductance of the antenna. We measured an inductance of about 1.8uH. Using the equation for the required capacitance needed to cancel out the inductive component, we found that it would be approximately 72.5 pF. The second method that we used was simply given by an equation from TI that described, approximately, the inductance of the antenna that we built given certain parameters. This equation told us that we needed a capacitance of 101.3 pF.

We performed the entire tuning process primarily using a function generator that was capable of producing signals up to 15MHz. We drove our antenna with the function generator and used a simple wire loop
antenna as a reception antenna for the spectrum analyzer. Figure 23 shows the setup that we used for the tuning process.

![Antenna setup](image)

**Figure 23 – The antenna tuning process in action**

We tuned the antenna by driving the antenna with a 13.56MHz signal. Looking in the spectrum analyzer we tried capacitance values near 100 pF. Once we got to a point that decreased the signal power by adding just a small amount of capacitance, we put on a variable capacitor. The variable capacitor was used to fine tune the antenna. In the end, the antenna had the following capacitors in parallel: 22pF, 68pF, and two 4.7pF capacitors, and the variable capacitor set to 10.9pF. This resulted in the antenna having a total capacitance of 110.3pF, a value very near the calculated value. The extra required capacitance can be accounted for in many practical areas including stray capacitance in the soldered joints or the size of the antenna not being exact.

Another important aspect of any antenna is the quality factor (Q). We determined, based on the calculations below, that the Q needed to be less than or equal to 16. The quality factor is a measure of the center
frequency (the resonating frequency) divided by the bandwidth. The higher the Q, the higher the gain that is achieved at the center frequency; but a higher Q also means a smaller bandwidth. Since the RFID cards respond with sidebands at 13.56M +/- 423.75kHz, our bandwidth needed to be at least 423.75x2 = 847.5kHz. Therefore, we needed a quality factor no more than Q = 13.56M/847.5k = 16. The Q of our antenna is adjusted primarily by changing the damping resistor values. However, when the resistor is changed, the antenna also has to be slightly retuned. After a process of using different resistors and measuring the bandwidth that each of them gave, we found that a 3.7k and 4.8k resistor in series, or total of 8.5k, gave us a quality factor of Q = 14 once the antenna was properly tuned. We determined the actual bandwidth of our antenna by driving the antenna with frequencies above and below 13.56MHz until we located both -3dB points in the spectrum analyzer. Figure 24 gives an idea of this process.

Figure 24 - Measuring the Antenna Bandwidth Using Spectrum Analyzer

The gamma arm on the antenna was used to further match the antenna to 50 Ohms. We also did this by using the spectrum analyzer and looking for the peak. Furthermore, using an oscilloscope and making us of the
voltage-divider law, we looked at the signal without a load, and then with a 50 Ohm terminator; then we verified that the signal decreased as much with our antenna as with the 50 Ohm terminator.

**CONCLUSION**

In this project we learned many things in multiple areas of Electrical Engineering. These were demonstrated in our designing DSP software, antennas, and front-end filters all for the single purpose of implementing and RFID reader in a software radio. Software radios are an extremely powerful tool as evidenced all the more in our minds by the progress we made this semester. The things that we accomplished, regardless of the overall state of the project, are in and of themselves solid and useful building blocks toward the final goal and successful continuation of such a project.

The software that we wrote, for example, can make a useful contribution to the people in the GNU Radio community. The feasibility of an RFID reader implemented on the USRP using the GNU Radio software was a topic of discussion amongst users in the GNU Radio community, and we have made strides towards accomplishing a successful implementation of this project.

Furthermore, with our hardware development, both antenna and filter proved to be independently effective in accomplishing their respective tasks. Despite some initial difficulties in testing the antennas and getting them to full working capacity due to lacking the standard testing tools used in industry, we successfully developed methods which allowed us to accurately test and effectively tune the antennas to the desired resonant frequency and the necessary quality factor. Also, the filter, across a number of necessary revisions in order to compensate for emergent complications in construction, was eventually effective in the task for which it was designed.

Though we were disappointed that we were not able to see this project to completion over the course of the semester, our periods of extensive testing, diagnosis, and attempts at solutions taught us many of the practical engineering design processes and hazards. We feel that despite our disappointment at the lack of completion of our original intentions, the progress that was made is a solid step in the direction of those intentions.
REFERENCES


[10] Wikipedia:


   Manchester code - http://en.wikipedia.org/wiki/Manchester_code
