Software Defined Radio

2006 Senior Capstone Project (EE 452)

Luke Vercimak and Karl Weyeneth

Advisors: Dr. In Soo Ahn and Dr. Thomas L. Stewart

Bradley University ECE Department

Abstract

A software defined radio is a transmitter and receiver system that uses digital signal processing (DSP) for coding, decoding, modulating, and demodulating data. This project focused on using the IEEE 802.11a specification to create a software radio. The feasibility of using Mathworks' Simulink and Texas Instrument's Code Composer Studio to design, test, and prototype an OFDM software radio system on a Texas Instruments CDSK6713 DSP development board was studied. Among the subjects examined were communication with the board through real time data exchange (RTDX), quadrature amplitude modulation (QAM), orthogonal frequency division multiplexing (OFDM), frame and carrier synchronization, and issues with Simulink DSP code generation for prototyping.

Acknowledgements

There are some people that need thanked for making this project possible:

- Dr. In Soo Ahn and Dr. Thomas L. Stewart for advising, giving us guidance, and a push in the right direction when needed.
- Bader Al-Kandari for obtaining DSP boards for us to use with this project.
- Texas Instruments for providing the DSP boards for us to experiment with.

Table of Contents

I. Introduction

This project focused on the design and implementation of a digital software radio. Digital signal processing (DSP) was used to implement the transmitter and receiver. This allowed for greater flexibility and accuracy when designing the radio. The project is a scaled down version of the IEEE 802.11a standard that uses QAM (Quadrature Amplitude Modulation) with OFDM (Orthogonal Frequency Division Multiplexing) to create the coding and modulation scheme. The project also focused on rapid development and prototyping by using Simulink block diagrams to program the Texas Instruments TMDSK6713 evaluation board.

II. Functional Description

A software defined radio is a transmitter and receiver system that uses digital signal processing (DSP) for coding, decoding, modulation, and demodulation. This allows much more power and flexibility when choosing and designing modulation and coding techniques. The Texas Instruments TMDSK6713 evaluation board with the TMS320C6713 DSP chip was selected to implement the radio. The system functions are shown in Figure 1.

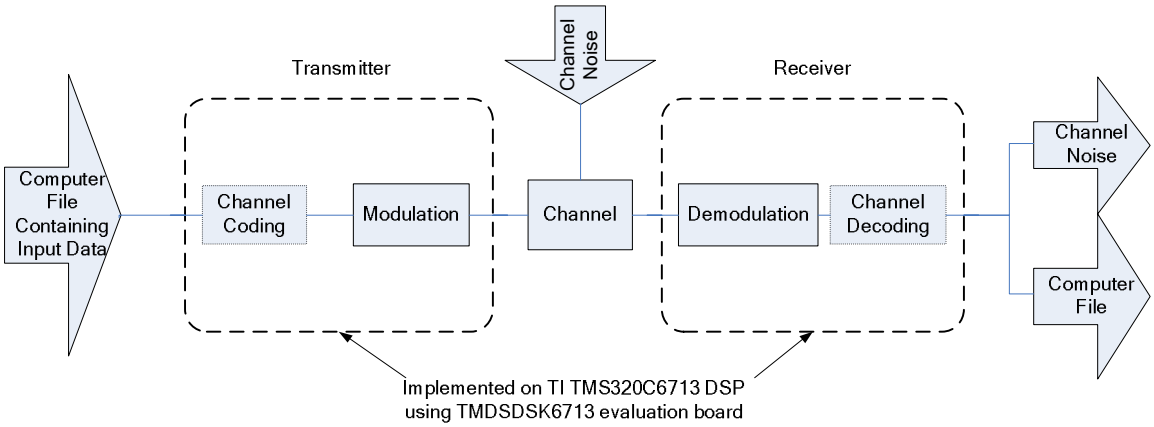


Figure 1 - I/O block diagram for transmitter and receiver radio systems

i. Inputs and Outputs

An overall block diagram for the software radio project is shown in Figure 1. The inputs to the system are a digital data source (computer file) and channel noise. The output of the system is the recovered input data. The recovered data should be received exactly as transmitted. This can be displayed on an oscilloscope coming out of the DSP evaluation board and/or stored on a computer file for further verification and analysis.

ii. Modes of Operation

The input from the digital data source will be sent into the transmitter. There it will have channel coding applied to provide protection from data corruption introduced by noise. This part will not be implemented in this project. After that, the encoded digital signal will then be modulated with an appropriate modulation technique and transmitted through the channel. An appropriate model and representation for the channel also needs determined. After this, the receiver demodulates the signal and applies appropriate channel decoding. From there the reconstructed digital signal will be available for further analysis.

Two development boards were used to construct the radio system. One board was designated for the transmitter and the other for the receiver. The system will be constructed and programmed entirely in Simulink using the embedded target for TI

C6000 Simulink library. Simulink generated the code based off of the model designed, and the code was then downloaded to the board through TI Code Composer Studio for testing.

III. Block Diagrams

Figure 2 - System Breakdown of the Software Radio

The input to the system will be digital data in a computer file. This data will be modulated by the transmitter and sent to the channel. The channel will introduce interference to the signal in the forms of attenuation, phase delay, and noise. At the receiver side, the signal will be demodulated and reconstructed to produce the original transmitted message.

i. Transmitter

The transmitter shown in Figure 3 will generate the signal that will be transmitted through the channel. The transmitter signal is constructed using demultiplexing, quadrature amplitude modulation (QAM), orthogonal frequency division multiplexing (OFDM), and up mixing.

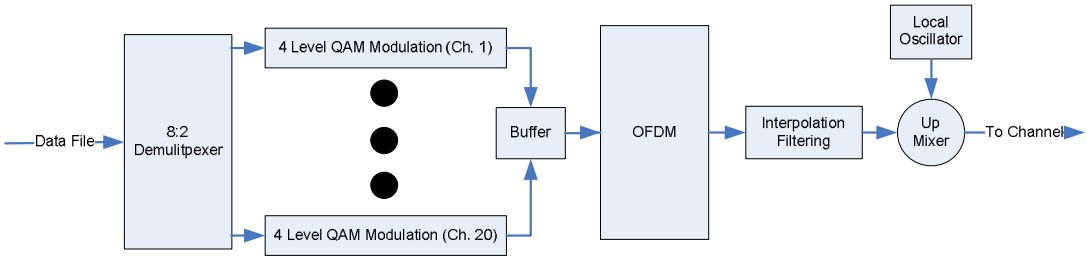


Figure 3 – Transmitter Subsystem Detailed Diagram

Demultiplexing & Modulation

The demultiplexing block takes a byte of binary data and then breaks the byte into four 2-bit streams. These 2-bit streams are each fed into a QAM modulation channel. Once the QAM channels have modulated the input data, a buffer collects a group of 20 QAM symbols that represent 5 bytes of data. The group of symbols is then passed into the OFDM block. The OFDM system multiplexes the QAM signals together to produce the final modulated output.

Interpolation

Interpolation increases the sampling rate and conditions the signal for transmission before it is modulated. This is implemented using a combination of up-sampling and filtering.

Up Mixer

Mixing is done to meet the bandwidth requirements of the channel. The up mixer increases the frequency of the OFDM signal by multiplying it by a greater carrier

frequency. The OFDM signal is imbedded in the carrier signal that the local oscillator produces. The output of the mixer is contained in bandwidth of the channel.

ii. Channel

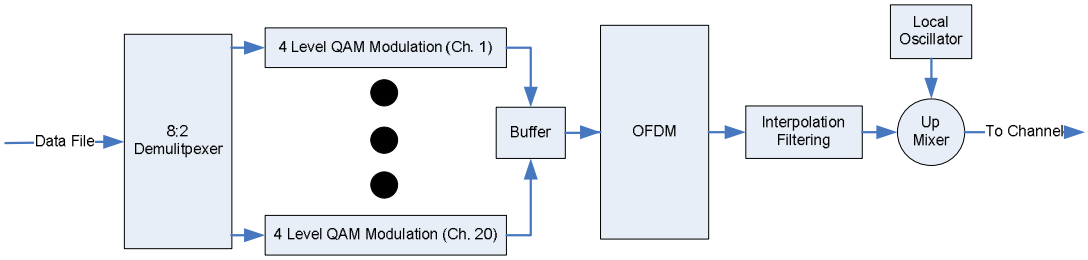


Figure 3 shows the detail of the channel block from Figure 1. The channel block implements a model of an actual transmission channel. Different parts of the channel model different channel effects on the transmitter output. These are channel gain, multi-path interference, and noise.

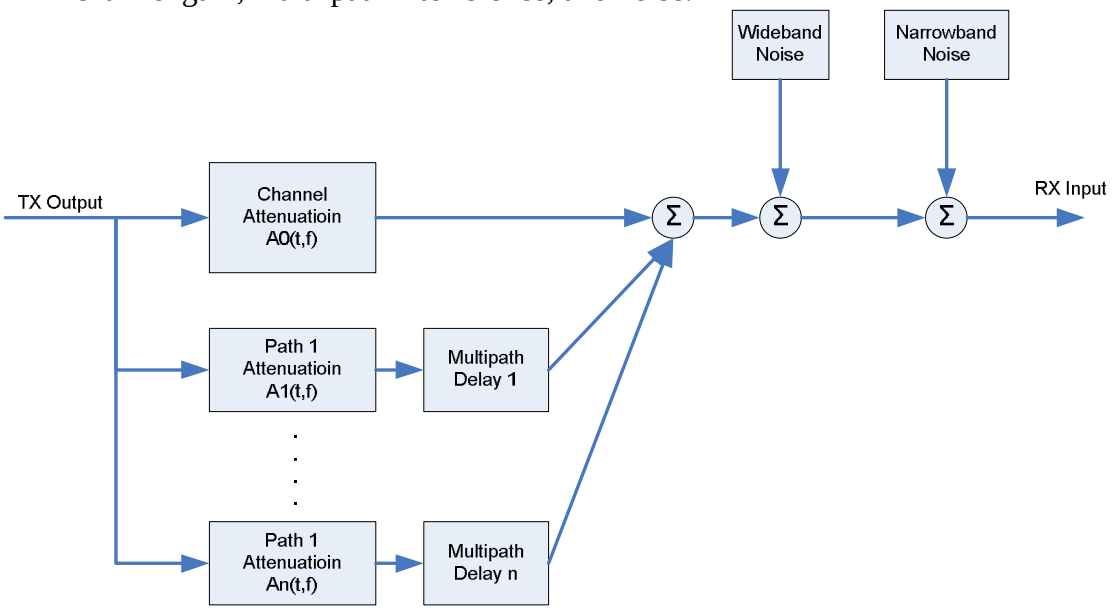


Figure 4 - Detail of Channel

Channel Attenuation

The channel attenuation models the attenuation or the gain effect that the channel has on the transmitted signal. This gain can vary with time and frequency.

Multi-path interference

The multi-path interference models reflections of the transmitted signal. These reflections arrive at the receiver at different times. Each one of these paths has its own attenuation that can vary with time and frequency.

Noise

Noise is also introduced into the signal. The noise is either specific to a limited frequency range (narrow band noise) or can affect the whole spectrum of the transmitted signal.

iii. Receiver

The receiver subsystem shown in Figure 5 recovers the sent message. The receiver extracts the carrier, symbol, and frame timing from the signal. It uses this information to extract the message from the phase, frequency, and amplitude noise of the channel. The receiver is constructed of the following parts; carrier synchronization, demodulation, symbol synchronization, and frame synchronization.

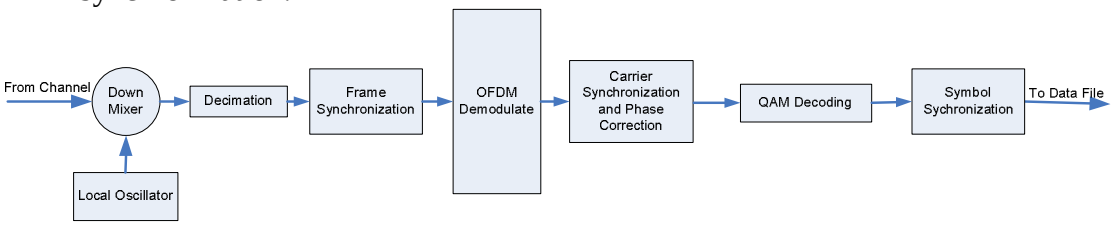


Figure 5 - Receiver Subsystem Detail

Frame Synchronization

The frame synchronization synchronizes the data frames. This aligns the start time of the message so the digital data can be interpreted correctly. This allows the compute file or message to be translated back into its original form.

Demodulation

The demodulation system consists of two parts: OFDM demodulation and QAM demodulation. The OFDM demodulation demodulates the signal into its constituent QAM sub signals. The QAM demodulation decodes the QAM carriers back into the bytes that were originally transmitted.

Carrier Synchronization and Phase Correction

The carrier synchronization subsystem corrects for frequency differences between the transmitter and the receiver. It also corrects for the phase delay introduced by the channel.

Symbol Synchronization

The symbol synchronization determines the time to sample the pulses coming from the QAM modulation. This allows the most accurate information to be extracted from the pulse stream. The output is the digital data that was originally sent into the system.

IV. Design Equations and Calculations

What follows are the theory, design and implementation of each of the radio subsystems detailed earlier.

i. QAM Modulation

Quadrature amplitude modulation is a modulation scheme that creates a modulation signal from a binary bit stream. The binary data is broken up into bit sets. Each bit set is represented on a constellation. The position of the point on the constellation representing the bit set is mapped to in-phase and quadrature components using the complex envelope. According to Couch [1], the complex envelope can be expressed as:

Equation 1 - Complex Envelope of QAM signal

$$g(t) = x(t) + jy(t)$$

In Equation 1, x(t) represents the in-phase and y(t) represents the quadrature component. Since the QAM in the software was at baseband frequencies, mixing of the in-phase and quadrature parts of the QAM symbol was not needed. However, for transmission of a QAM symbol it must be mixed to higher frequencies for transmission, and can be represented as:

Equation 2 - Equation defining a QAM signal

$$s(t) = x(t)\cos(\omega_c t) - y(t)\sin(\omega_c t)$$

Using the complex envelope notation in Equation 1, a four level QAM constellation was used (Figure 6) to represent the combinational pairs of binary values.

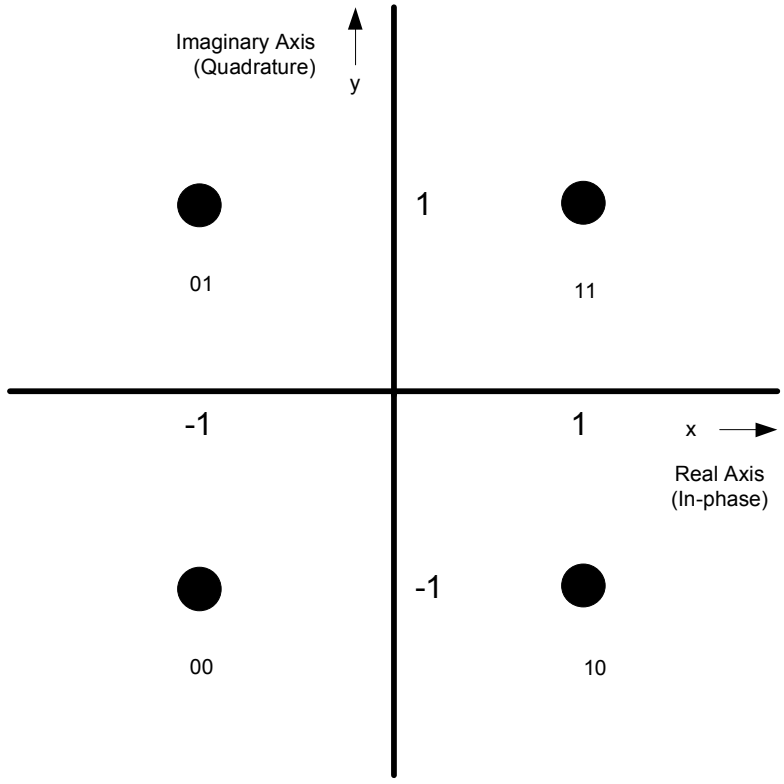


Figure 6: 4 Level QAM Constellation

For example, the QAM Constellation in Figure 6 would map the bits "10" to the symbol "1-j."

ii. QAM encoding

The design QAM encoder is shown in Figure 7. The data comes into the system one byte at a time. The byte of data is broken up into four pairs using the "extract bits" block. Once the bits are extracted, they are passed to a lookup table. The lookup table contains the QAM symbols representing each bit pair on the QAM constellation. The output of the QAM encoder blocks are complex numbers. The symbols are multiplexed together and converted into frame data. The buffer on the output of the convert to frame allows for multiple bytes to be placed into the OFDM spectrum.

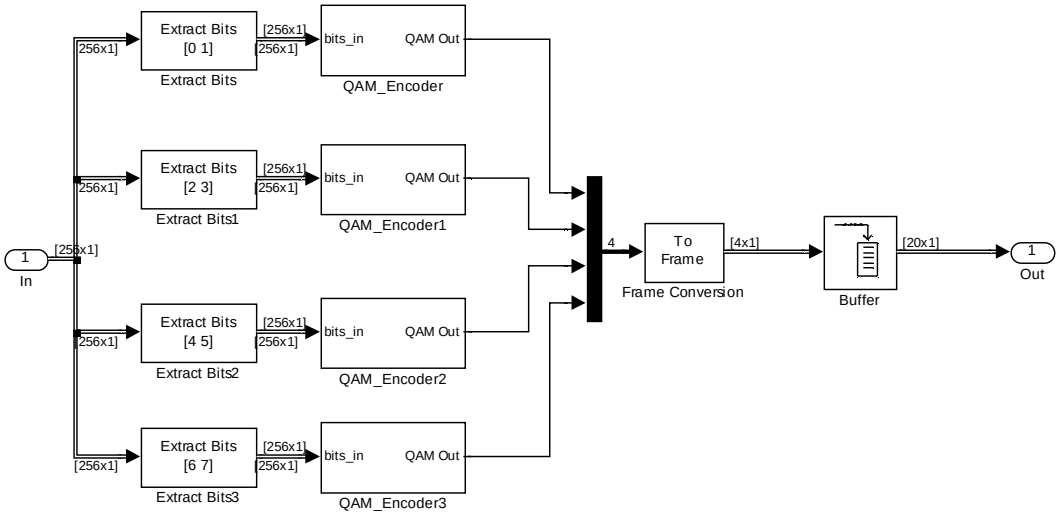


Figure 7 - QAM Encoder

iii. OFDM Modulation

Orthogonal Frequency Division Multiplexing or OFDM is a way to transmit many modulated signals at once by multiplexing them over a large number of frequencies. OFDM is different from normal frequency multiplexing because the individual frequency carriers are orthogonal to each other. This allows them to be closely space and not interfere with each other. Leon Couch in his book Digital and Analog Communication Systems [1] gives the complex envelop for OFDM in equation 1.

$$f_n = \frac{1}{T}\left(n - \frac{N-1}{2}\right) \text{Orthogonal Carrier Frequency}$$

$$\varphi_n(t) = e^{j2\pi f_n t} \text{ - Orthogonal Carrier Function}$$

$$g(t) = A_c \sum_{n=0}^{N-1} w_n \varphi_n(t) \text{ where } 0 > t > T, \text{ w}_\text{n} \text{ is}$$

$$\text{the nth element of the data vector } [w_0, w_1, ...w_{N-1}]$$

Equation 3 **- OFDM Complex Envelope definition**

T is the duration of data symbol on each carrier. According to Equation 3 this will create sub-carriers 1/T Hz apart. If the part in the parenthesis of the equation for f_n is ignored, and the equations are combined the result is the definition for the Inverse Fast Fourier Transform or IFFT. The ignored part is just a frequency shift and the results of this can be accomplished by properly placing the data in the FFT input vector.

Guard Interval

Prasad in OFDM for Wireless Communication Systems [4] states that a guard interval is needed to compensate for the effects of a multi-path channel. The legnth of the guard interval should exceed the maximum excess delay of the multi-path echo channels. This guard interval prevents the effects of the multi-path channel from corrupting the system due to the properties of cyclic convolution. As long as the guard interval is longer than the maximum delay of the multi-path channel, the corruption of the channel should be limited to the guard interval. This guard interval is created by appending a cyclic repetition of the FFT output to the FFT output's beginning.

Window and Window Interval

Since each OFDM symbol changes instantly, it can be thought of being windowed by a rectangular windowing function. A rectangular windowing function has a very large bandwidth due to the large sidelobes. This also creates problems with inter-symbol interference with adjacent OFDM symbols. To correct this problem, Prasad [4] and the IEEE specification [2] recommend using a raised cosine windowing function to reduce the size of the side lobes.

Implementation

The stream of QAM symbols is then sent into the OFDM modulation block "OFDM-4" shown Figure 8.

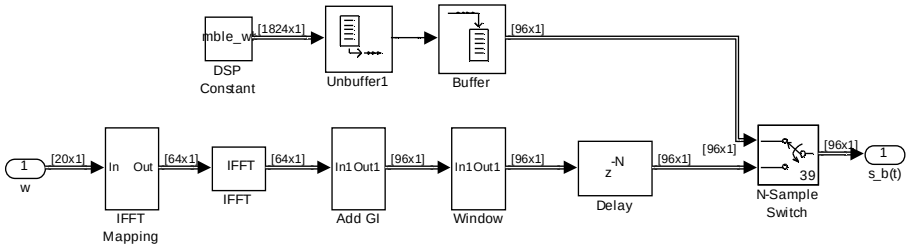


Figure 8 - OFDM Modulation

This block takes the stream of QAM symbols, maps them onto OFDM sub-carriers, creates the OFDM complex envelope with the IFFT, interpolates it to a higher sampling rate, and then breaks it into in-phase and quadrature components.

IFFT Mapping

The QAM data stream needs to be mapped to appropriate OFDM sub-carriers. A 64 point IFFT/FFT was chosen for this project because that is what is used in the 802.11a standard [2]. Figure 6 shows the OFDM sub-carriers as given by the 802.11a standard.

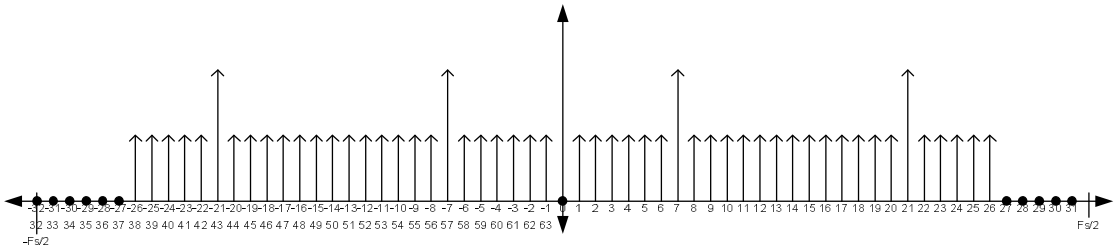


Figure 9 - 802.11a OFDM sub-carriers

The sub-carrier at 0 or DC is not used to prevent problems with RF circuitry and ADCs. 11 sub-carriers are not used on either end of the spectrum to make it practical to filter when up-sampling. Pilots are also inserted at carriers -21,-7, 7, and 21 to help with synchronization. Five bytes will be transmitted concurrently in the current design as shown in Figure 10.

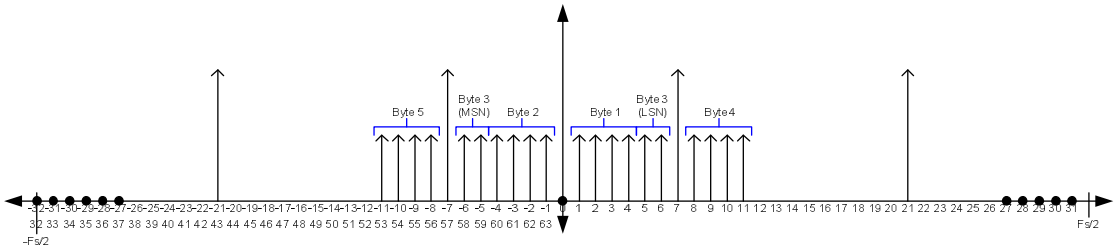


Figure 10 - Project current OFDM sub-carrier map

Insertion of pilot signals and mapping is in the Simulink block diagram in Figure 11.

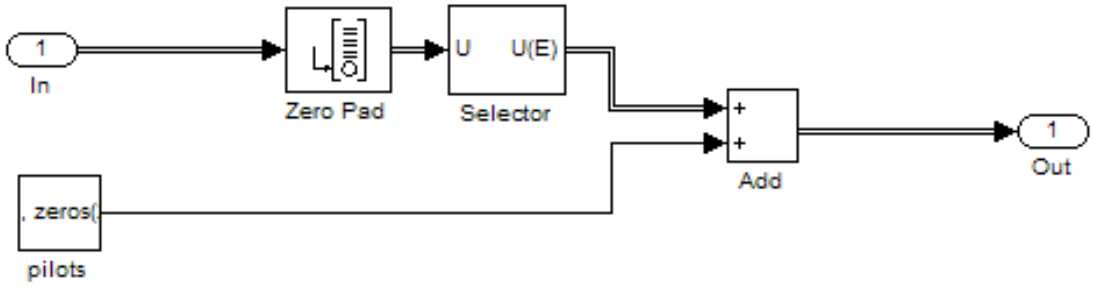


Figure 11 - OFDM sub-carrier mapping

The 20 QAM symbols coming into the block get padded with zeros to change the frame size from 20x1 to 64x1. The zero based frame index for each sub-carrier is shown in the bottom row of numbers in figure 6. Next the QAM symbols are moved from sub-carriers 0-3 to 1-4. After that, the pilot signals are inserted and the OFDM frame is sent to the IFFT.

Guard Interval and Windowing function

As Prasad indicates [4], a guard interval should be between 1/10 and ¼ the length of the symbol period. Since the symbol is 64 samples long, a guard interval of 16 samples was

appended to the front of the symbol. It was also chosen that the windowing function be eight samples long on either side of the OFDM symbol. This extended the OFDM symbol to a length of 96 samples. This was done by copying 24 symbols from the end of the FFT output, appending them to the beginning, copying 8 symbols from the end of the FFT output and appending them to the end of the FFT output. The windowing function was computed with the following code:

```
% Windowing Calculations for OFDM packet
k = -24:-17;
coef = 0.5*(1-cos(pi.*(k./125+0.192)./0.064));
w = [coef ones(1, 80) coef(8:-1:1)];
```

The coefficients were calculated with the equations on pg. 124 of Prasad's book [4].

iv. Interpolation

A higher sampling rate is needed to represent signals at a higher frequency. The interpolation subsystem increases the sampling rate of the transmitted signal so it can be later modulated onto a higher frequency and become a real signal.

Theory

Interpolation or up-sampling begins with a pre-sampled signal of x(t) that can be expressed as x(n). The signal y(m) is the signal that is passed out of the interpolation at the desired sampling rate. L denotes the up-sampling factor [5].

Equation 4 - Interpolation Equation

$$y(m) = \begin{cases} x(m/L); \text{ if } m/L \text{ is an integer} \\ 0 \quad ; \text{otherwise} \end{cases}$$

The pre-sampled signal x(n) and the up-sampled signal can be represented as:

$$x(n) = \{x(0), x(1), x(2), \ldots, x(n-1), x(n)\}$$

$$y(m) = \{x(0), (L-1 \text{ zeros}), x(2), (L-1 \text{ zeros}), \ldots x(n-1)\}$$

Then

$$X(\omega') = x(0) + x(1)e^{-j\omega} + x(2)e^{-j2\omega} + x(3)e^{-j3\omega} + \ldots$$

$$Y(\omega') = y(0) + y(1)e^{-jL\omega} + y(2)e^{-j2L\omega} + y(3)e^{-j3L\omega} + \ldots$$

$$\omega' = L\omega''$$

$$Y(\omega'') = X(\omega')$$

X(ω') can be shown as:

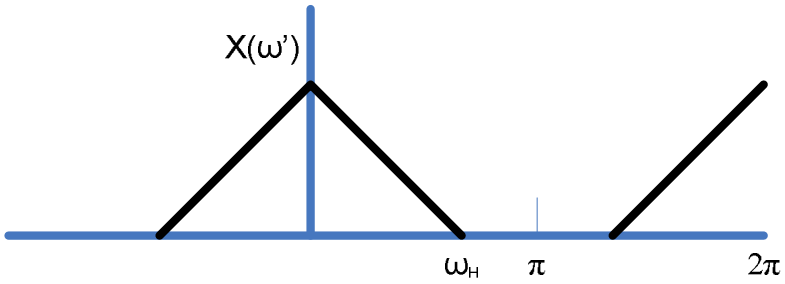


Figure 12 - Spectrum showing aliasing

Then $Y\left(\frac{\omega'}{L}\right) = X(\omega')$ where $Y\left(\frac{\omega'}{L}\right)$ can be shown as:

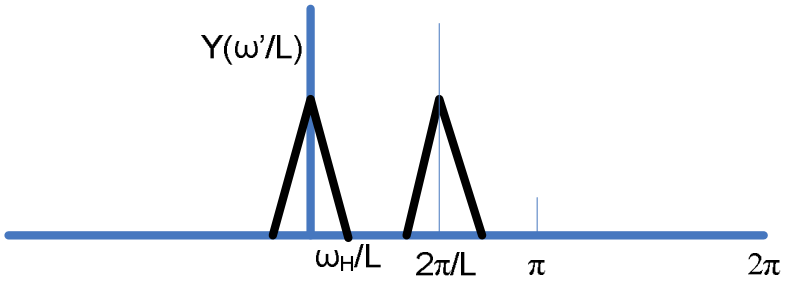


Figure 13 - Interpolation aliasing

Figure 13 shows that the up-sampled signal y(m) must be filtered. An interpolation filter is a low-pass filter used to filter the up-sampled signal. The cut off frequency of the filter is set at ω_H/L or π/L. After the filtering has been completed, the signal x(n) has been successfully up-sampled.

Implementation

In the interpolation block, the OFDM complex envelope output from the IFFT is up-sampled to a higher sampling rate. The Simulink block diagram used to do this is shown in Figure 14.

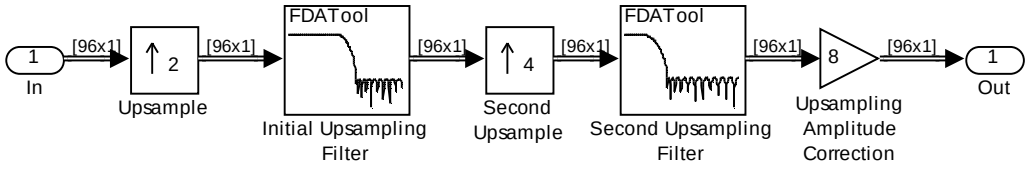


Figure 14 - Interpolation Simulink Block Diagram

The target sampling rate is 96 KHz. To reduce the filter order, this will be done in two stages. The first stage up-samples the data by 2, and the second up-samples by 4. Since the output frequency is 96 KHz, the input frequency must be 12 KHz and the mid sampling rate will be 24 KHz.

Interpolation Filters

All filters used to filter an OFDM signal need to have a predictable phase curve. This is because the QAM signal encoded on the OFDM sub-carriers uses the phase of the signal to encode data. Initially, elliptic IIR filters were used on the signal because of their computational efficiency. However, these filters have an irregular phase curve and will corrupt the QAM signal beyond recognition. Therefore, these filters were redesigned with linear phase or a constant group delay. Since all FIR filter only use zeros, the phase shift will be constant, equal to the order of the filter. Using the sub-carriers that are set to 0 on the ends of the complex envelope OFDM spectrum, the first up-sampling filter was designed. Since there are 11 signals total that are left zero, the transition region for the initial up-sampling filter is 11+1=12 times the spacing between the sub-carriers wide. The spacing of the OFDM sub-carriers is 187.5 Hz, which is the frame rate going into the IFFT. This was derived earlier when investigating OFDM. Therefore, the transition region is 2250 Hz wide with an 1125Hz band on either side of the original Nyquist frequency of the OFDM complex envelope, 6KHz. The pass band for this filter is 0-4875Hz and the stop band begins at 7125Hz as shown in Figure 15.

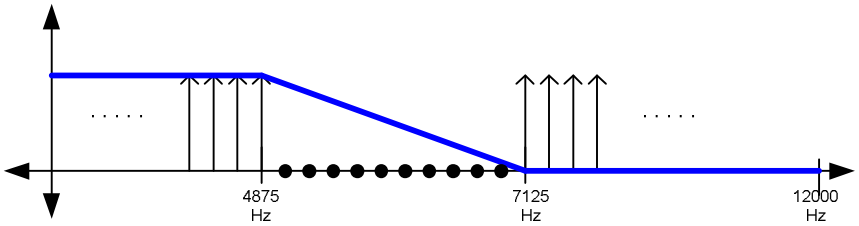


Figure 15 - Initial Up-sampling filter design

The pass band ripple for this filter is set to 0.1dB to prevent scattering of the QAM constellation. The stop band was set to 40dB to reduce noise. The second stage of interpolation up-samples the signal by 4. Therefore, the old alias that occurs around the old sampling frequency of 24 KHz needs to be filtered out. The bandwidth of the OFDM signal is approximately 12 KHz (including transition region null carriers). To filter this alias out, the filter needs to pass the original signal up to 12 KHz and be totally cut off before the alias starts at 6 KHz below 48 KHz. The pass and stop bands for the second interpolation filter are shown graphically in Figure 16.

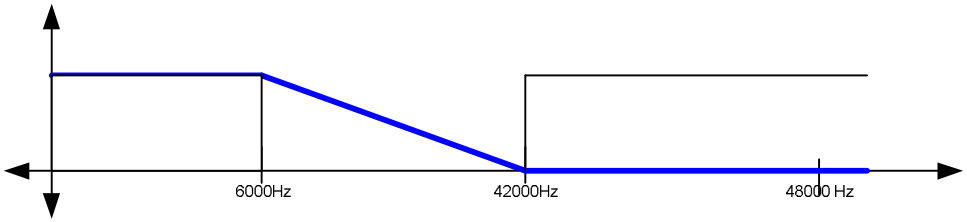


Figure 16 - Second Up-sampling filter design

The pass band ripple for this filter was set to 0.1dB to prevent scattering of the QAM constellation. The stop band was set to 40dB.

v. Quadrature Modulation

Quadrature modulation is needed for an OFDM signal to increase its frequency so that it can pass through the channel. It is also needed to change the baseband complex OFDM signal into a one that can actually be realized (real signal).

Theory

The OFDM signal coming into the quadrature modulation section is broken up into real and imaginary magnitudes. The signal that is passed to the channel can be represented by the equation where ω is the radian frequency carrier used to modulate the incoming signal above baseband frequencies [1].

Equation 5 - Quadrature Modulation Signal Representation

$$s(n) = RE(s'(n))\cos(\omega n) + IMG(s'(n))\sin(\omega n)$$

The cosine function is used to modulate the real part of the signal, and the sine function is used to modulate the imaginary part. The summation of the two signals forms the transmitter output signal.

Implementation

Figure 17 shows the Simulink design that was used to modulate the data from the interpolation system. Using Simulink, the complex symbol signal is broken up into two parts. The in-phase part is represented by the real part of the symbol, and the quadrature part is represented by the imaginary part. The magnitude of both the real and imaginary parts of the symbol are extracted and then mixed with there corresponding carriers. The two signals are summed and sent to the channel.

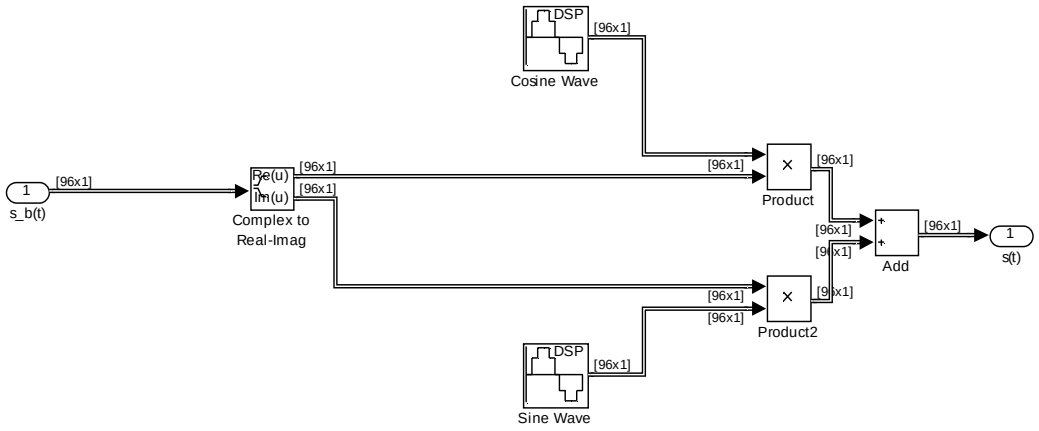


Figure 17- Quadrature Modulator Simulink Design

vi. Channel

The purpose of the channel model is to introduce corruption that is encountered when transmitting and IEEE 802.11a signal. This includes multi-path interference, signal attenuation, time delay, and Gaussian noise [1].

Theory

In theory an ideal channel would not affect the transmitted signal. However, in practice all channels do not even come close to being ideal channels. Therefore in the software radio, a channel was designed to account for multiple kinds of interference that occur to signal in the surroundings.

Implementation

The output signal from the transmitter is passed into the channel model in Figure 18. The signal is split into three copies. The top path represents the main signal, which is attenuated and delayed the least. The bottom two signals represent signal reflections due to the high attenuations they encounter. They are also delayed longer than the main signal. After the time delay operations, the signals are all added together with noise to create the receiver signal. This channel signal is then passed to the receiver.

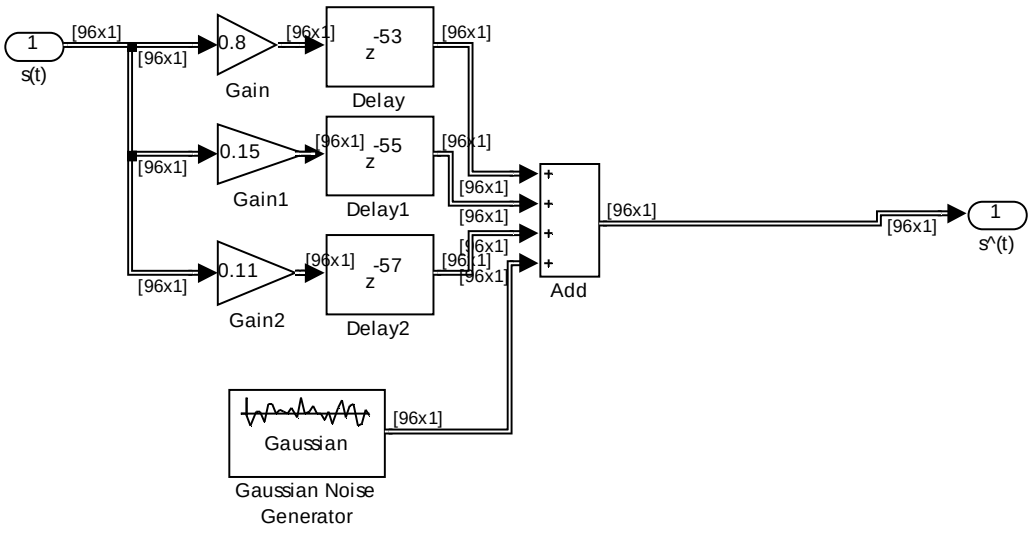


Figure 18 - Simulink Channel model used in the software radio

The frequency response of the channel model in Figure 18 is shown below in Figure 19.

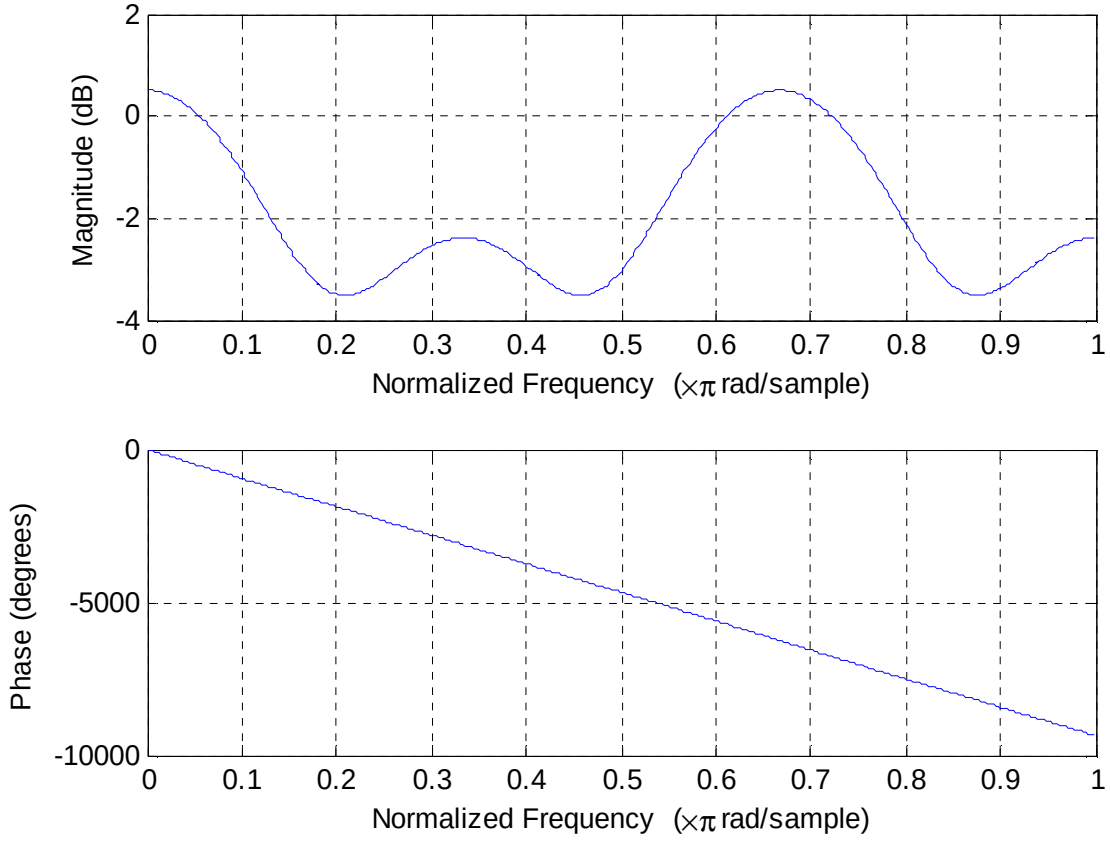


Figure 19 - Frequency response of channel

The magnitude response shows how the signal is attenuated at different frequencies. This attenuation is due to the multiple paths in the channel due to reflections and also to the noise that was added. The phase response is affected by the channel delays. The receiver of the radio was designed to compensate for this magnitude and phase corruption.

vii. Quadrature Demodulation

Quadrature demodulation moves the OFDM signal back to baseband from the intermediate radio frequency that it was modulated to for transmission.

Theory

The incoming signal is detected from the channel as the input to the quadrature demodulator. The cosine and sine generators mix with the incoming signal to bring the signal back down to baseband by subtracting the oscillator frequency from the incoming frequencies of the signal. The resulting in-phase and quadrature signals are brought together to form and OFDM symbol [1].

Implementation

The design in Figure 20 shows how the quadrature demodulation was designed in Simulink based off of the theoretical explanation above. The gain in the system was implemented to account for attenuation due to the channel.

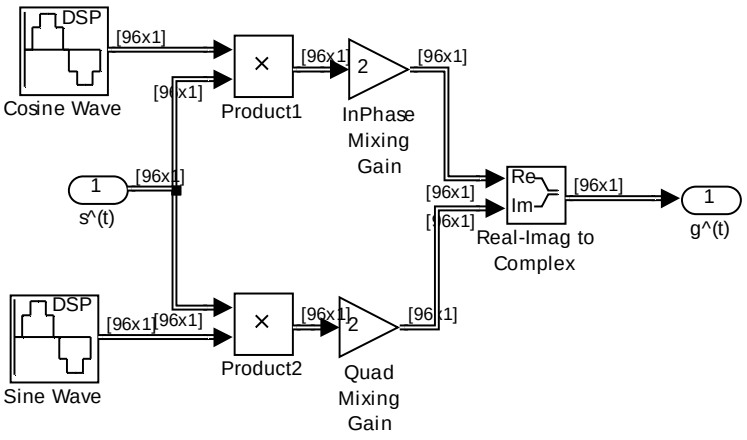


Figure 20- Quadrature Demodulator Simulink Design

viii. Decimation

The purpose of decimation in the receiver is to reduce the amount of data that represents the signal to improve signal processing performance. The information in one OFDM frame needs to fit in a single 64 point FFT. To do this, the signal is decimated back down to its original sampling rate of 12 KHz.

Theory

Equation 6 shows how down-sampling is determined.

Equation 6 - Equation for Decimation

$$y_D(n) = x(Dn)$$

In down-sampling, only every *Dth* Sample of the input x(n) is transferred to the output signal. While this is easily accomplished, a precaution needs to be taken before the actual down-sampling takes place. As shown in Figure 21, the frequency range from the previous Nyquist frequency to the down-sampled Nyquist frequency needs to be clear to prevent the spectrum from "wrapping around" when the signal is down-sampled. This wrapping around would combine the information in the frequency range from π/D to π with the information below π/D, corrupting the useful signal [5].

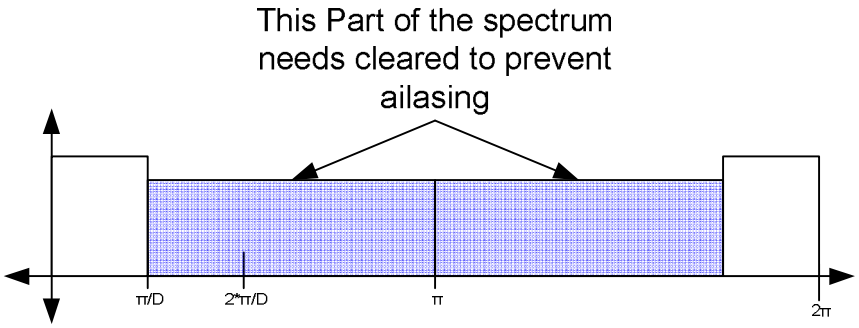


Figure 21 - Filtering needed for decimation

Implementation

In order to reduce the filter complexity, the filtering strategy shown in figure 19 was chosen.

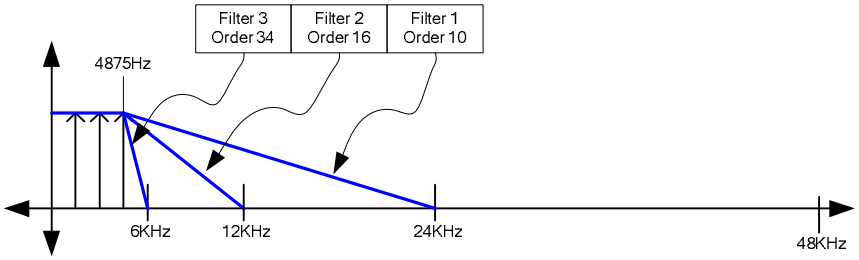


Figure 22 - Decimation Filter Design

In figure 19, the signal is down-sampled by 2 three times for a total down-sampling of 8. This is done so that the signal can be cleared between each down-sampling with a filter that has the widest possible transition region; thus reducing filter complexity. While doing this, the original signal from 0Hz to 4875Hz needs to be preserved. The final down-sampling filter must take advantage of the null carriers at either end of the spectrum shown in Figure 9 in order to be able to filter the signal while providing room for filter transition. Figure 20 shows the actual implementation of the decimation in Simulink.

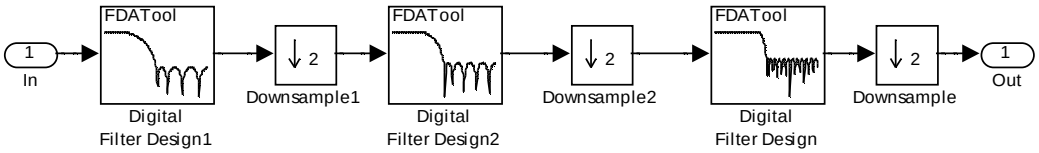


Figure 23 - Decimation Block

The transition regions for the filters are shown in Figure 22. The pass band for each filter was specified as 0.1 dB of ripple and the stop band was specified at 40 dB of attenuation.

ix. OFDM Frame Synchronization

In order to properly demodulate the transmitted data, the start of each OFDM frame needs to be found with reasonably accuracy. This is the task of the OFDM frame synchronization subsystem. The OFDM frame synchronization subsystem ignores input before the preamble comes in and then aligns the input directly after the preamble on frame boundaries. Figure 1 shows the need for frame synchronization.

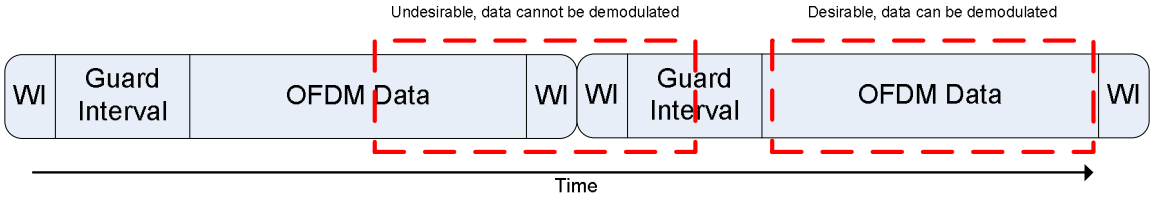


Figure 24 - Why frame synchronization is needed

Each OFDM symbol consists of a window interval, WI, a guard interval, the OFDM data, and then another WI. In order to properly demodulate the OFDM symbol, the FFT in the receiver needs to be filled with data from only one OFDM symbol. If it gets information from overlapping symbols, as shown in Figure 24, the data will be corrupt. Since every symbol is the same length, if the start of the OFDM symbols can be found, then the decoding of the symbols can be properly performed.

Theory

A pseudo-random sequence has a unique property in that its autocorrelation is very peaked. This is very useful in determining the location of a sequence in s signal and can be used to implement frame synchronization in communication systems [3].
The 802.11a specification defines an initial pseudo-random preamble sequence that is used for frame synchronization. What is given is 64 QAM symbol pseudorandom in a sequence show in Equation 7 [2].

Equation 7- 802.11a Short Preamble sequence in the frequency domain

$S_{-26,26} = \sqrt{(13/6)} \cdot \{0, 0, 1+j, 0, 0, 0, -1-j, 0, 0, 0, 1+j, 0, 0, 0, -1-j, 0, 0, 0, -1-j, 0, 0, 0, 1+j, 0, 0, 0, 0, 0, 0, 0, -1-j, 0, 0, 0, -1-j, 0, 0, 0, 1+j, 0, 0, 0, 1+j, 0, 0, 0, 1+j, 0, 0, 0, 1+j, 0,0\}$

This short sequence, S, is then padded with zeros on either side to bring it to $S_{-32,31}$, a 64 length vector. This padded vector is then fed into a 64 point IFFT to bring it to the time domain and then cyclically extended to 161 samples, according to specification. This set of 161 samples in the time domain, is then windowed to reduce inter symbol interference to create one short symbol. This short symbol is then repeated 10 times to create the whole short preamble, r, that will be used to find the start of the OFDM frame.
Sridhar Nandula and K Giridhar [6] detail a technique that can be used to synchronize with this preamble. Figure 25 from Nandula and Giridhar's article shows this technique.

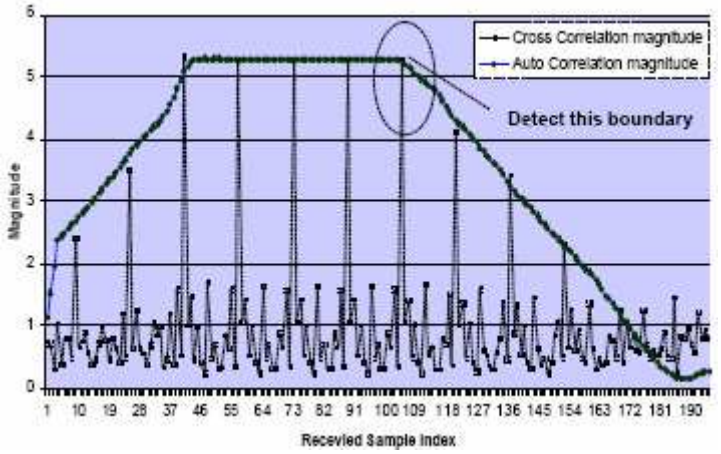


Figure 25 - **Detail of frame synchronization technique from Nandula and Giridhar's Paper**

The "dome" in Figure 25 shows the results of the signal being auto correlated with itself delayed by one symbol length (Equation 8).

Equation 8 - **Autocorrelation Equation**

$$A(n) = \sum_{k=0}^{N} r(k+n) r^{*}(k+n+L)$$

In Equation 8, L is the length of one short symbol, r is the whole transmitted preamble, and N is the length of the OFDM data (64 samples).
The result of Equation 8 is rather noisy. It is then smoothed with a moving average filter as shown in Equation 9.

Equation 9 - **Moving Average Equation**

$$Y(n) = \frac{1}{2l+1} \sum_{k=-l}^{l} A(n+k) + A(n)$$

In Equation **9**, l is the length of the moving average.

The spikes in Figure 25 are generated by Equation 10. In this equation, the whole preamble, r, is cross correlated with one short symbol, s. S is the number of FFT samples, as in Equation 8 and M is the number of short symbols that the cross correlation is averaged over.

Equation 10 **- Equation for Cross Correlation with the Preamble**

$$C(n) = \sum_{l=0}^{M} \sum_{k=1}^{N} r(l * N + k + n) s^{*}(l * N + k)$$

The start of the OFDM symbol can then be found by timing off of the last large spike inside the flat part of the dome.

Preamble Insertion

Figure three details the Matlab code that is used to generate the short preamble. This code creates a preamble as specified in annex G.3 in the IEEE 802.11a specification [2]. The short preamble is created with the Matlab code below:

```
% Preamble Short Sequence
short_seq = (sqrt(13/6))*[0, 0, 1+j, 0, 0, 0, -1-j, 0, 0, 0, 1+j, 0, 0, 0, -1-j, 0, 0, 0, -1-j, 0, 0, 0, 1+j, 0,
0, 0, 0, 0, 0, 0, -1-j, 0, 0, 0, -1-j, 0, 0, 0, 1+j, 0, 0, 0, 1+j, 0, 0, 0, 1+j, 0, 0, 0, 1+j, 0,0];
short_seq = [zeros(1,6), short_seq, zeros(1,5)];
short_seq = [short_seq(33:64),short_seq(1:32)];
short_seq_t = ifft(short_seq);
short_seq_r = [short_seq_t, short_seq_t, short_seq_t(1:33)];

% window the preamble
short_seq_r(1) = short_seq_r(1).*0.5;
short_seq_r(161) = short_seq_r(161).*0.5;

% Assemble the whole preamble
preamble = [];
for i = 1:10,
    preamble = [preamble, short_seq_r];
end
```

Figure 26 - Matlab code to create the preamble

This preamble is then inserted before the OFDM frames with the Simulink diagram shown below in Figure 27**.**

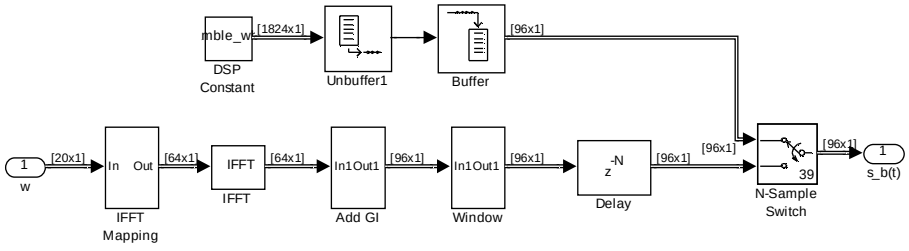


Figure 27 - Insertion of preamble

The preamble has random noise added to the front until it is a multiple of 96. The preamble ends up being 39 frames of 96 samples each when padded. The OFDM frames are then delayed by 39 frames and the signal is switched from the preamble to the OFDM signal after 39 frames**.**

Statistics Implementation

The method shown above will be modified slightly before implementing in the actual radio. First, the cross correlation symbols will not be averaged. Averaging the cross correlation symbols improves the method's performance under a frequency offset. It can be implemented later to improve the method's robustness if desired. The second thing that changed is the moving average. A larger moving average filter will be used than shown in Figure 25. This is done to match the sample rates in the Simulink program and will be further explained later. Next Equation 8, Equation 9, and Equation 10 were modified and filled in with actual values to create the Equation 11, Equation 12, and Equation 13 which were used for implementation of the frame synchronization.

Equation 11 - Autocorrelation equation used in implementation

$$A(n) = \sum_{k=0}^{160} r(k+n) r^*(k+n+161)$$

Equation 12 - Moving average used to smooth the autocorrelation

$$A_{ave}(n) = \frac{1}{128} \sum_{k=0}^{127} A(k+n)$$

Equation 13 - Cross correlation equation used in implementation

$$C(n) = \sum_{k=0}^{160} r(k+n) r^*(k)$$

The next thing that needed done is to figure out how to detect when the autocorrelation was "domed." This was done with comparing the autocorrelation to the input signal's standard deviation. When A(n) was greater than 7 times the standard deviation of the input signal, the receiver was "inside" the dome. This factor of 7 times was determined through visual calibration and statistically represents a value that the input signal's magnitude is less than 99.9% of the time. The last high peak inside the dome is what is used to synchronize the receiver. Figure 28 shows the part of the frame synchronization diagram that is used to do this.

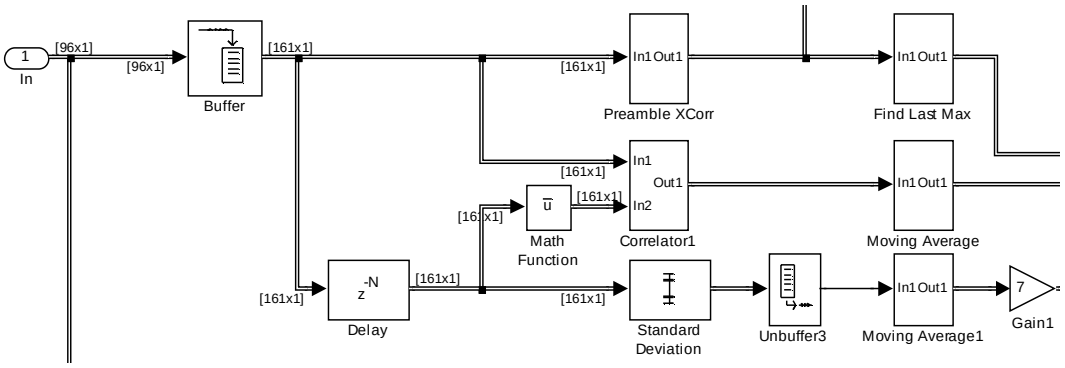


Figure 28 - The statistical part of the frame synchronization

The signal enters on the left side of the diagram. Here it is diverted for later alignment and fed into a buffer. This buffer is a delay line of the size of one short symbol. This signal is then fed into the preamble cross correlation and autocorrelation for processing. Figure 29 shows the implementation of a correlation system.

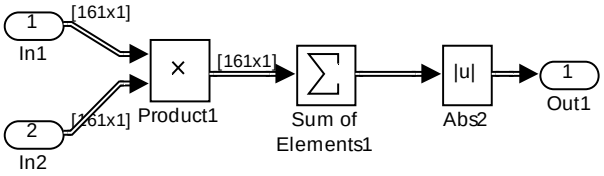


Figure 29 - Simulink diagram for correlation system

The cross correlation shown in Equation 13 is performed by connecting one input of Figure 29 to a locally stored copy of the conjugate of the short symbol . Equation 11 is implemented by delaying the signal by one frame, conjugating it, and feeding it into the other input of the other correlation system. The standard deviation of the delayed signal is also computed to determine when the autocorrelation goes high. The output of the standard deviation and the autocorrelation is noisy so they are fed through a moving average, as seen in Figure 30. The autocorrelation is fed through a moving average of length 128 and the standard deviation of length 64. This was done to make sure the standard deviation would rise before the autocorrelation and prevent false triggering when the receiver started to process data.

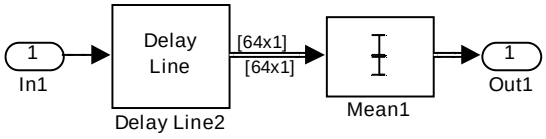


Figure 30 - Simulink Diagram for moving average

Figure 31 shows the output of the statistical part of the frame synchronization.

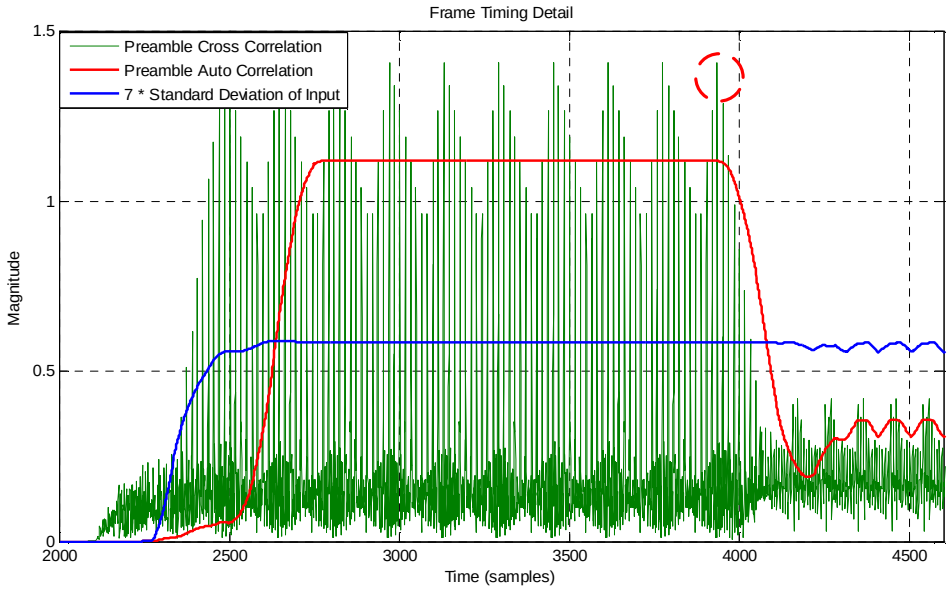


Figure 31 - Simulated Cross correlation, autocorrelation, and standard deviation of preamble

The circled area is the peak that that will be used for timing.

Peak Detection

The next thing that must be done is to detect the circled peak in Figure 31. This is done with the parts of the frame synchronization diagram shown in Figure 32. It consists of two main parts, the find last max block, detailed in Figure 33 and the pulse on preamble block, detailed in Figure 34.

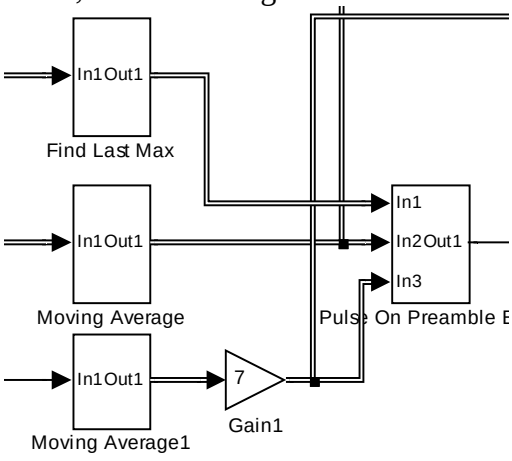


Figure 32 - Peak detection part of frame alignment

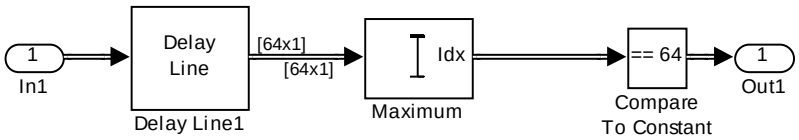


Figure 33 - Find last max Simulink Diagram

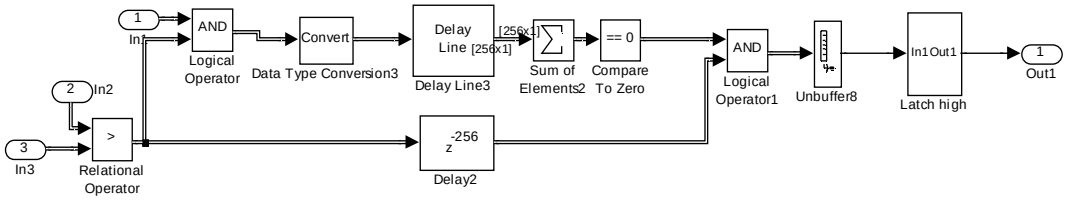


Figure 34 - Pulse on preamble end Simulink diagram

The find last max block outputs a high or 1 when the last sample in the frame or newest sample to enter the block is the new maximum. This will produce high signals on the increasing side of the 10 high parts of the cross correlation shown in Figure 31. This block will produce undesired spikes when not inside the "dome" in Figure 31. The pulse on preamble end block detailed in Figure 34 uses the max signal, autocorrelation signal, and standard deviation signal to create a signal that goes high when the OFDM symbols start. To generate this OFDM symbol start signal, the standard deviation signal and autocorrelation signal are compared to determine when the receiver is inside the "dome." The find last max signal is then anded with this signal to filter out all of the maximum signals that are not inside the dome. This is the blue signal in Figure 35. Next, the last samples of the maximums inside the dome are summed together. When this sum is 0, there are no more maximums left inside the dome, and the receiver has found the desired cross correlation spike. This is the green signal in Figure 35.

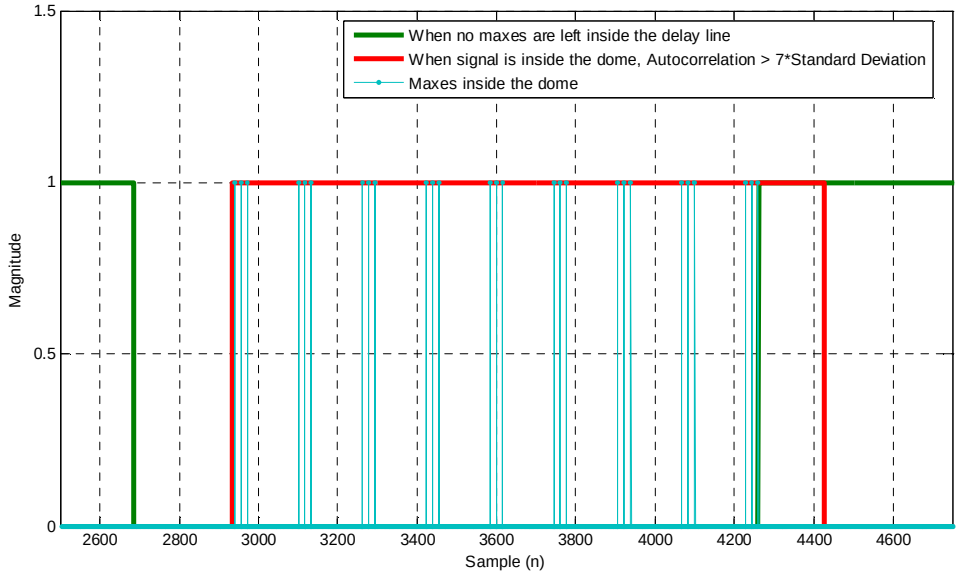


Figure 35 - Details of the pulse on end of preamble block

This signal is then anded with a delayed version of the signal that indicates when the autocorrelation is high or domed. This creates a pulse that is then latched to create a signal that is low until the end of the preamble is reached. The signal then remains high. This is the signal that is needed to perform the frame alignment. Figure 36 shows the magnitude of the input signal (red) and the start of the frame alignment signal (green). The frame alignment signal goes high as soon as the preamble is over.

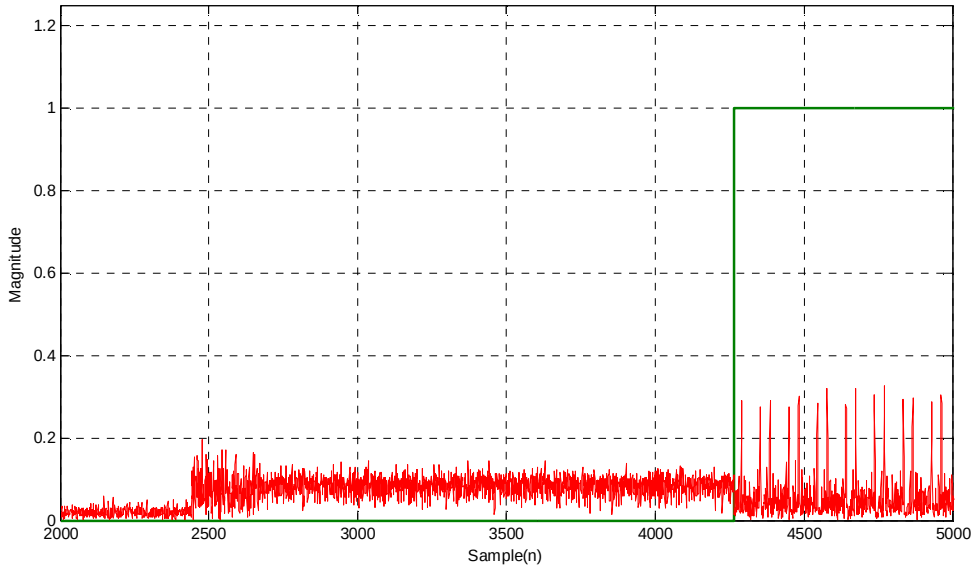


Figure 36 - Input signal and frame alignment signal

Frame Alignment

After the proper alignment signal is generated, the frames need aligned so that the frame contains one whole OFDM symbol that starts at the beginning of the frame. Figure 37 shows the parts of the frame synchronization used to do this.

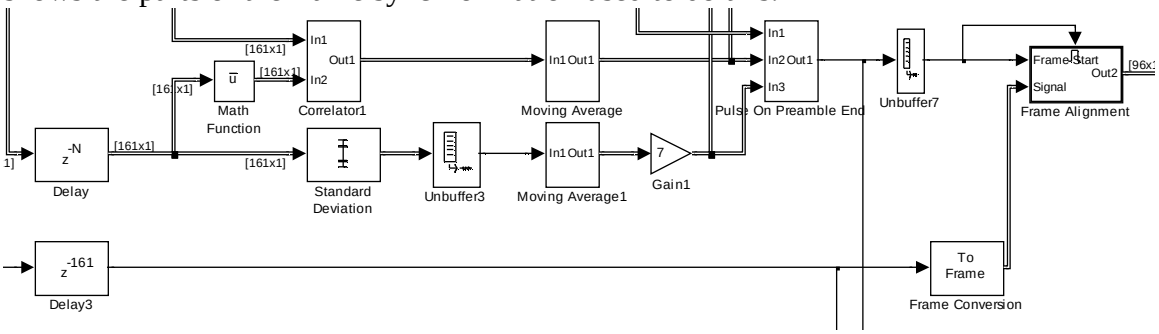


Figure 37 - Frame alignment part of frame synchronization

Delay 3 shown in Figure 37 is used to align the signal generated on the pulse on preamble end block and the original incoming data. This compensates for the delays caused by the frame alignment signal generation discussed earlier. Both the delayed signal and the frame alignment signal are then fed into the frame alignment block shown in Figure 38.

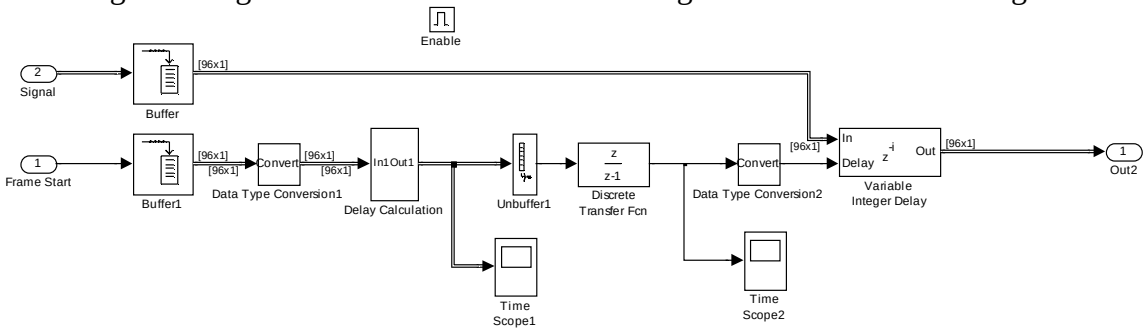


Figure 38 - Frame alignment block

The enable on the enabled subsystem brings the alignment to within one frame. However, Simulink starts buffering on the clock cycle before the enable goes high. Therefore a delay needs added to bring the start of the OFDM symbol to the start of the frame. This is done by taking a similar frame of the frame start signal and produced earlier. The number of positions that the first frame of this signal has that are one should be number of delays needed to delay the signal back to the start of the next frame. The blocks in Figure 39 do this by summing the frames of the alignment signal and then subtracting that value from 96 to find the number of zeros in the frame. When this value is greater than zero, the number of ones is then outputted for one clock cycle.

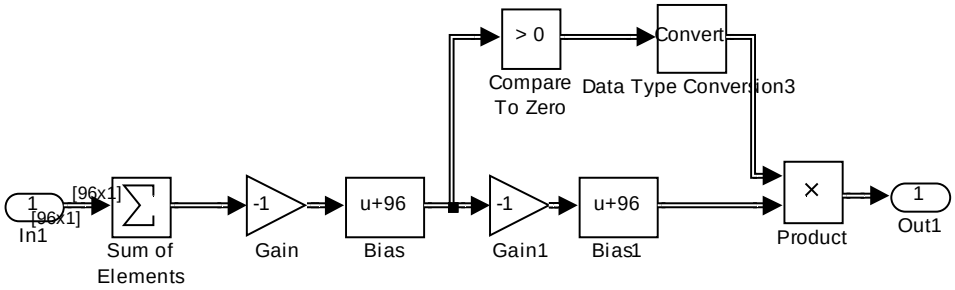


Figure 39 - Delay calculation block

The output of Figure 39 is then fed into an accumulator that will create a signal that will rise to the number of sample delays need and remain at that value while the radio is receiving data. A variable delay is then used to align the data as shown in Figure 38.

x. OFDM Demodulation

After each OFDM symbol is aligned on a frame boundary, the OFDM modulation needs to be undone so that the original QAM symbols can be operated on. This demodulation process needs to also take into account the effects that are added by the windowing, guard interval, and channel delay characteristics.

Theory

The QAM symbols can be recovered from the OFDM symbol by reversing the IFFT used to encode them on the OFDM symbol. This is done using a Fast Fourier Transform (FFT). The FFT will take the time domain signal and convert it back into the frequency domain where the QAM symbols were encoded [1]. Once in the frequency domain, the effects of the phase distortion can be compensated for.

Implementation

Figure 38 shows the implementation of the OFDM demodulation in Simulink.

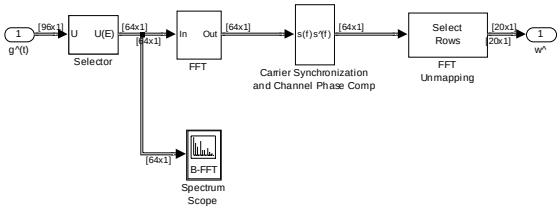


Figure 40 - Implementation of the OFDM demodulation in Simulink

The selector in the beginning discards the guard interval and window function that was added to the OFDM frame during modulation. This signal is then passed to the FFT block

for conversion to the frequency domain. After this is done, the signal is ready for phase correction.

xi. Carrier Synchronization and Channel Phase Correction

Once the QAM symbols are recovered from the OFDM demodulation, they need to be corrected for phase distortion resulting from buffering and delays in the transmitter and receiver, miss matches in the transmitter and receiver carrier modulation phase and frequency, and also channel phase effects. The linear relationship that this distortion has with the sub-carrier frequency is used with along with the known characteristics of the pilot signals to correct for these effects.

Theory

The pilot signals for the OFDM frame are inserted with a known phase 0 zero. The radio's FIR filters and delays have a constant group delay. This means that the phase of the filters and delay is proportional to the frequency of the signal going through it. Figure 41 shows the frequency response of a single sample delay.

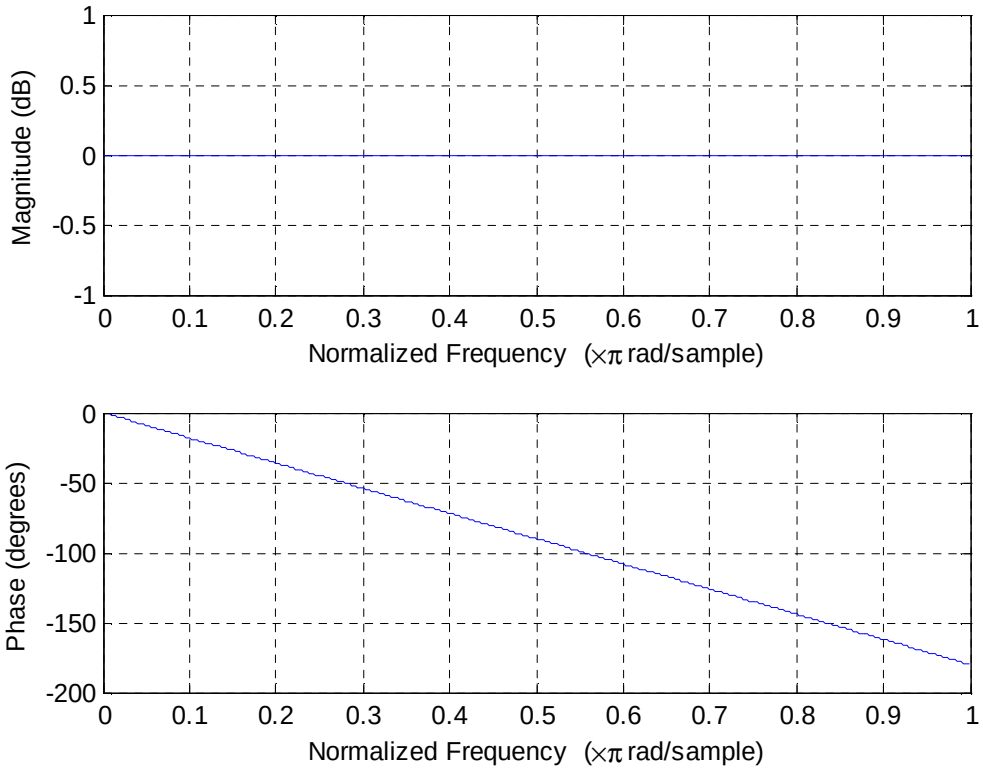


Figure 41 - Frequency response of a single sample delay

Each unit delay adds a group delay of 2 radians or 360 degrees per digital Hz. The effect of each additional delay will add to this group delay, i.e. 2 delays will be 4 rad/Hz. The effect of this on an OFDM symbol can be seen in Figure 42.

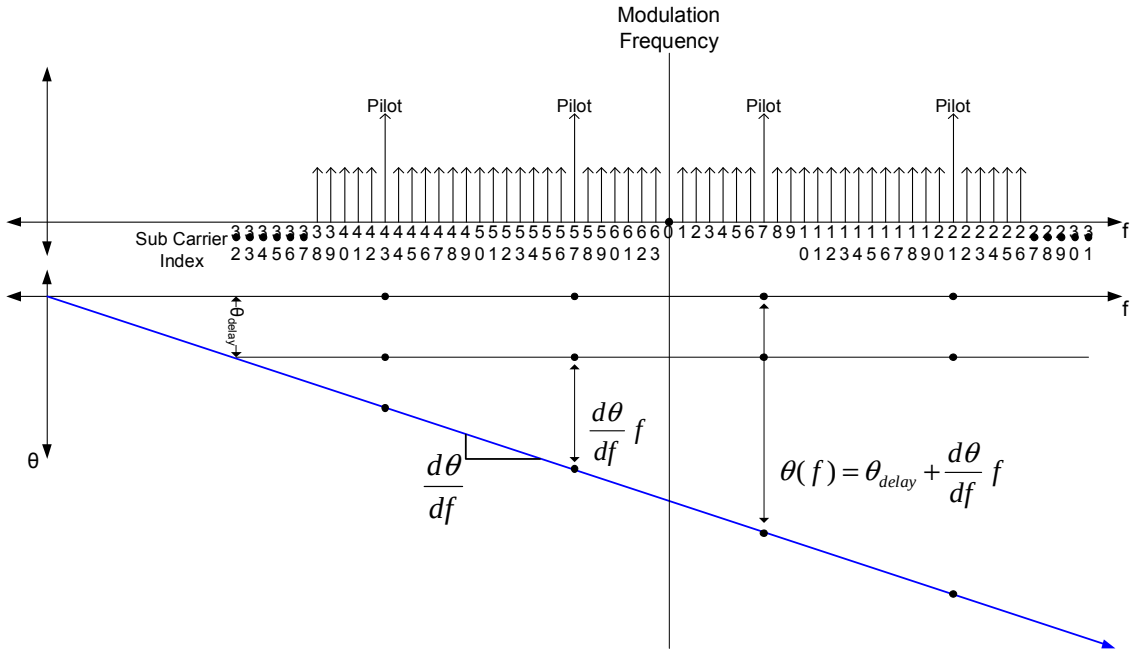


Figure 42 - Strategy for correcting phase corruption of channel and radio

Figure 42 shows the modulated OFDM symbol and a constant unknown group delay. The phase effect on each sub-carrier will be linearly related to the frequency by Equation 14. If θ_{delay} and $\frac{d\theta}{df}$ can be estimated from the pilot signals, then the phase correction for each sub-carrier can be calculated and applied.

Equation 14 - Phase effect caused by a constant group delay

$$\theta(f) = \theta_{delay} + \frac{d\theta}{df} f$$

Implementation

Figure 43 shows the Simulink diagram used to estimate the parameters of Equation 14 and apply them to the QAM symbols.

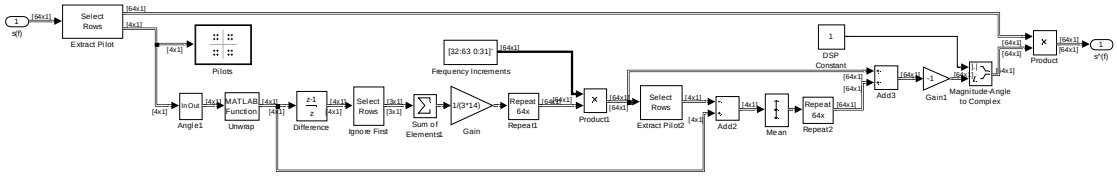


Figure 43 - Simulink diagram for phase correction of QAM symbols

Pilot Angle Extraction

The first step in this diagram is to extract the pilot signals and determine their phase. This part of the diagram is shown in Figure 44.

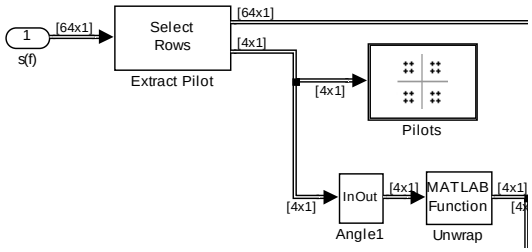


Figure 44 - Pilot angle extraction

The pilot signals are extracted by their index with the extract pilot block. Their angle is then determined with a magnitude and phase extraction block. The magnitude is unimportant and discarded. These angles are then fed into the Matlab unwrap command. This command "unwraps" radian phases by changing absolute jumps greater than π in the phase to their 2π complement. After unwrapping the angles, they should now all lie on a line.

Estimation of Group Delay

The next step is estimate $\frac{d\theta}{df}$ from the unwrapped pilot phases. This is done in the part of Figure 43 shown in Figure 45.

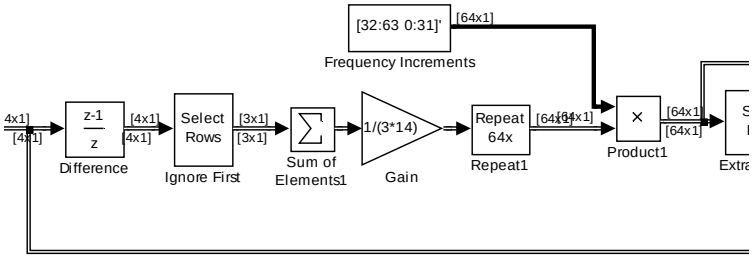


Figure 45 - Estimating the group delay

The first thing that is done is a difference taken with the discrete transfer function $\frac{z-1}{z}$. The first value of this difference is ignored because it is the difference between frames and is meaningless. The 3 values that are left are then averaged and divided by 14. This should estimate $\frac{d\theta}{df}$ when f is measured in sub-carrier spacings. This value is then multiplied by the vector [32:63 0:31] to calculate the proportional part of Equation 14 for each sub-carrier.

Estimation of the constant phase effect

After this the next step is to estimate θ_{delay}. This is done in the part of Figure 43 shown in Figure 46.

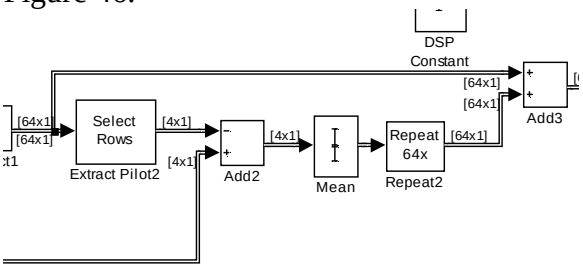


Figure 46 - Estimating the constant phase effect

Equation 15 is used to estimate θ_{delay}.

Equation 15 - Equation used to estimate the constant phase effect

$$\theta_{delay} = \theta(f) - \frac{d\theta}{df} f$$

The proportional part of the phase delay that was just calculated is applied to the unwrapped pilot angles. The 4 values that are left should be approximately the same. These values are then averaged and added to the result of Figure 46.

Application of phase correction

Now that the phase correction is calculated, it needs applied to the QAM symbols so that they can be decoded. This is shown in Figure 47

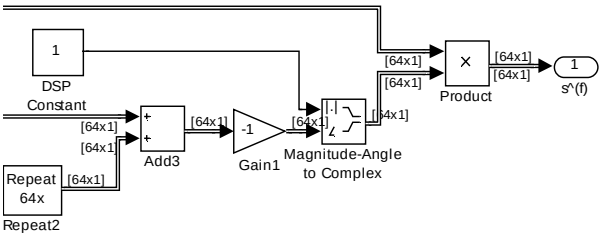


Figure 47 - Application of the phase correction

To do this, the angles are first negated so that they correct for the phase effects and made into unit phasors. This frame of QAM symbols phasors is then multiplied by these phasors to correct for the phase effects of the transmitter and receiver.

xii. QAM Decoding

Finally the compensated symbols are ready for QAM decoding. In this subsystem the four level QAM symbols are decoded back into the bits that they represent that the original transmitted byte is reconstructed.

Theory

As shown previously in QAM encoding, the complex envelope represents each QAM symbol as [1]:

Equation 16 – Complex Envelope of QAM signal

$$g(t) = x(t) + jy(t)$$

To decode the symbols, the QAM constellation is used again. The decoding process is the reverse of the encoding process. Each QAM symbol is mapped to the point representing the transmitted bits that the symbol represents. For example, using the constellation shown in the encoding section, the symbol "1-j" is decoded back into the bit pair "10." The decoded bit sets are then reconstructed to form the original bit stream that was sent from the transmitter.

Implementation

The QAM decoder design is shown in Figure 48 and is the final stage of the receiving process. The data comes into the system serially and needs to be buffered. The specified rows are selected to grab each symbol needed.

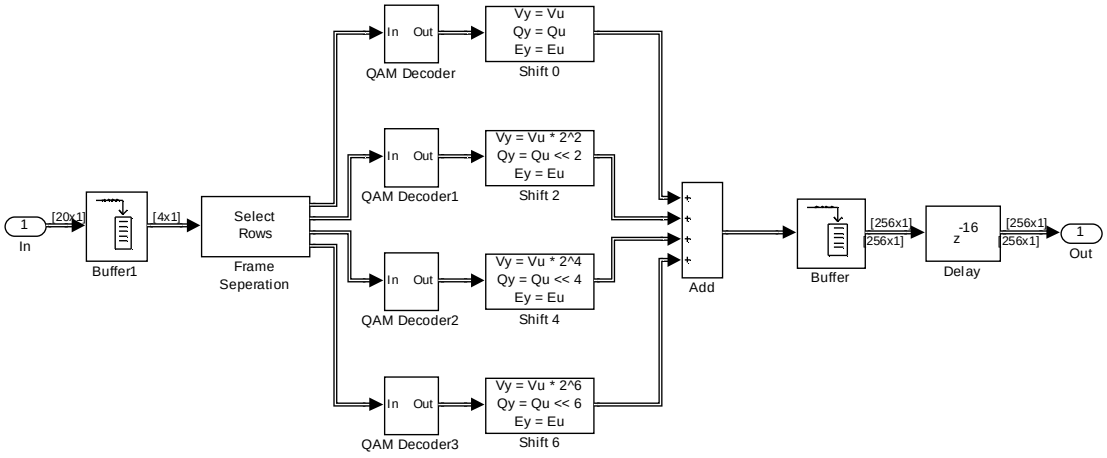


Figure 48 - Simulink Diagram of QAM decoder

The symbol is passed into the QAM Decoder block. The content of this block is shown in Figure 49. First the symbol is broken into its real and complex parts. The compare to "0" is used as the level detection, since each magnitude either above or below 0. If the magnitude is greater than 0 then a "1" is passed from the "compare to zero," otherwise a "0" is passed. The shifting block is used to create a second bit so the two bits

representing the real and complex components can be added together in the designated order with the real part representing the most significant bit. The addition creates the bit pair representing the symbol that was detected.

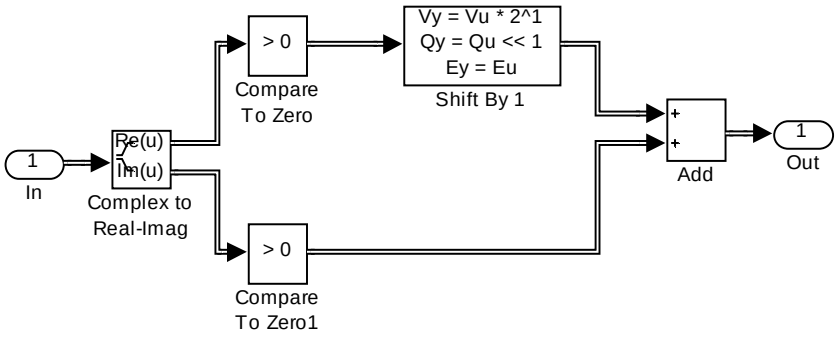


Figure 49 - QAM decoder block for a single QAM symbol

Once each bit pair is passed from the QAM decoder blocks, the bit shifting operations are performed to combine all of the bit pairs into the original transmitted byte of data. The buffer is placed on the output for easy viewing in the Simulink Matrix Viewer, and the delay is placed on the buffer output due to buffer delays in the system. Since this a basic four level system, level detection can be used to detect the amplitude of the signals to map the symbols back to bits represented when the data was sent.

V. Software

In addition to the model of the radio itself, additional software had to be programmed to support the development of the software radio and its implementation on the DSK board

i. Image Viewer

The image viewer blocks provide a way to visually check the functionality of the radio. These blocks will display a 256x256 grayscale image as the image is being transmitted and received through the radio. Figure 50 shows the Simulink diagram that does this.

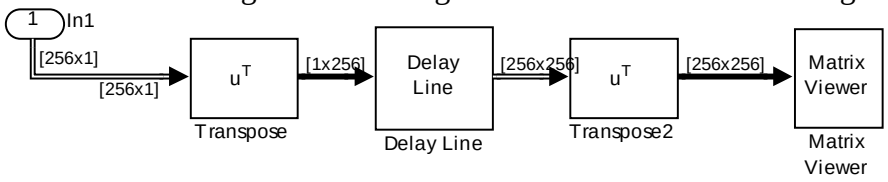


Figure 50 - Simulink diagram for image viewer

The data enters the block as a vertical column of data in the image. It is then transposed so that the delay line is able to buffer entire lines over time. The matrix is then transposed again so that the image retains its proper orientation. It is then displayed with a matrix viewer using a 256 level gray color map. Figure 51 shows the results of this.

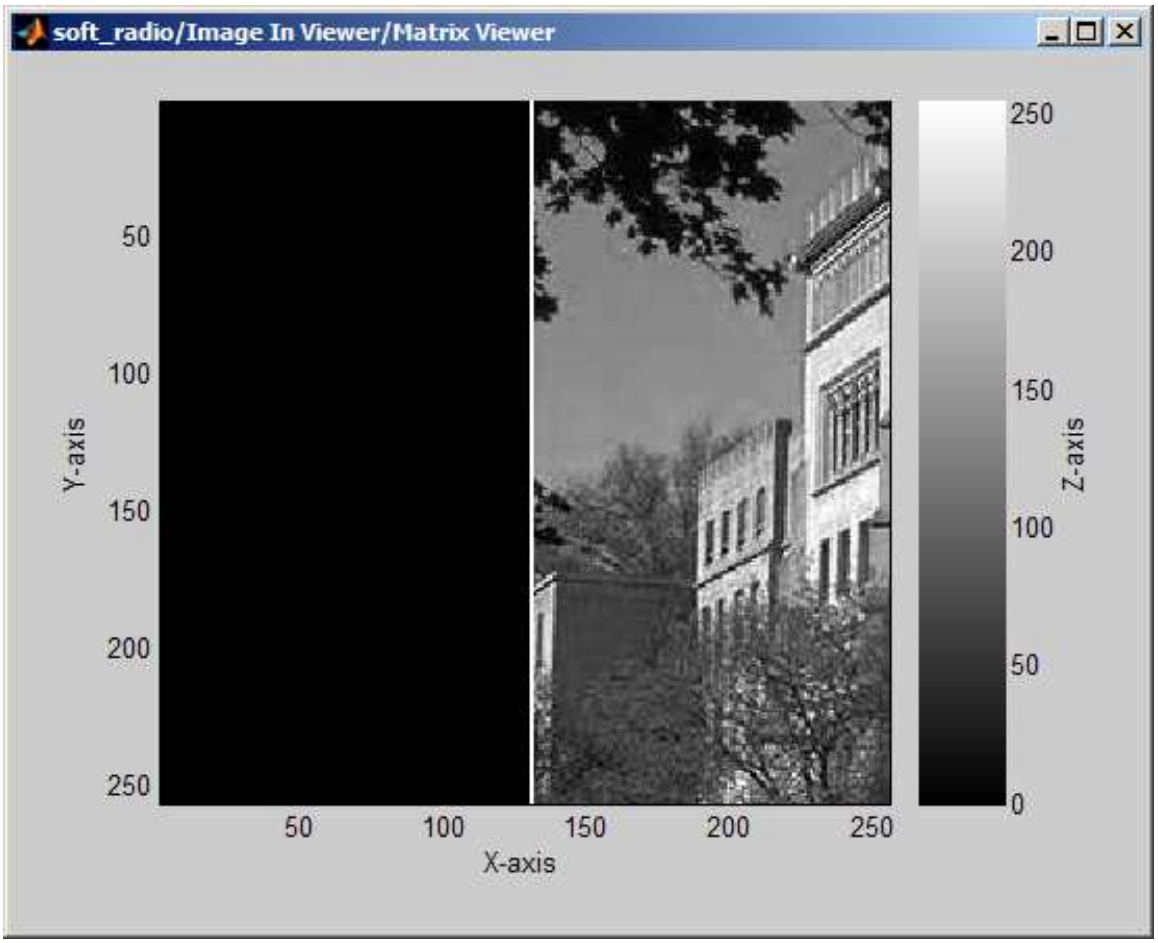


Figure 51 - Results of image viewer diagram

ii. RTDX

The RTDX (Real Time Data Exchange) was used to transmit digital data to and from the Texas Instruments 6713 DSK board. To utilize this tool, an interface was needed to specify what data would be transmitted. A Windows application was programmed using Microsoft Visual Studio 2005 using the C# language. The main form is shown below in Figure 52.

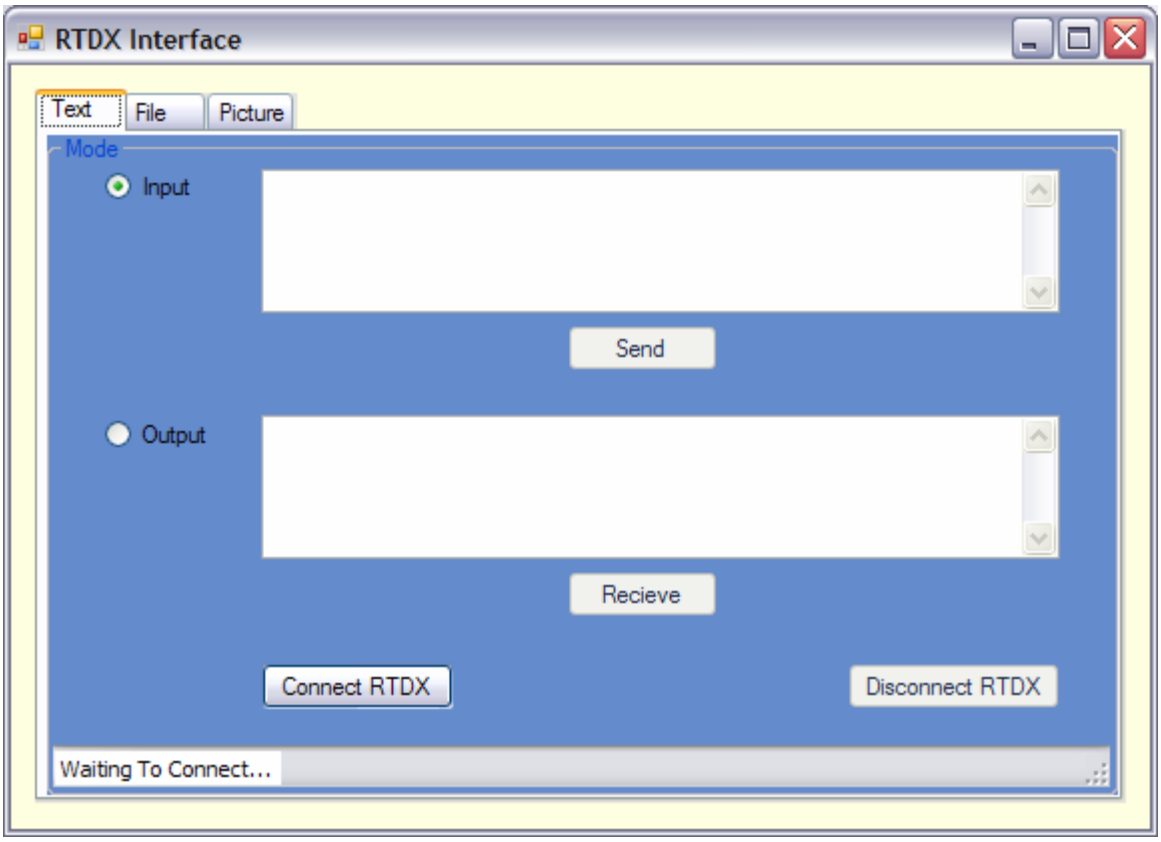


Figure 52 - The main Windows Application Form for the RTDX interface

This interface was designed to provide three different types of digital data transmission and reception. The "Text" transmission and reception was the primary focus for quick testing and implementation.

Steps in Using the RTDX interface

A Simulink diagram or C file must be completed that initializes the RTDX link on the DSP board. A block diagram was developed using Simulink that targeted the 6713 DSK board initializing its RTDX link.

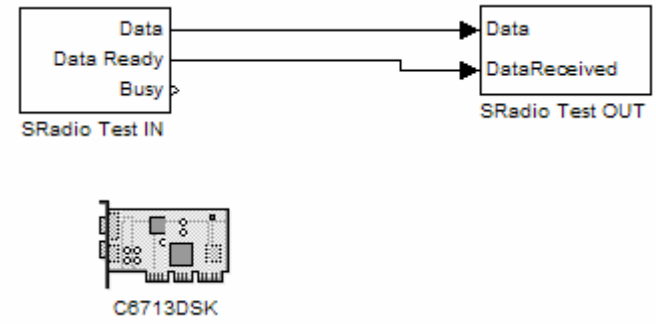


Figure 53 - Simulink block diagram of the RTDX interface

Compiling this model using Simulink, generated a “test.out” file which can be downloaded to the DSP board using Code Composer Studio. After the program is downloaded to the board, it needs to be run. Once the program is running on the board, the Windows application interface can be executed to create a connection with the board via RTDX. When the Windows application loads, press the “connect RTDX” button on the screen. A channel is created that reads from and writes to the RTDX link. In the input text box, type any text that is to be transmitted, and then press the send button.

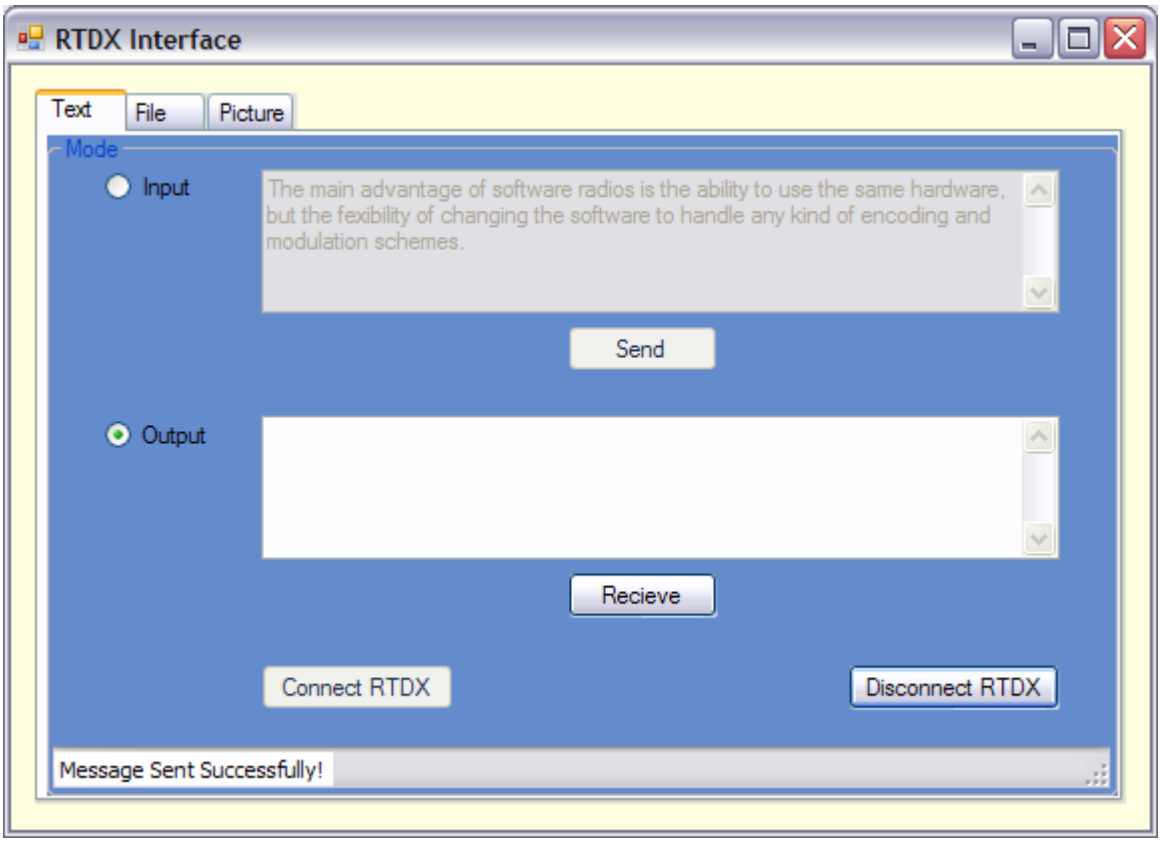


Figure 54 - Sending a text message with RTDX

In the status bar, a message should appear that says “Message Sent Successfully!” Press the “Receive” button to receive the message back from the board. The message may take some time to be redisplayed in the output text box.

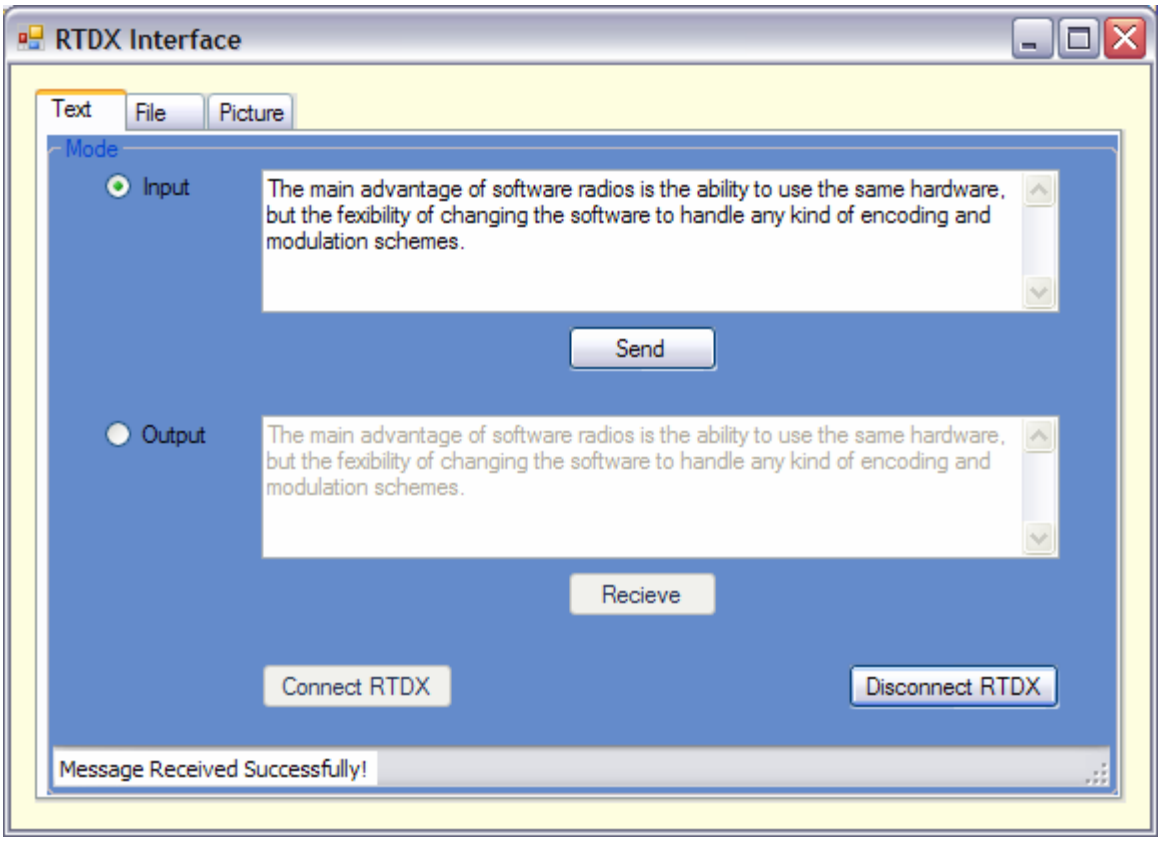


Figure 55 - Receiving a text message with RTDX

After receiving the message you can disconnect from the RTDX link or continue sending text messages.

iii. DSP Testing Interface

The figure below shows a block diagram of the testing software:

Figure 56 - Block diagram for testing interface

Below is shows a flowchart for the flow for the testing software.

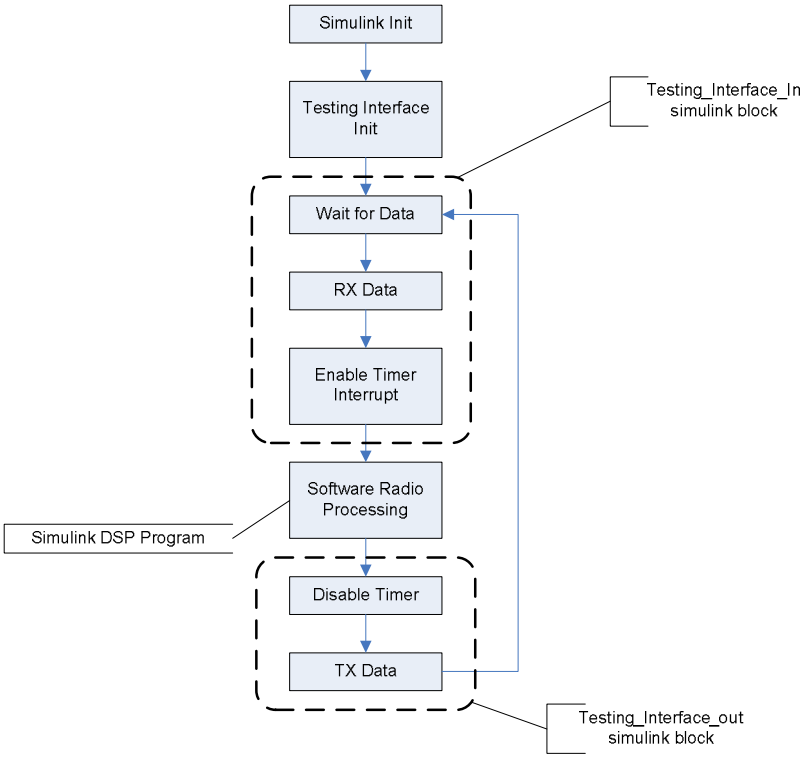


Figure 57 - DSP Board Testing Flow

The testing software for the DSP board needs 3 parts, initialization, data receiving, and data sending. The DSP board timer controls the execution of the Software Radio processing. The timer is disabled when waiting to receive, receiving, and sending data because there is no software radio data processing that needs to be done at these locations. Also, the Simulink program is set to abort on an "overrun." Although this is supposed to be able to be disabled from the "Configuration Parameters" dialog in Simulink:

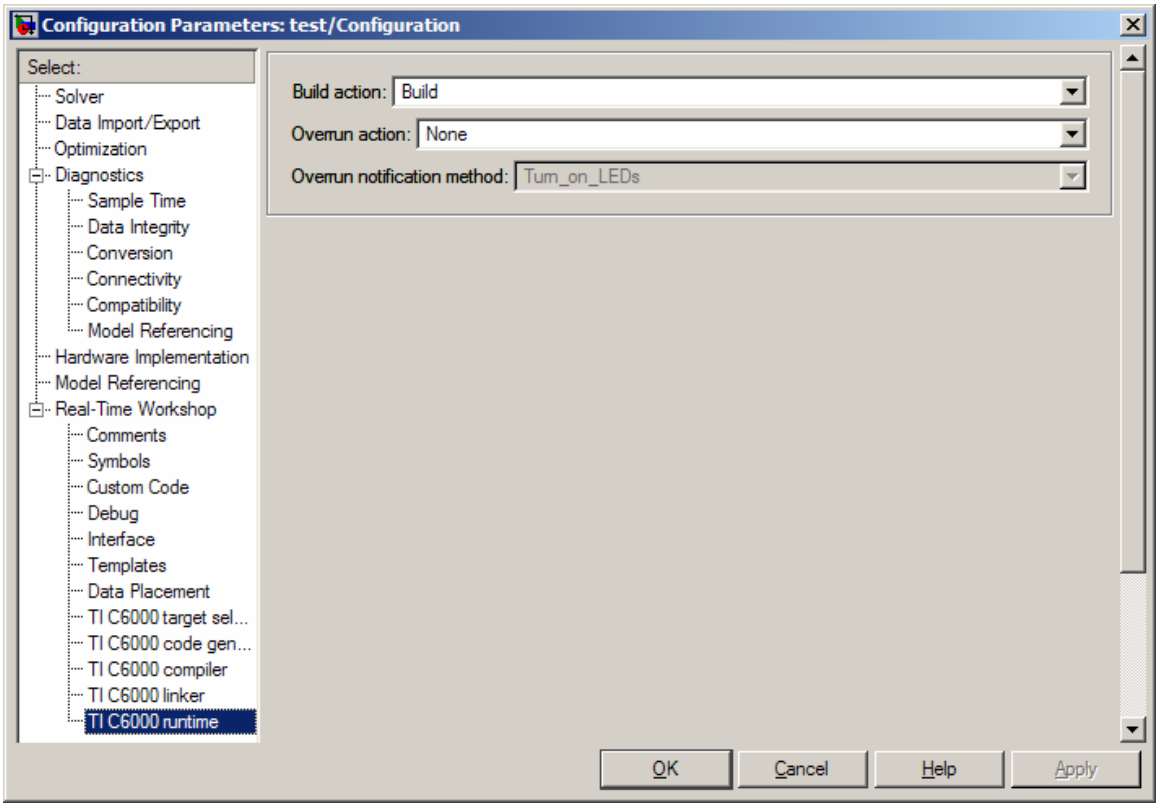


Figure 58 - Configuration parameters for overrun actions

Changing the overrun action to none still makes the board halt on overruns. This seems to be a bug in the Simulink real-time workshop target for C6700 platforms. By pausing the timer while waiting for RTDX communication, the timer interrupt service handler does not fire and the board does not overrun. It is almost like freezing the execution of the Software radio Simulink program until the data is properly received and sent.

Testing Interface Block Flow

The next thing that needs designed is the flow for the Testing_Interface_In block and Testing_Interface_Out block.

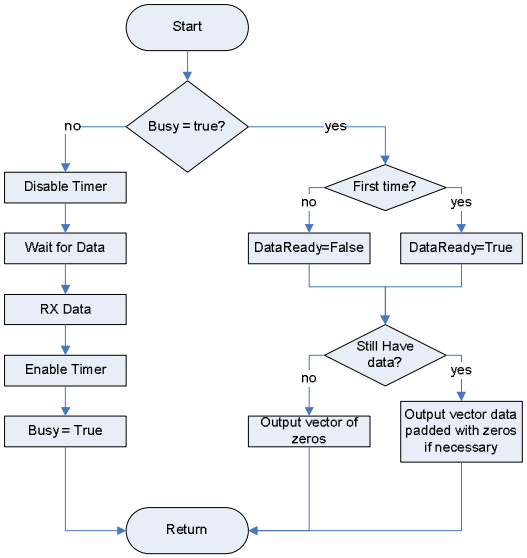


Figure 59 - Testing_Interface_In Simulink block flow for DSP board

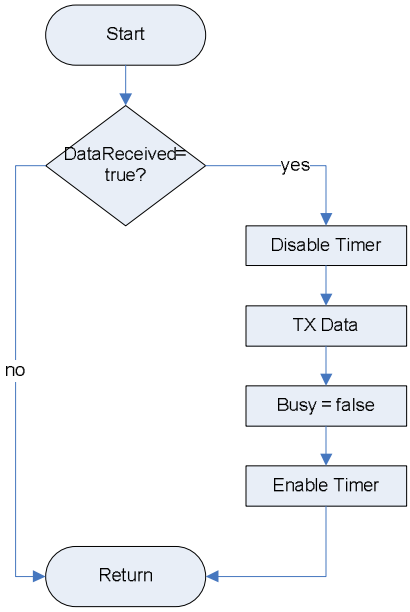


Figure 60 - Testing_Interface_out Simulink block flow for DSP board

S-Function Design in Simulink

In order to make a custom C S-Function Simulink block, there were 3 files that need to be developed

- **Simulink C File** – This file contains the code to set up the Simulink block. It sets the port widths, sample time, parameters, and contains the call back functions that get executed when outputs are needed.
- **TLC File** – This file shows Matlab's real-time workshop how to generate the code for the DSP board itself. It is written in a language called TLC which stands for Target Language Compiler. This is a template file that shows Real-time

Workshop where to insert the appropriate parameters and function calls in the generated code for the DSP board

- **Algorithm file** - This is a file written in C that contains the actual work that the block is going to do. This can be a special processing procedure that needs to be done, communication with hardware, or other things. The code contained in it, unless specified, will get executed when the program is run in Simulink, and when run on the DSP board. Code can be specified to compile on either Simulink or for the DSP board with the C preprocessor instruction: #ifdef MATLAB_MEX_FILE. The code will be compiled only for Simulink and not the DSP board if surrounded with this preprocessor instruction and a #endif instruction. This will be used extensively in the creation of the DSP Testing interface.

The comments in the code explain its operation. The C files in appendix 2 could then be compiled into an S-Block for use with Simulink by the Matlab command: "mex testing_interface_in.c test_interface.c" and "mex testing_interface_out.c test_interface.c". This uses a C compiler to create a .dll file (dynamic link library) for use with Simulink to implement the blocks functionality in Simulink. When appropriately masked. The blocks look like Figure 61.

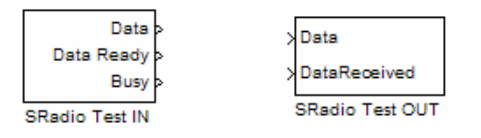


Figure 61 - Simulink Testing Interface Blocks

DSP Code Generation

To make the DSP code generate appropriately, the TLC files needed develop. These essentially place C code that does the operation of the blocks in the appropriate location. Appendix 2 shows the code for these TLC files.

VI. Simulation Results

At this point the radio is functioning and the working of the channel is examined at various channel, noise, and frequency offset conditions. The constellation and spectrum results are shown. The units for the parameters for each simulation are given below:

	Value
pnoise	Amplitude
freq_off	Hz
delay1	Samples
atten1	Scale Factor
delayinc2	Samples past delay1
atten2	Scale Factor
delayinc3	Samples past delay1
atten3	Scale Factor

i. Working Radio

Perfect Channel

Initially no effects were applied to the radio:

	Value
pnoise	0
freq_off	0
delay1	0
atten1	0
delayinc2	0
atten2	0
delayinc3	0
atten3	0

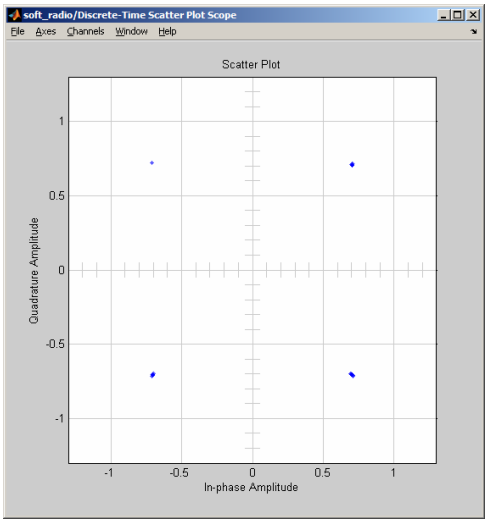


Figure 62 - Synchronized Constellation

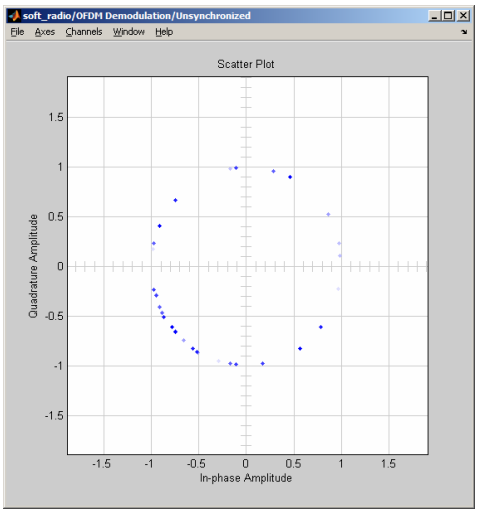


Figure 63 - Unsynchronized Constellation

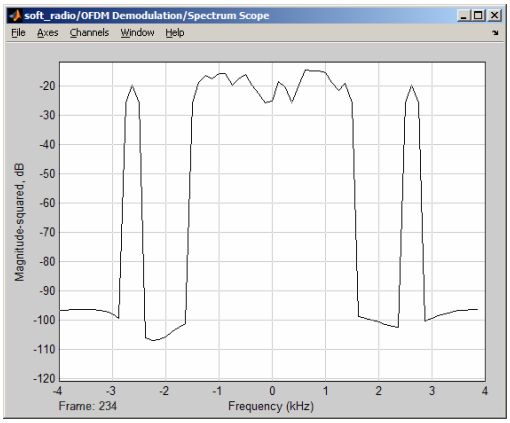


Figure 64 - Receiver Spectrum

Noisy Channel

Next somewhat “typical” values were added to the channel, with extra noise:

	Value
pnoise	0.001
freq_off	0
delay1	53
atten1	0.18
delayinc2	2
atten2	0.1
delayinc3	2
atten3	0.12

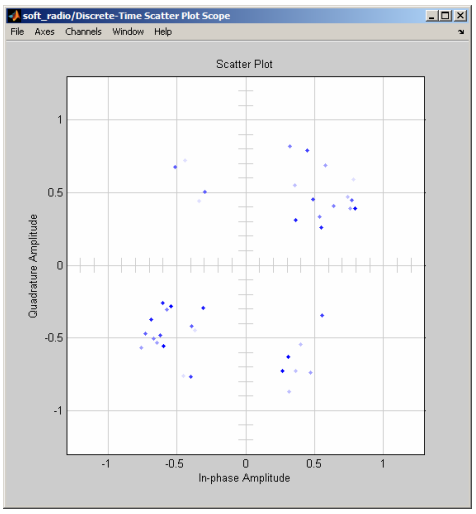


Figure 65 - Synchronized Constellation

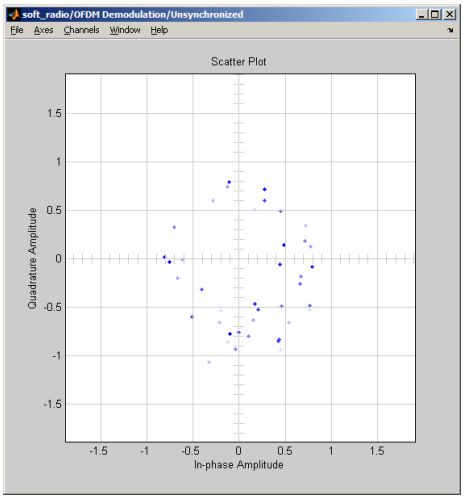


Figure 66 - Unsynchronized Constellation

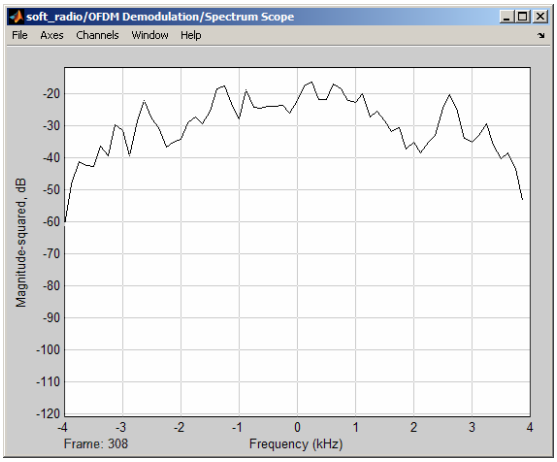


Figure 67 - Receiver Spectrum

25 Hz Frequency Offset

Next the effects of a just a frequency offset were examined.

	Value
pnoise	0
freq_off	25
delay1	0
atten1	0
delayinc2	0
atten2	0
delayinc3	0
atten3	0

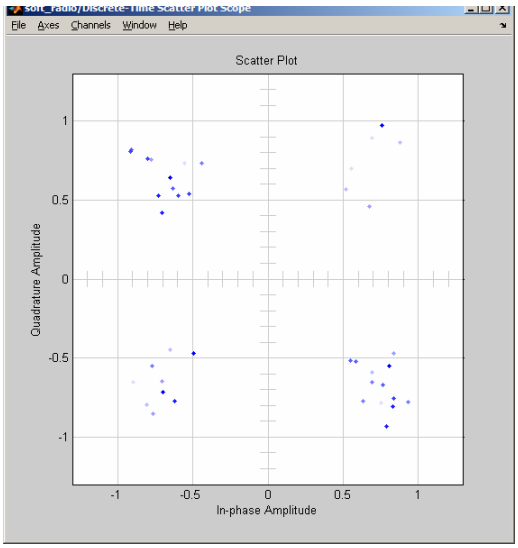


Figure 68 - Synchronized QAM Constellation

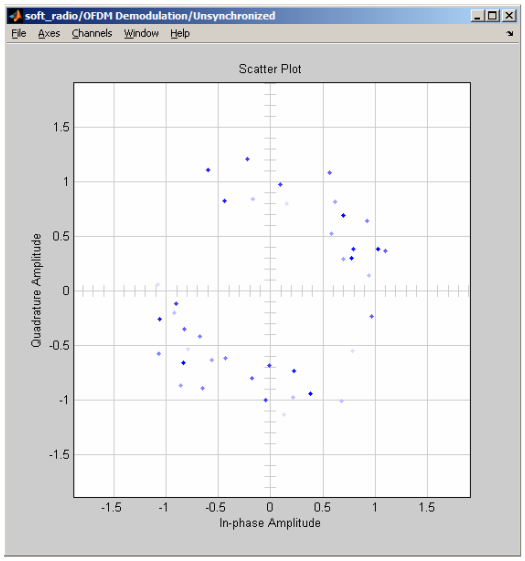


Figure 69 - Unsynchronized Constellation

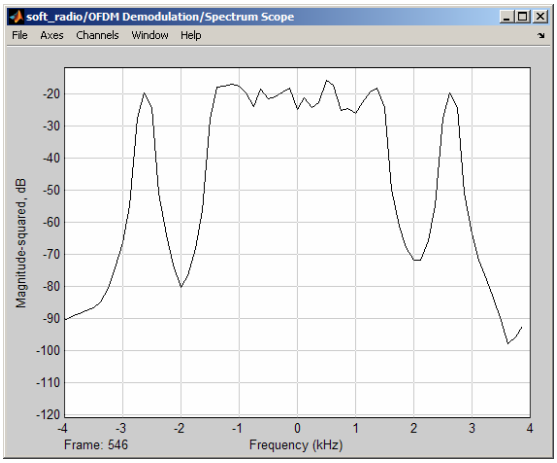


Figure 70 - Receiver Spectrum

No Noise, Typical Channel

Next just the effects of "typical" multi-path interference were examined:

	Value
pnoise	0
freq_off	0
delay1	53
atten1	0.18
delayinc2	2
atten2	0.1
delayinc3	2
atten3	0.12

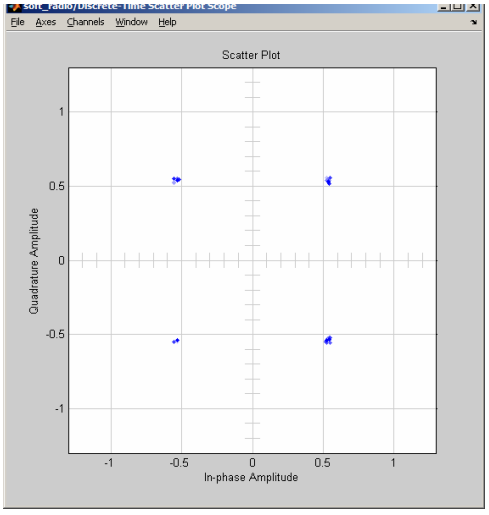


Figure 71 - Synchronized Constellation

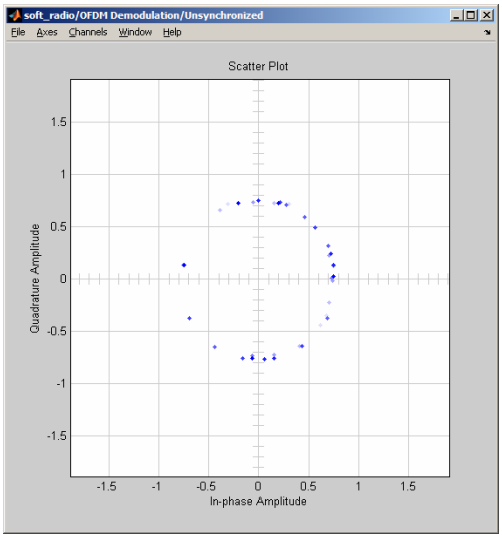


Figure 72 - Unsynchronized Constellation

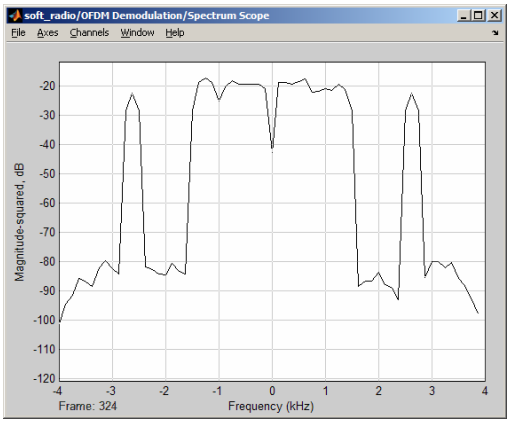


Figure 73 - Receiver Spectrum

Weak Fast Echo

For this test the following parameters were used for a weak fast echo:

	Value
pnoise	0
freq_off	0
delay1	100
atten1	1
delayinc2	5
atten2	0.1
delayinc3	1
atten3	0

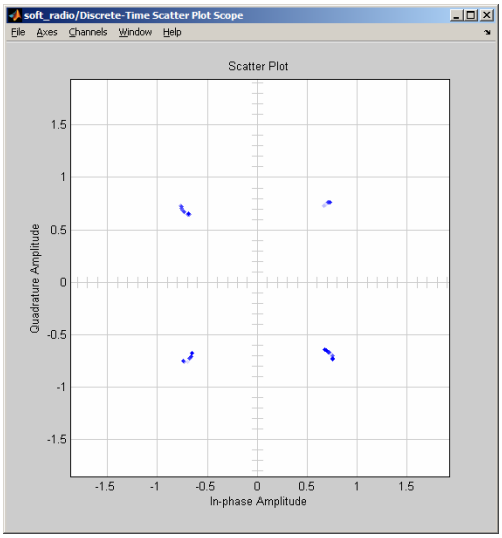


Figure 74 - Synchronized Constellation

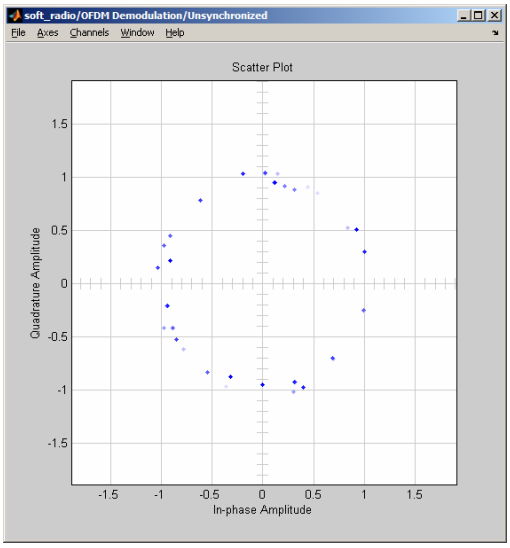


Figure 75 - Unsynchronized Constellation

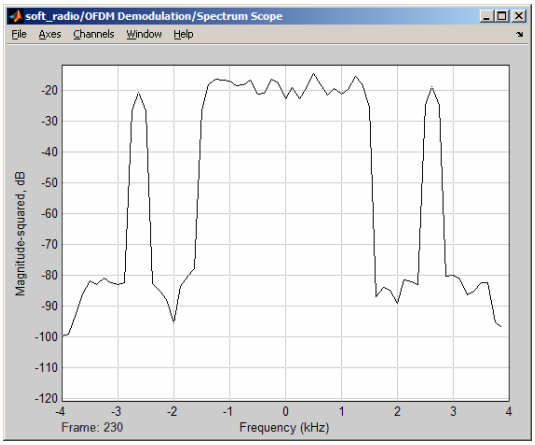


Figure 76 - Receiver Spectrum

Strong Echo

Next the effects of a single, slow, strong echo were examined:

	Value
pnoise	0
freq_off	0
delay1	100
atten1	1
delayinc2	10
atten2	0.5
delayinc3	0
atten3	0

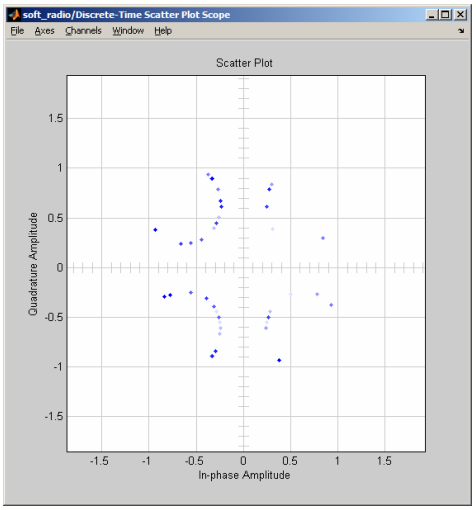


Figure 77 - Synchronized Constellation

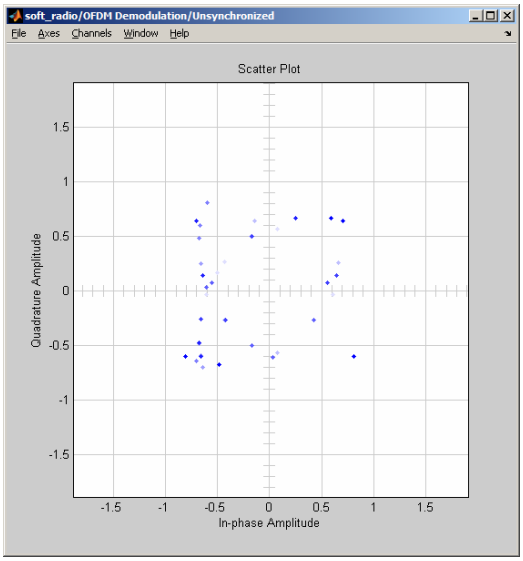


Figure 78 - Unsynchronized Constellation

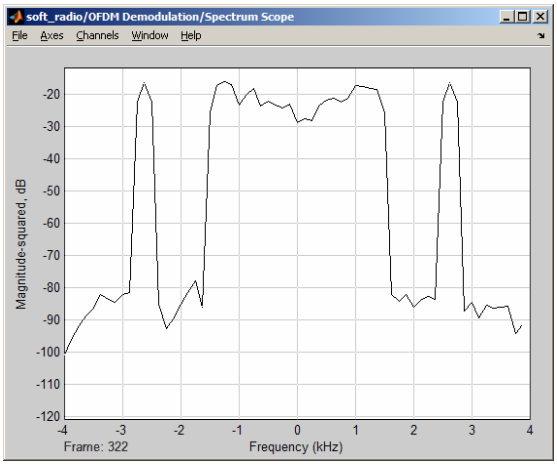


Figure 79 - Receiver Spectrum

ii. Non working radio

Too much noise

In this test, a large amount of noise was added to the system:

	Value
pnoise	0.01
freq_off	0
delay1	100
atten1	1
delayinc2	0
atten2	0
delayinc3	0
atten3	0

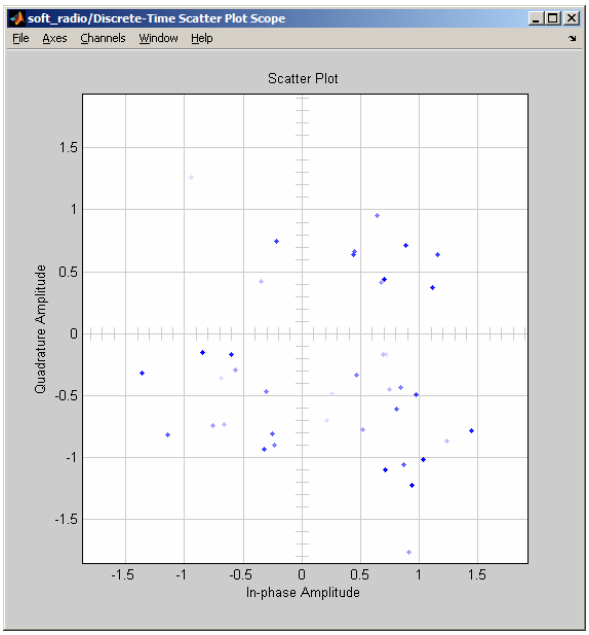


Figure 80 - Synchronized Constellation

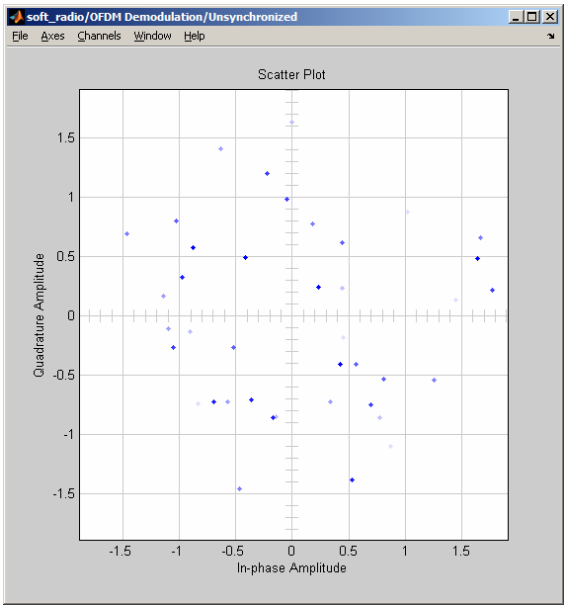


Figure 81 - Unsynchronized Constellation

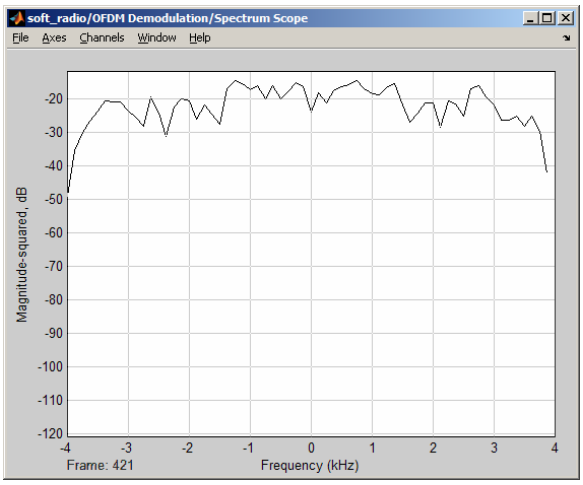


Figure 82 - Receiver Spectrum

Too much frequency Offset

In this test an extreme amount of frequency offset was applied:

	Value
pnoise	0
freq_off	50
delay1	100
atten1	1
delayinc2	0
atten2	0
delayinc3	0
atten3	0

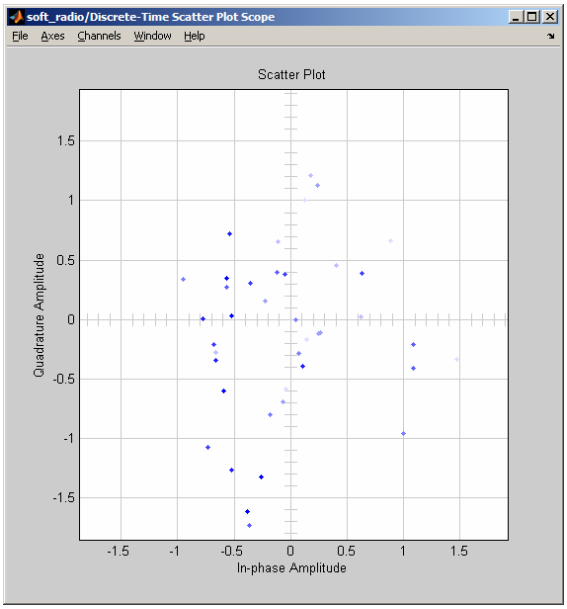


Figure 83 - Synchronized Constellation

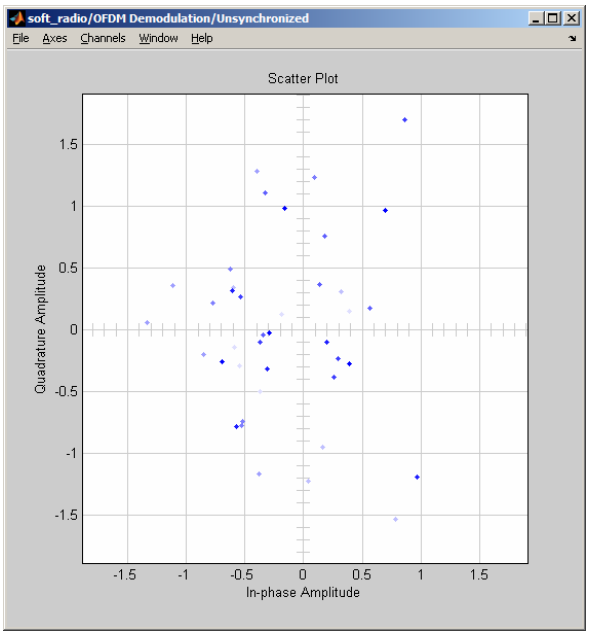


Figure 84 - Unsynchronized Constellation

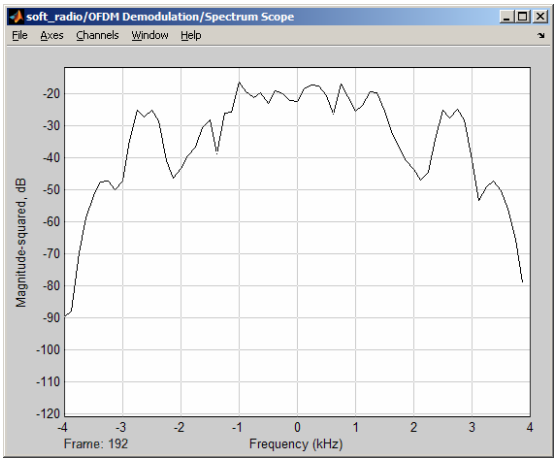


Figure 85 - Receiver Spectrum

Too much Echo

In this test the echo time was increased until the radio constellation became unstable

	Value
pnoise	0
freq_off	0
delay1	100
atten1	1
delayinc2	5000
atten2	0.3
delayinc3	0
atten3	0

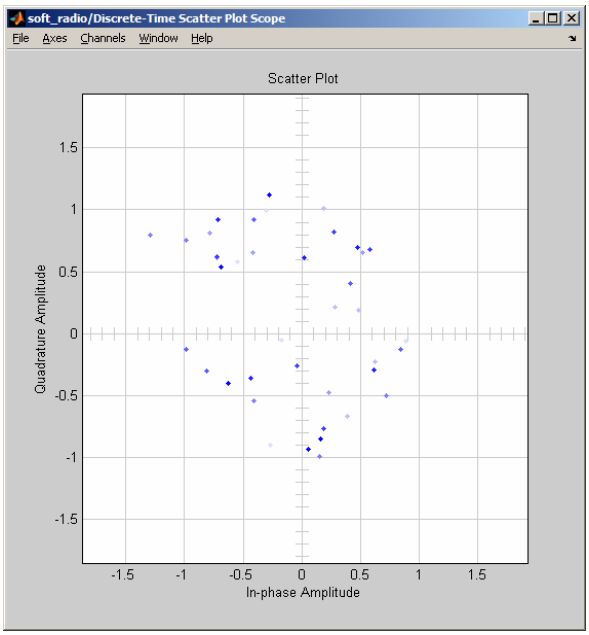


Figure 86 - Synchronized Constellation

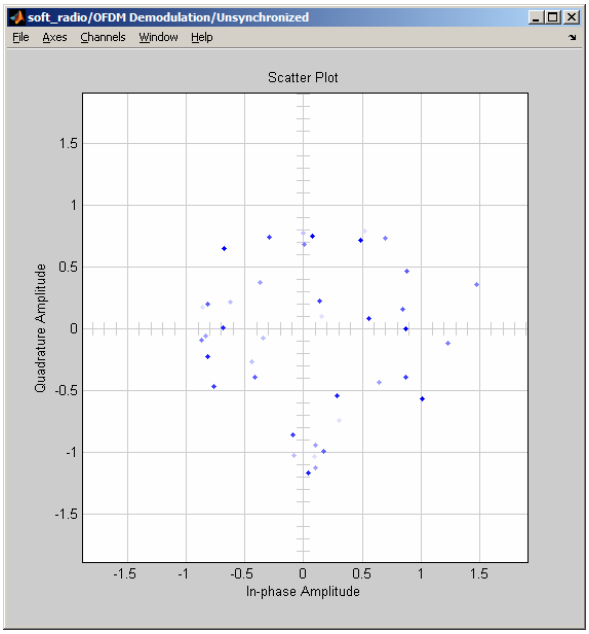


Figure 87 - Unsynchronized Constellation

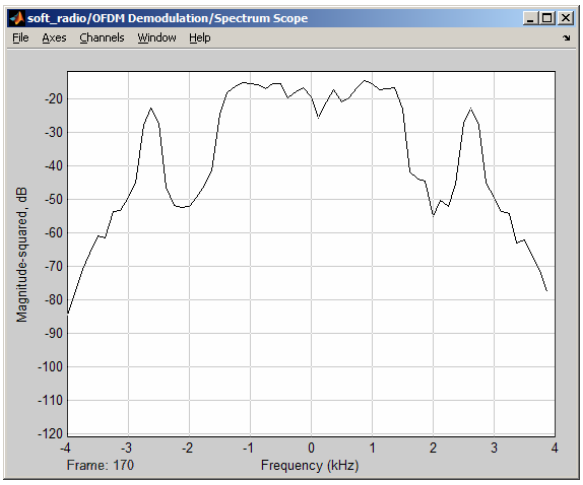


Figure 88 - Receiver Spectrum

VII. Analysis

The original goal of this project was to get a working software radio on a DSK board and analyze its performance. However, due to unforeseen circumstances this did not happen. However the radio did work well in simulation and the performance of the design could be analyzed in simulation.

i. DSP Board Problems

When the radio was completed, code was generated for it and it was downloaded to be run on the DSP board. However, it was found that the DSP boards would not support the required data rate needed for the radio. When the code was run, the board would "overrun." This means that another timer interrupt is received by the DSP before the first is finished processing. This means the following things were examined to try and improve the radio:

Reduce the sampling rate

The sampling rate for the output was set at 96 KHz. This is to provide the most bandwidth for the signal to be outputted on. 96 KHz is 8 times the IFFT sampling rate. A minimum of 4 times the sampling rate is needed to keep the OFDM spectrum away from DC. However, the transmitter, itself, did not run at 96 KHz. Since the receiver is more than twice as complex as the transmitter, the chances that it would run, even at 48 KHz is slim.

Optimize block performance

Optimizing block performance is essentially guess work. Since there is no way available in the version of Matlab that we had to profile the model and see where the model is slow, it is very hard to figure out where to optimize the model to get it to run faster. The C code itself could be profiled and traced back to the model. However, since it did not run on the board in the first place, this cannot be used to find the bottlenecks in the model. What would have to be done in this case is optimize each specific block as much as possible and by trial an error find the slow blocks.

Fixed Point vs. Floating Point

Currently, the algorithms used are designed with floating point numbers in mind. If these were converted to a fixed point implementation, there could be some improved speed. However, it has not been researched thoroughly. A brief examination of the DSK instruction set indicates that the 6713 has floating point operations built in. It seems to process these in 1 to 2 instruction cycles which indicates that it would execute floating point instructions as fast and fixed point. This does not look like a very promising solution.

Faster Hardware

Faster hardware would run the radio model. However, faster hardware is significantly more costly. Also, the DSK boards that we are using are one of the faster ones currently available. This is not a very cost effective solution and is kind of a last resort.

ii. Radio Error Rate Analysis

There are some error rate specifications given in the 802.11a specification. However, these error rates depend on the type of coding used so they are not very applicable to our radio which does not use coding. Since our radio uses many QPSK channels, it seems reasonable to compare it to the theoretical QPSK values for errors. According to *Digital and Analog Communication Systems* by Leon Couch, pg. 501, the theoretical error rate for a QPSK system is given by $p_e = Q\left[\sqrt{2\frac{E_b}{N_0}}\right]$ where $\frac{E_b}{N_0}$ is the bit power to noise power ratio and Q is the Gaussian cumulative distribution function. The average signal power of the transmitted OFDM signal will be greater than the bit energy because there are pilot signals inserted into the OFDM symbol for synchronization purposes. The data gathered from the radio tabulates symbol error. If there was only on bit error per symbol, the BER will equal the symbol error rate. If there are always two bit errors per symbol, the BER will be twice the symbol error rate. Figure 89 shows a comparison between the theoretical QPSK error rate and the experimental BER.

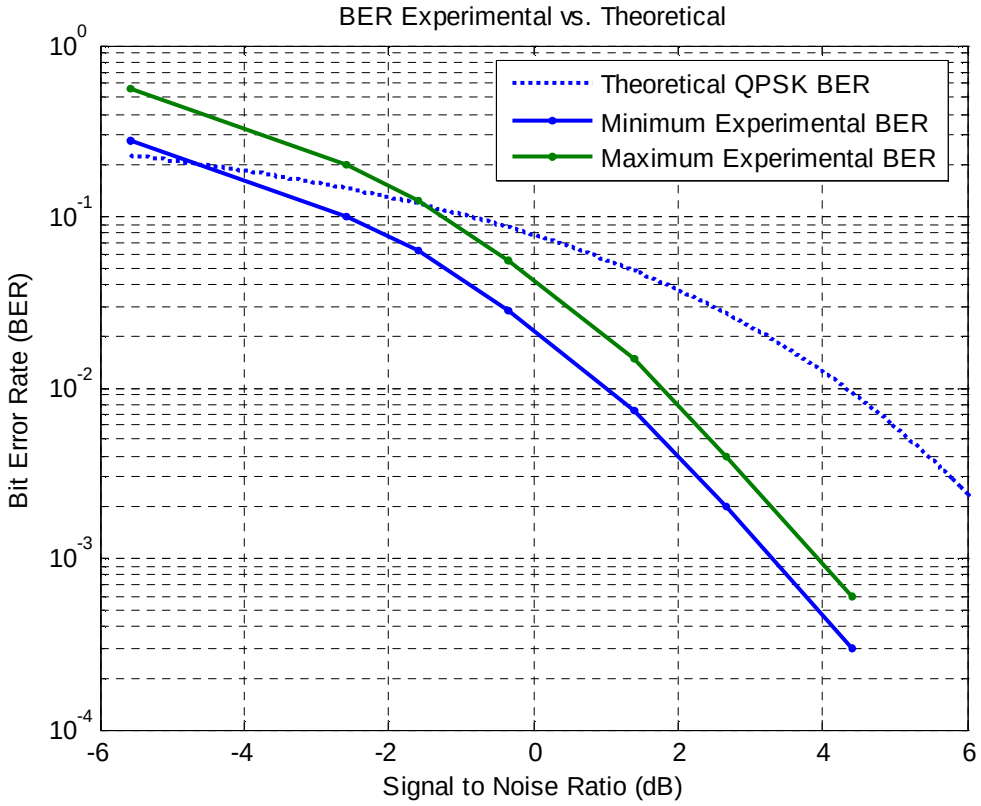


Figure 89 - BER experimental vs. theoretical

As can be seen from Figure 89, the experimental BER exceeds the theoretical BER for SNR above -1.5 dB. After that, the theoretical BER lies within the experimental BER. At high SNR values, the BER exceeds 0.75. A BER of 0.75 is pure guessing as there is a 0.25 probability of getting the right points from guessing. This is probably due to the

non-randomness of the test data increasing the number of errors due to the white indicator stripe transmitted at the beginning of the image.

iii. Synchronization Performance Analysis

Experimental data was also collected to determine how well the synchronization worked. Figure 90 compares when the synchronization looses lock at different noise powers.

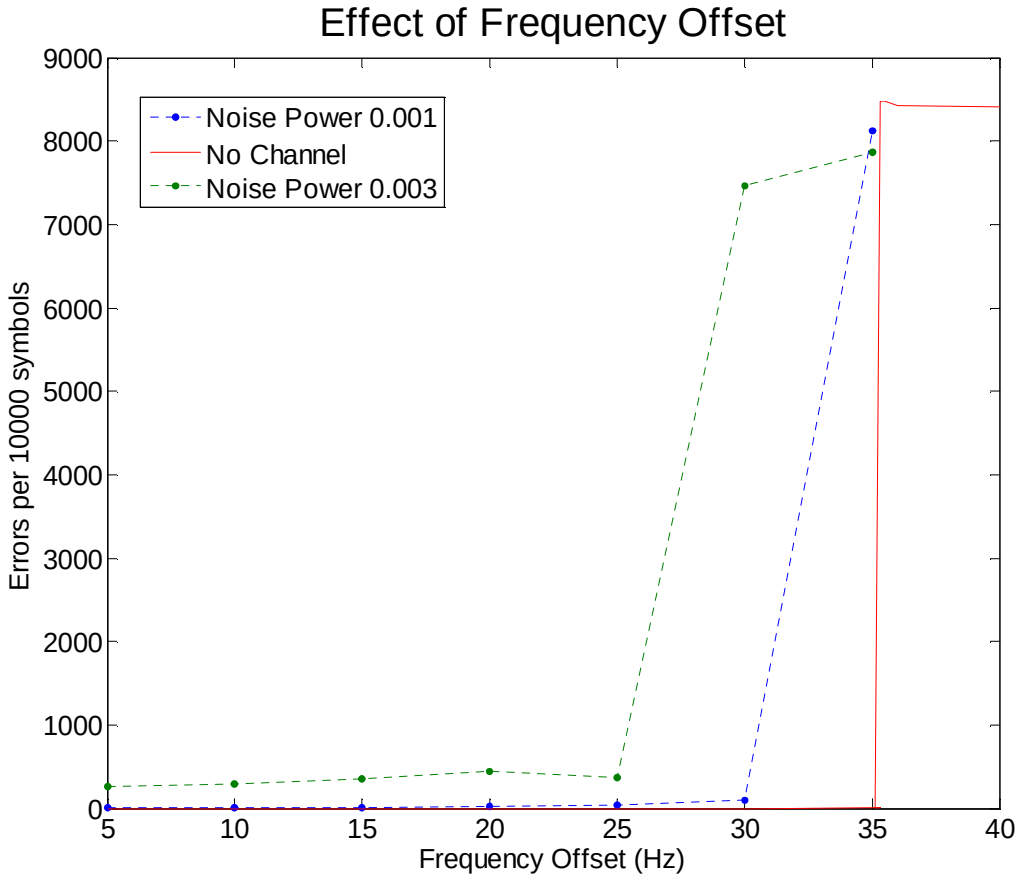


Figure 90 - Synchronization Performance Graph

From this graph we can see two trends. The first is that additive white Gaussian noise causes the synchronization to loose lock faster than without the noise. Secondly, it can be seen that a frequency offset increases the error rate over no frequency offset. This is due to the fact that there will be a constellation spread that increases as the frequency offset increases. If individual channel estimation was used instead of averaging the effects of the pilot signals, then this would most likely improve.

VIII. Conclusions

A software radio system was designed and implemented in Simulink. Potential channel problems of phase offset, frequency offset, multi-path interference, and white Gaussian noise were examined and compensated for. The effects of these disturbances were then analyzed. The radio design was able to be converted into C code and downloaded to the DSP boards. However, the code was too slow to run on the DSP board. Additional optimization needed to be done on the model, but was hampered by the inability to profile the model and determine the bottlenecks. An efficient and reliable way to exchange data with the DSP board was also developed and would be very useful in future senior projects. If this project would be redone, the following things would be improved upon:

- More attention would be paid to the phase effects of the different parts of the radio. Figuring out the problem with the phase of the filters wasted a significant amount of time early in the radio's development.
- The radio would be worked on before more of the hardware was so that a working radio model could be used to experiment with the hardware rather than test programs. This would also have allowed us additional time to see what could be improved upon and optimized on with the radio instead of working with the hardware itself.
- The complex parts of the radio (i.e. The OFDM symbol synchronization and the carrier synchronization) need to be examined earlier. This would allow more time for the design iterations for these parts to be worked out.
- A sampling rate needs to be settled upon earlier so that the radio does not have to be redesigned for different sampling rates multiple times.
- The goals for this projected needed to be scaled down and made more reasonable so that the significant parts of the project could be focused on and completed more fully.

Future expansions for this project could include:

- Figuring out the Simulink diagram optimization so that the radio speed could be increased to run on the board.
- Implement channel coding on the radio to reduce the error rate due to the additive white Gaussian noise.
- Looking into implementing actual RF hardware to simulate the radio's performance under real world conditions.
- Investigate multilevel QAM and automatic gain control in order to increase the radio's data rate. The rest of the carriers on the OFDM frame could also be easily utilized to increase this.

IX. Equipment List

The equipment in table 1 was used to complete this project.
2 DSP Starter Kits (Texas Instruments TMDSDSK6713)
2 Computers
Matlab 7.0.4 with Simulink 6.2
Code Composer Studio 3.1
Microsoft Visual Studio 2005

Table 1 - Equipment List

X. Datasheet

Parameter	Symbol	Limits			Units
		Min	Typical	Max	
Modulation Frequency	Fmod		24K Hz		Hz
Data Rate	R		5Kbps		bits / sec
Sampling Rate	Fsamp		96K Hz		Hz

Table 2 - Specifications for Software Radio

XI. Patents and Standards

Standards and Patents Research

The purpose of the standards and patent research was to gather information and ideas that would help in the design and implementation of the project.

Standards

The primary standard that this project is based on is IEEE 802.11a. IEEE 802.11a is a standard for wireless networking. In the standard there are details for the implementation and operation of a digital radio for use as a network interface. Our project is an adapted version of this standard, scaled to the scope of the project and the time available for the project.

Patents

The following patents and patent applications were discovered when researching patents and patent applications applicable to this project.

Patent Number	Description
6,091,765	Reconfigurable radio system architecture
6,353,640	Reconfigurable radio frequency communication system
6,937,877	Wireless communication with a mobile asset employing dynamic configuration of a software defined radio
6,954,628	Radio receiver
Patent Application	
2005243952	Methods for processing a received signal in a software defined radio (SDR) system, a transceiver for an SDR system and a receiver for an SDR system
20050190827	Modulation/demodulation apparatus for the encoding and decoding of data and method for encoding and decoding data

Table 3 - Applicable patents and patent applications

XII. Bibliography

[1] Couch, Leon W. Digital and Analog Communication Systems. New Jersey: Prentice Hall, 2002.

[2] IEEE Std. 802.11a. "Part 11: Wireless LAN Media Access Control and Physical Layer Specifications." IEEE SA-Standards Board. June 12, 2003. "http://pdos.csail.mit.edu/decouto/papers/802.11a.pdf". Accessed October 21, 2006.

[3] Johnson, C. Richard, Jr., Sethares, William A. Telecommunication Breakdown. 2004. New Jersey:Prentice Hall, 2004.

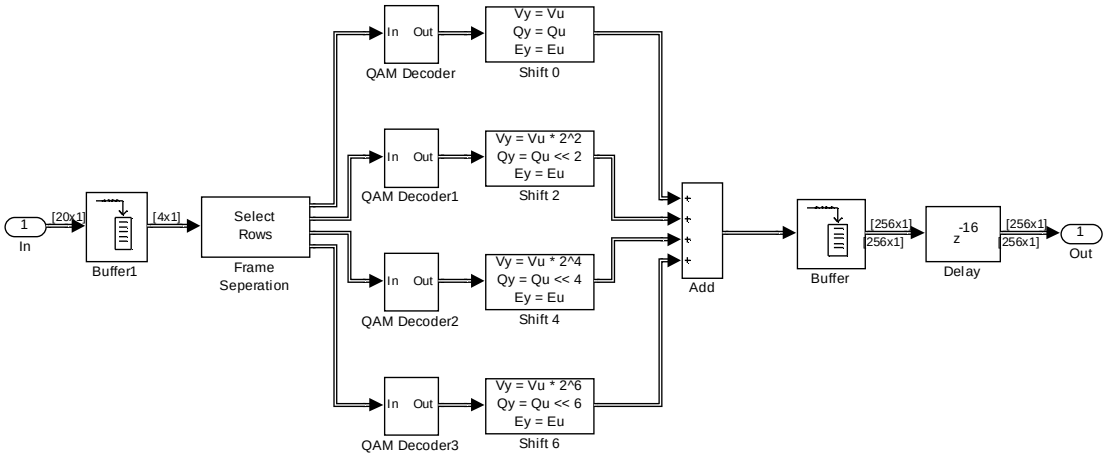
[4] Prasad, Ramjee. OFDM for Wireless Communication Systems. 2004. Artech House.

[5] Proakis, John G. Digital Signal Processing: Principles, Algorithms, and Applications, 3rd Edition. New Jersey : Prentice-Hall, 1996.

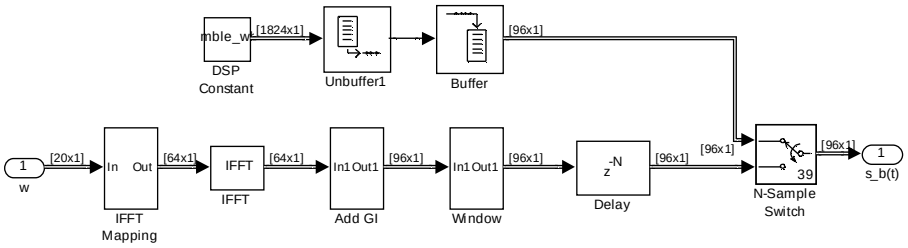
[6] Sridhar Nandula, K Giridhar. "Robust Timing Synchronization for OFDM Based Wireless System." "www.ewh.ieee.org/ecc/r10/Tencon2003/Articles/659.pdf" Accessed March 29, 2006.

Appendix 1 – Simulink Block Diagrams

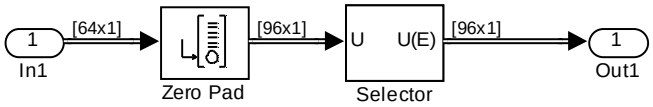
i. QAM Symbol Encoder



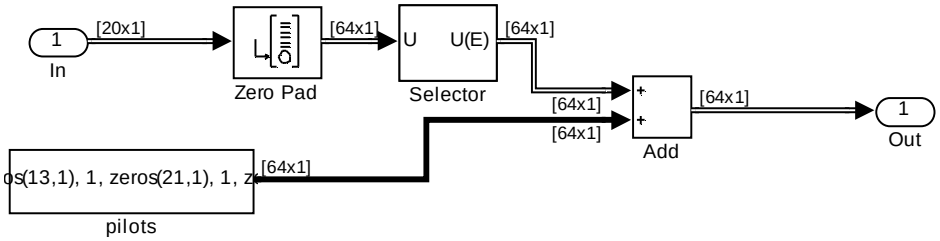
ii. OFDM Modulation



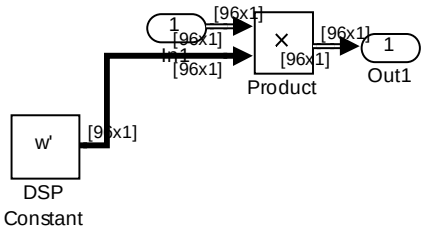
Add Guard Interval



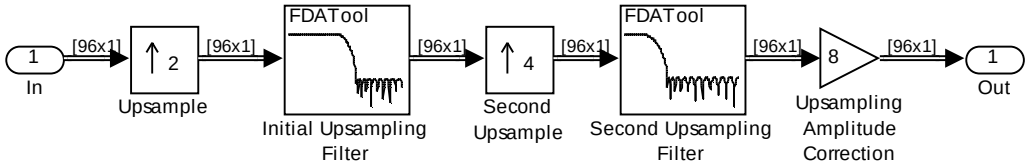
IFFT Mapping



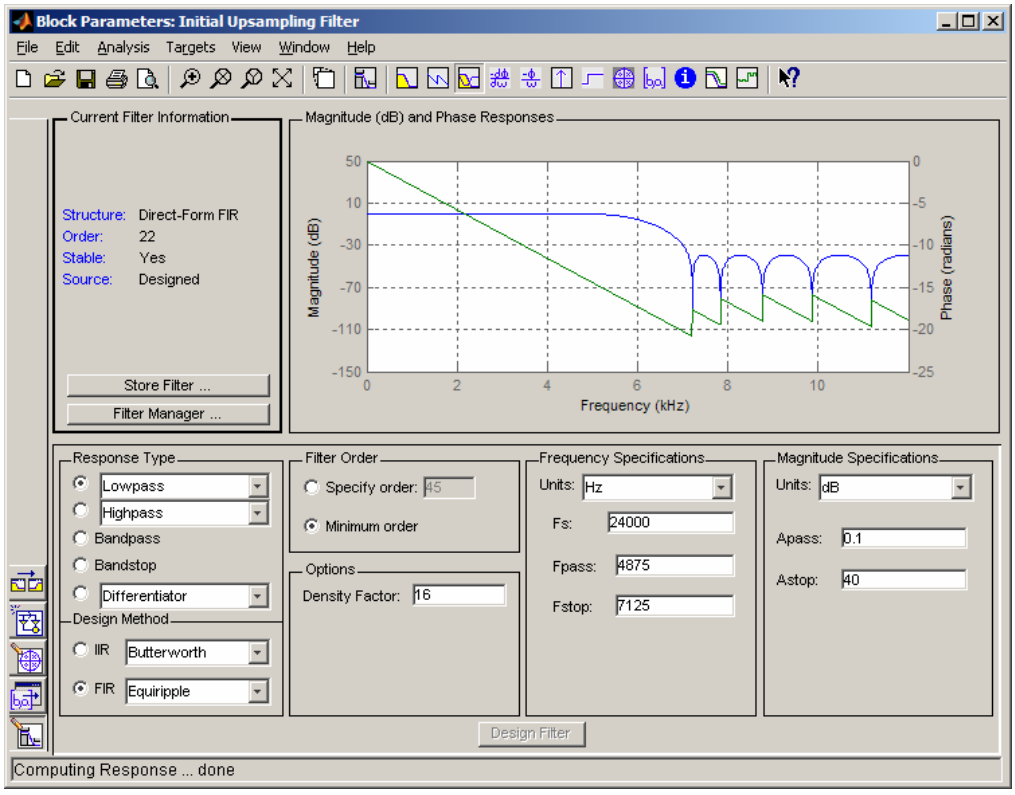
Apply Window



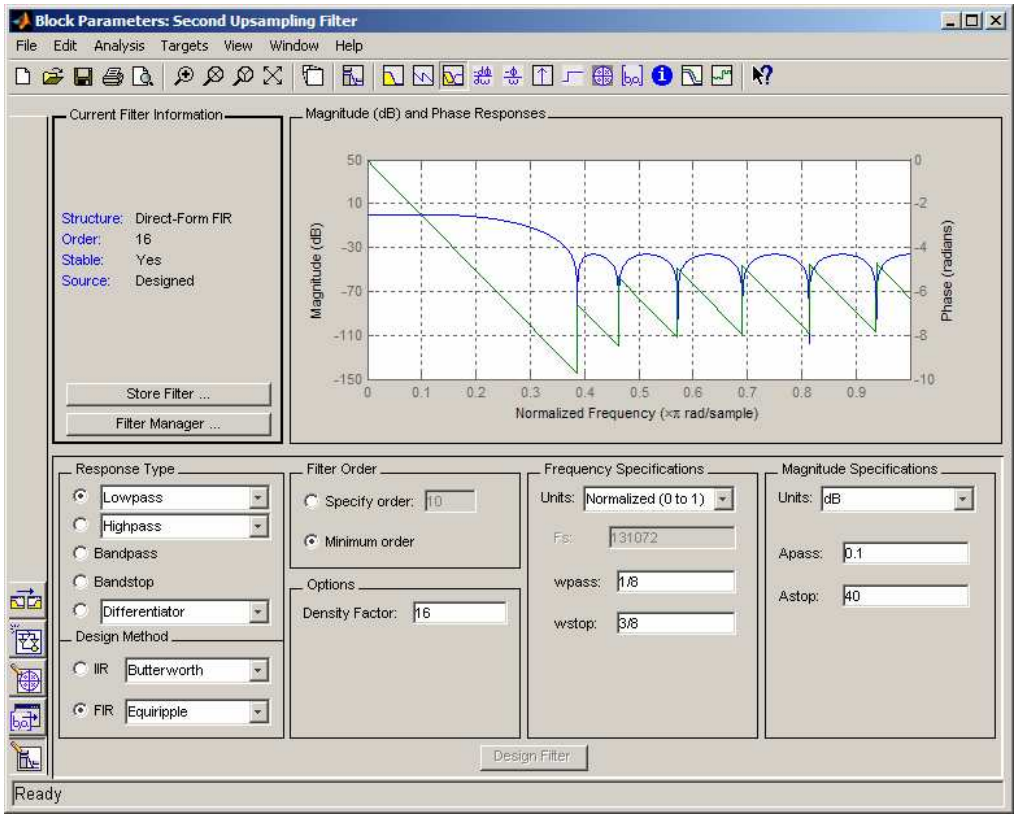
iii. Interpolation



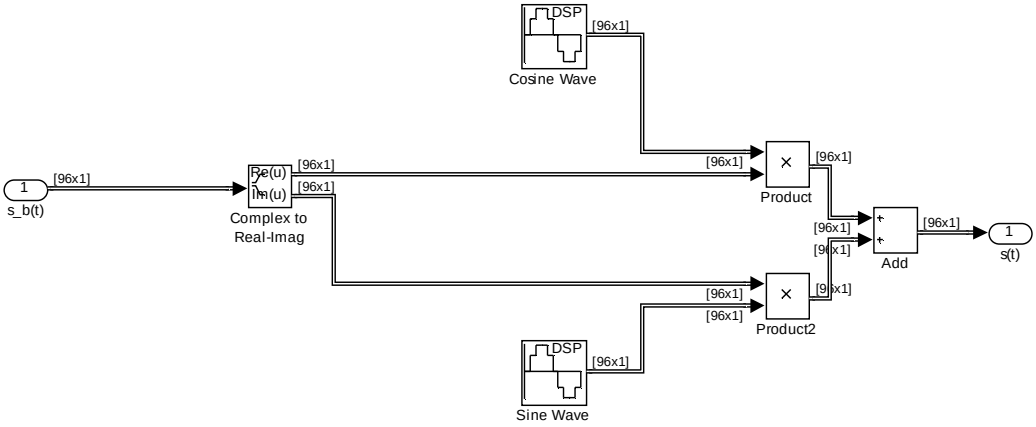
Initial Upsampling Filter



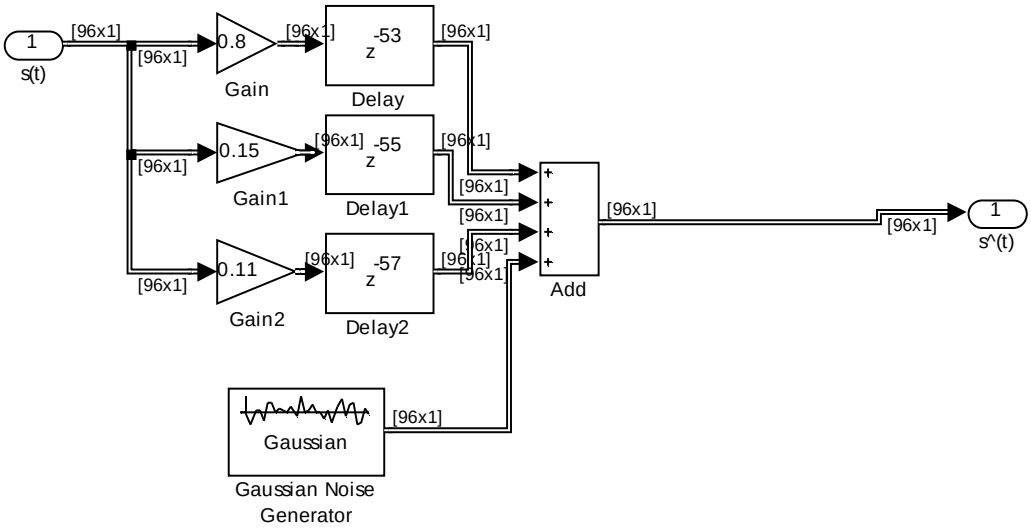
Second Upsampling Filter



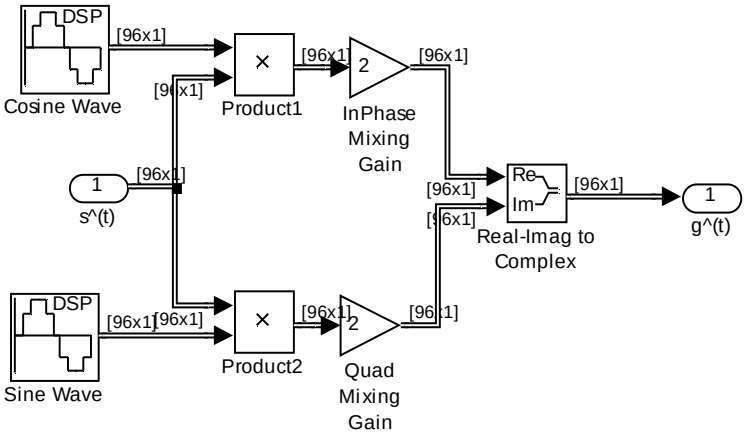
iv. Quadrature Modulator



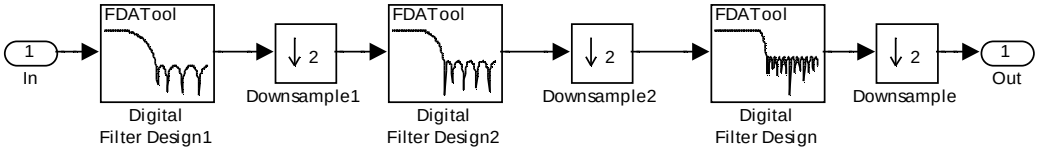
v. Channel



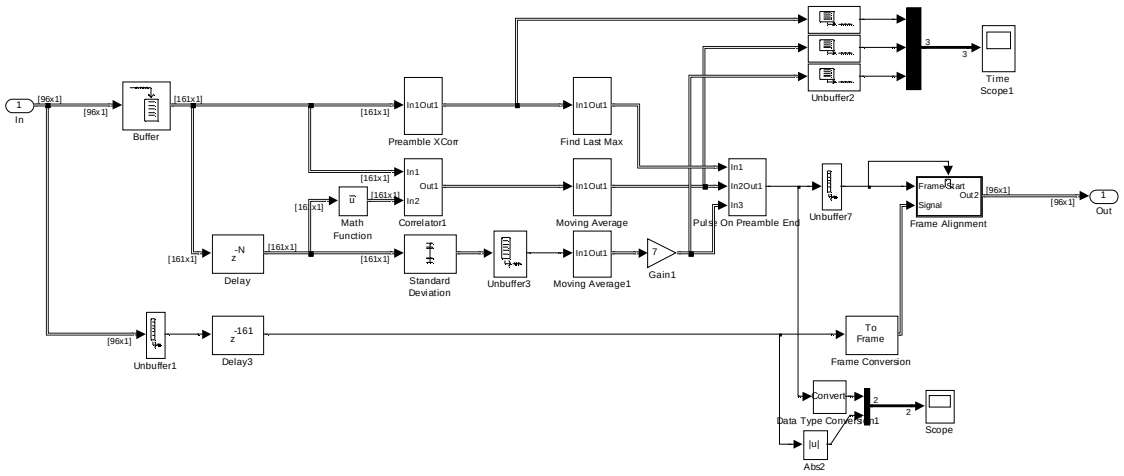
vi. Quadrature Demodulator



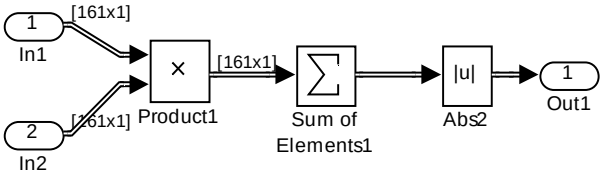
vii. Decimation



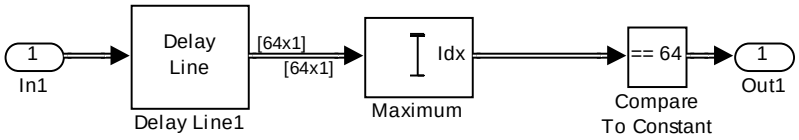
viii. Frame Synch



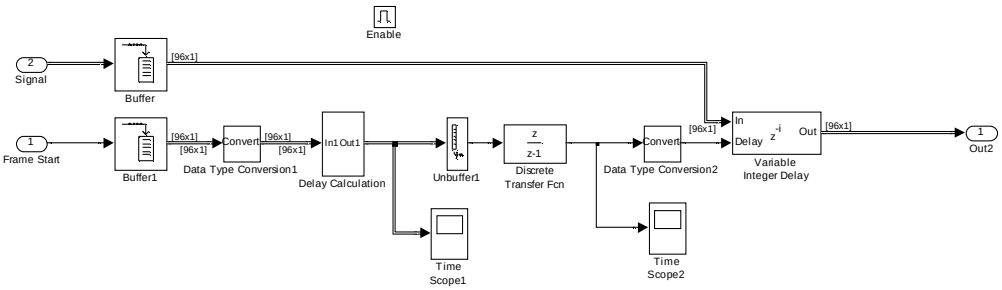
Correlator



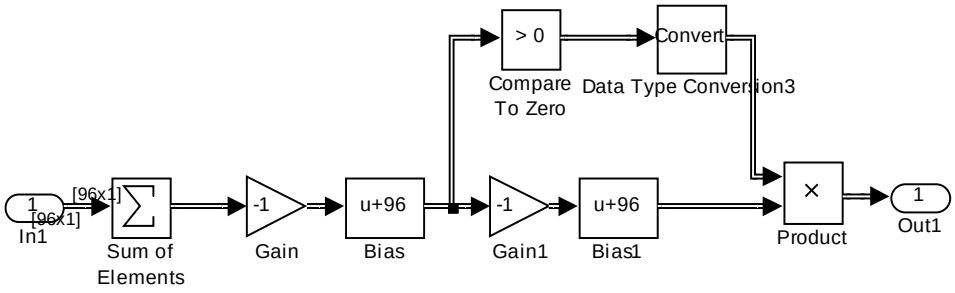
Find Last Max



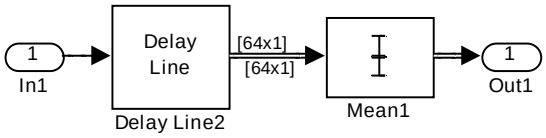
Frame Alignment



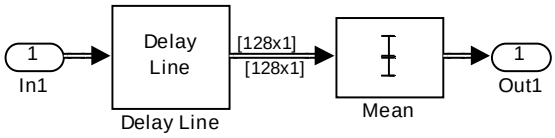
Delay Calculation



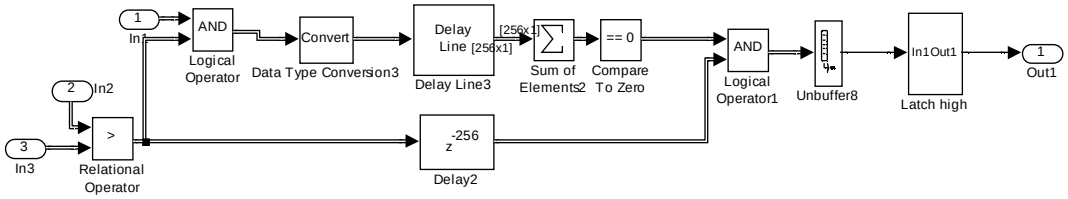
Moving Average 64



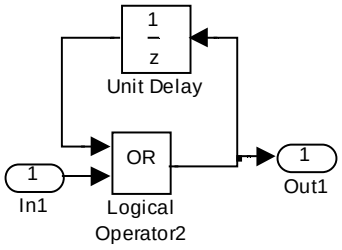
Moving Average 128



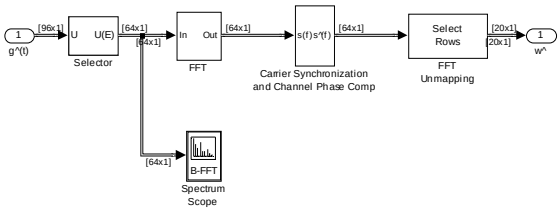
Pulse On Preamble End



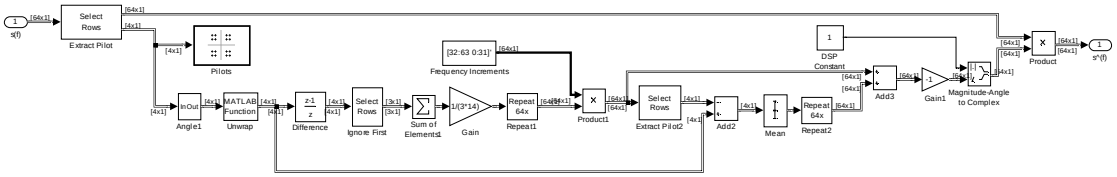
Latch high



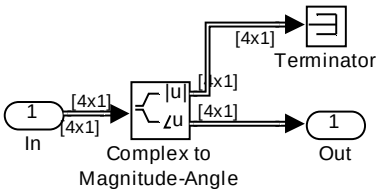
ix. OFDM Demodulation



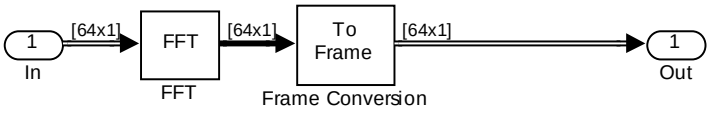
Carrier Synchronization and Channel Phase Comp



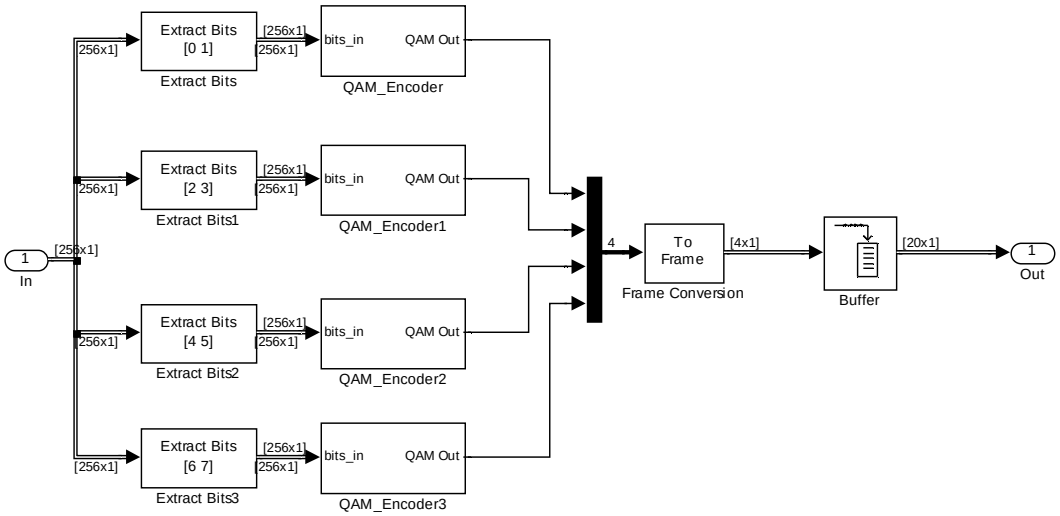
Angle1



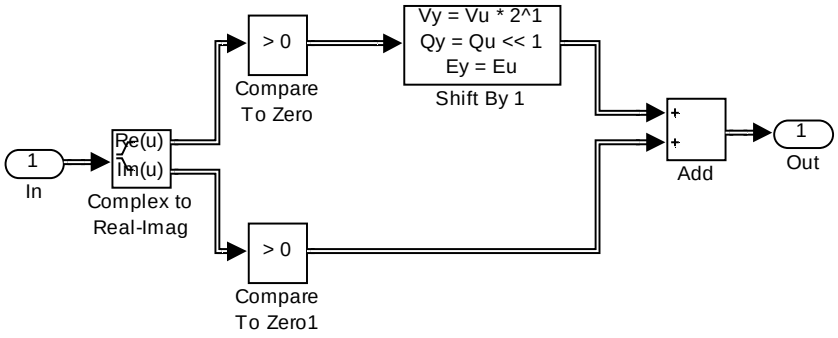
FFT



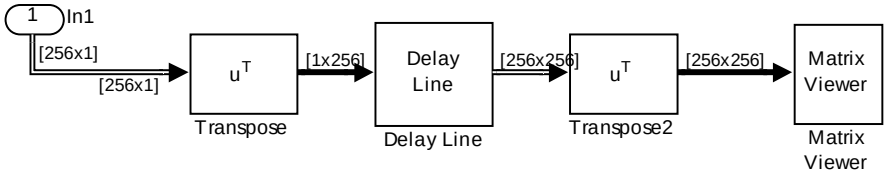
x. QAM Decoder



QAM Symbol Decoder



xi. Image Viewer



Appendix 2 – S-Block Source Code

i. Test_interface.c

```
// Testing_Interface.c
// Author: Luke Vercimak
// Bradley University Senior Capstone Project 2006, Software Radio
// Purpose:
// Testing_Interface contains the functions to implement RTDX
communication
// With the DSP board to send and receive test text messages. RTDX
comm. is
// not included when compiled as a mex module.

#include <stdio.h>
#include <stdlib.h>

// Include DSP RTW specific headers
#ifndef MATLAB_MEX_FILE

    #include "rtwtypes.h"
    #include <rtdx.h>
    #include "testcfg.h"

// Simulink MEX specific headers and defines
#else
    #include "tmwtypes.h"

    #define TRUE    1
    #define FALSE   0
#endif



// Declare and Initialize input and output channels
// Input channel has a name of rad_in
// Output channel has a name of rad_out

#ifndef MATLAB_MEX_FILE
RTDX_CreateInputChannel(rad_in);
RTDX_CreateOutputChannel(rad_out);
#endif

static unsigned int Busy;
//static unsigned int DataReady;

void Testing_Interface_Open()
{
    static unsigned init = false;

    if(!init)
    {
        Busy = FALSE;
        //DataReady = FALSE;
```

```
        // Enable the channels
        #ifndef MATLAB_MEX_FILE
        RTDX_enableInput(&rad_in);
        RTDX_enableOutput(&rad_out);
        #endif
    }
}

void Testing_Interface_Close()
{
    static unsigned close = false;

    if(!close)
    {
        // Disable the channels
        #ifndef MATLAB_MEX_FILE
        RTDX_disableInput(&rad_in);
        RTDX_disableOutput(&rad_out);
        #endif
    }
}

void Testing_Interface_In(uint8_T* data_out, uint8_T* DataReady_out,
uint8_T* busy_out)
{
    static unsigned char data_buffer[256];
    static unsigned int FirstTime = TRUE;

      if(Busy)
      {
        // Clean out output buffer
        memset(data_out, 0, 256);

        // Output data padded with nulls if first time, just nulls
otherwise
            if(FirstTime)
        {
            *DataReady_out = TRUE;
            strcpy(data_out, data_buffer);
            FirstTime = FALSE;
        }
        else
        {
            *DataReady_out = FALSE;
        }
    }
      else
      {
            int i = 0;

        // Disable Timer and set busy to be true
        #ifndef MATLAB_MEX_FILE
            TIMER_pause(hTimer1);
        #endif

            // Wait for data
            // RX Data
```

```
        #ifndef MATLAB_MEX_FILE
        for (i = 0; i < 255; i++ )
          {
                  char in_char = 0;

               // Request an integer from the host
                  while(RTDX_read( &rad_in, &in_char, sizeof(in_char))
!= sizeof(in_char));

                  // Copy inputted character into buffer
                  data_buffer[i] = in_char;

                  if(in_char == 0)
                        break;

          }

            // If buffer reached limit, put null at the end of buffer
to
            // make sure it is a valid input
            if(i == 255)
            {
                  data_buffer[255] = 0;
            }
        #else
        strcpy(data_buffer, "Test string.");
        #endif

             *DataReady_out = FALSE;
            Busy = TRUE;
        FirstTime = TRUE;

            // Enable timer
        #ifndef MATLAB_MEX_FILE
            TIMER_resume(hTimer1);
        #endif
      }
    *busy_out = Busy;
}

void Testing_Interface_Out(uint8_T* data_in, uint8_T DataReceived_in)
{
    char out_buffer[256];

      if(DataReceived_in)
      {

        // Disable Timer and set busy to be true
        #ifndef MATLAB_MEX_FILE
            TIMER_pause(hTimer1);
        #endif

        memset(out_buffer, 0, 256);
            memcpy(out_buffer, data_in, 256);
        out_buffer[255] = 0;

         // Spit string back out to host
```

```
        #ifndef MATLAB_MEX_FILE
          RTDX_write( &rad_out, out_buffer,
sizeof(char)*strlen(out_buffer)+1 );

          // Wait for Target-to-Host transfer to complete
          while ( RTDX_writing != NULL );
        #endif

            // Enable timer
        #ifndef MATLAB_MEX_FILE
            TIMER_resume(hTimer1);
        #endif

        Busy = FALSE;
    }
}
```

ii. Test_interface.h

```
extern void Testing_Interface_Open();
extern void Testing_Interface_Close();
extern void Testing_Interface_In(uint8_T* data_out, uint8_T*
DataReady_out, uint8_T* busy_out);
extern void Testing_Interface_Out(uint8_T* data_in, uint8_T
DataReceived_in);
```

iii. Testing_interface_in.c

```
#define S_FUNCTION_NAME testing_interface_in
#define S_FUNCTION_LEVEL 2
#include "simstruc.h"
#include "test_interface.h"

/*
 * mdlInitializeSizes - initialize the sizes array
 */
static void mdlInitializeSizes(SimStruct *S)
{
    // There are no parameters right now, later a sampling time will
    // Need to be added explicitedly
    ssSetNumSFcnParams( S, 1 ); /*number of input arguments*/

    // There are no input ports
    if (!ssSetNumInputPorts(S, 0)) return;

    // Set up 3 output ports, data, DataReady, and Busy
    if (!ssSetNumOutputPorts(S,3)) return;
    ssSetOutputPortWidth(S, 0, 1);
    ssSetOutputPortWidth(S, 1, 1);
    ssSetOutputPortWidth(S, 2, 1);

    // Set up data as a 256 byte frame
    ssSetOutputPortMatrixDimensions(S,  0, 256, 1);
    ssSetOutputPortDataType( S, 0, SS_UINT8 );
    ssSetOutputPortFrameData(S, 0, 1);

    // Set DataReady and Busy as single byte/sample time inputs
    ssSetOutputPortDataType( S, 1, SS_UINT8 );
    ssSetOutputPortDataType( S, 2, SS_UINT8 );

    // The whole block runs at a single sampling time
    ssSetNumSampleTimes( S, 1);

    // Init the testing interface
    Testing_Interface_Open();
}

/*
 * mdlInitializeSampleTimes - indicate that this S-function runs
 * at the rate of the source (driving block)
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    double *samp_time = mxGetPr(ssGetSFcnParam(S, 0));
    ssSetSampleTime(S, 0, *samp_time);
    ssSetOffsetTime(S, 0, 0.0);
}


/*
 * mdlOutputs - compute the outputs by calling my_alg, which
 * resides in another module, my_alg.c
```

```
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    // Get the pointers to the output signals and pass them to the
    // Testing_Interface function for work
    uint8_T* data_out = (uint8_T *)ssGetOutputPortSignal(S,0);
    uint8_T* DataReady_out = (uint8_T *)ssGetOutputPortSignal(S,1);
    uint8_T* busy_out = (uint8_T *)ssGetOutputPortSignal(S,2);

    Testing_Interface_In(data_out, DataReady_out, busy_out);
}
/*
 * mdlTerminate - called when the simulation is terminated.
 */
static void mdlTerminate(SimStruct *S)
{
    // Clean up the testing interface
    Testing_Interface_Close();
}

#ifdef MATLAB_MEX_FILE /* Is this file being compiled as a MEX-file? */
#include "simulink.c" /* MEX-file interface mechanism */
#else
#include "cg_sfun.h" /* Code generation registration function */
#endif
```

iv. Testing_interface_in.tlc

```
%% File    : testing_interface_in.tlc
%% Abstract:
%%      Inlined tlc file for testing_interface.c
%%

%implements "testing_interface_in" "C"

%% Function: BlockTypeSetup
%% ================================================
%% Abstract:
%%      Create function prototype in model.h as:
%%      "extern void (real_T u);"
%%
%function BlockTypeSetup(block, system) void
  %openfile buffer
    #include "test_interface.h"
  %closefile buffer
  %<LibCacheFunctionPrototype(buffer)>
%endfunction %% BlockTypeSetup

%% Function: InitializeConditions ============================
%%
%% Abstract:
%% Invalidate the stored output and input in rwork[1 ...
%% 2*blockWidth] by setting the time stamp (stored in
%% rwork[0]) to rtInf.
%%
%function InitializeConditions(block, system) Output
  /* %<Type> Block: %<Name> */
  Testing_Interface_Open();
  %<LibBlockOutputSignal(1, "", "", 0)>=0;
  %<LibBlockOutputSignal(2, "", "", 0)>=0;

%endfunction %% InitializeConditions

%% Function: Terminate ========================================
%% Abstract:
%%      X[i] = U[i]
%%
%function Terminate(block, system) Output
  /* %<Type> Block: %<Name> */
  Testing_Interface_Close();
%endfunction %% Terminate

%% Function: Outputs
%% ============================================================
%% Abstract:
%%      y = my_alg( u );
%%
%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %assign u = LibBlockOutputSignalAddr(0, "", "", 0)
  %assign v = LibBlockOutputSignalAddr(1, "", "", 0)
  %assign w = LibBlockOutputSignalAddr(2, "", "", 0)
```

```
  %% PROVIDE THE CALLING STATEMENT FOR "algorithm"
  Testing_Interface_In(%<u>,%<v>,%<w>);

%endfunction %% Outputs
```

v. Testing_interface_out.c

```
#define S_FUNCTION_NAME testing_interface_out
#define S_FUNCTION_LEVEL 2
#include "simstruc.h"
#include "test_interface.h"

/*
 * mdlInitializeSizes - initialize the sizes array
 */
static void mdlInitializeSizes(SimStruct *S)
{

    ssSetNumSFcnParams( S, 0); /*number of input arguments*/

    if (!ssSetNumInputPorts(S, 2)) return;
    ssSetInputPortWidth(S, 0, 1);
    ssSetInputPortWidth(S, 1, 1);
    ssSetInputPortDirectFeedThrough(S, 0, 1);
    ssSetInputPortDirectFeedThrough(S, 1, 1);

    ssSetInputPortMatrixDimensions(S,  0, 256, 1);
    ssSetInputPortRequiredContiguous(S, 0, 1);
    ssSetInputPortDataType( S, 0, SS_UINT8 );
    ssSetInputPortFrameData(S, 0, 1);

    ssSetInputPortDataType( S, 1, SS_UINT8 );

    if (!ssSetNumOutputPorts(S,0)) return;

    ssSetNumSampleTimes( S, 1);

    Testing_Interface_Open();
}

/*
 * mdlInitializeSampleTimes - indicate that this S-function runs
 * at the rate of the source (driving block)
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}


/*
 * mdlOutputs - compute the outputs by calling my_alg, which
 * resides in another module, my_alg.c
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    uint8_T* data_in = (uint8_T *)ssGetInputPortSignal(S,0);
    uint8_T* DataReceived_in = (uint8_T *)ssGetInputPortSignal(S,1);

    Testing_Interface_Out(data_in, *DataReceived_in);
```

```
}
/*
 * mdlTerminate - called when the simulation is terminated.
 */
static void mdlTerminate(SimStruct *S)
{
    Testing_Interface_Close();
}

#ifdef MATLAB_MEX_FILE /* Is this file being compiled as a MEX-file? */
#include "simulink.c" /* MEX-file interface mechanism */
#else
#include "cg_sfun.h" /* Code generation registration function */
#endif
```

vi. Testing_interface_out.tlc

```
%% File    : testing_interface_out.tlc
%% Abstract:
%%      Inlined tlc file for testing_interface_out
%%

%implements "testing_interface_out" "C"

%% Function: BlockTypeSetup
%% ================================================
%% Abstract:
%%      Create function prototype in model.h as:
%%      "extern void (real_T u);"
%%
%function BlockTypeSetup(block, system) void
  %openfile buffer
    #include "test_interface.h"
  %closefile buffer
  %<LibCacheFunctionPrototype(buffer)>
%endfunction %% BlockTypeSetup

%% Function: Outputs
%% ============================================================
%% Abstract:
%%      y = my_alg( u );
%%
%function Outputs(block, system) Output
  /* %<Type> Block: %<Name> */
  %assign u = LibBlockInputSignalAddr(0, "", "", 0)
  %assign v = LibBlockInputSignal(1, "", "", 0)

  %% PROVIDE THE CALLING STATEMENT FOR "algorithm"
  Testing_Interface_Out(%<u>,%<v>);

%endfunction %% Outputs
```

Appendix 3 – RTDX Controlling Program Source

```
//RTDX application interface
//Bradley University Senior Capstone Project:  sradio
//Author:  Karl Weyeneth
//Created:  4/8/06

//make sure that the board is connected through
//code composer studio, and the test.out file is loaded
//on the board and running, before you run this program

//Last Revised: 5/5/06

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using RTDXINTLib;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;



namespace SRadio_Send_Recieve_Interface
{
    public partial class Form1 : Form
    {
        //creating RTDX objects
        private RTDXINTLib.RtdxExpClass rtdx = new
RTDXINTLib.RtdxExpClass();
        private RTDXINTLib.RtdxExpClass rtdxReadCH = new
RTDXINTLib.RtdxExpClass();
        private RTDXINTLib.RtdxExpClass rtdxWriteCH = new
RTDXINTLib.RtdxExpClass();

        public Form1()
        {
            InitializeComponent();

        }
        int nMessages; //number of messages in RTDX
        int sBytes; //number of bytes send
        byte[] TextMessage; // byte array of the text message that is
sent to board
        int sendcount = 0; //keeps count for messages sent back from
board

        private void Output_rd_CheckedChanged(object sender, EventArgs
e)
        {
            //Form formatting when radio button changed
            Recieve_btn.Enabled = true;
            Output_tb.Enabled = true;
```

```
            Send_btn.Enabled = false;
            Input_tb.Enabled = false;
        }

        private void Input_rb_CheckedChanged(object sender, EventArgs
e)
        {
            //Form formatting when radio button changed
            Send_btn.Enabled = true;
            Input_tb.Enabled = true;
            Recieve_btn.Enabled = false;
            Output_tb.Enabled = false;

        }

        private void InFile_rb_CheckedChanged(object sender, EventArgs
e)
        {
            //Form formatting when radio button changed
            InFile_tb.Enabled = true;
            Browse_In_btn.Enabled = true;
            Send_File_btn.Enabled = true;
            Rec_File_btn.Enabled = false;
            OutFile_tb.Enabled = false;
            Browse_Out_btn.Enabled = false;

        }

        private void OutFile_rb_CheckedChanged(object sender, EventArgs
e)
        {
            //Form formatting when radio button changed
            OutFile_tb.Enabled = true;
            Browse_Out_btn.Enabled = true;
            Rec_File_btn.Enabled = true;
            Send_File_btn.Enabled = false;
            InFile_tb.Enabled = false;
            Browse_In_btn.Enabled = false;

        }

        private void Browse_In_btn_Click(object sender, EventArgs e)
        {
            //File Dialog for sendign a file to RTDX
            openFileDialog1.FileName = "";
            openFileDialog1.InitialDirectory = @"C:\Documents and
Settings\luke\My Documents";
            openFileDialog1.Title = "Select A File";
            openFileDialog1.ShowDialog();
            InFile_tb.Text = openFileDialog1.FileName;
        }

        private void Browse_Out_btn_Click(object sender, EventArgs e)
        {
            //File Dialag for receiving a file from RTDX
            openFileDialog1.FileName = "";
```

```
            openFileDialog1.InitialDirectory = @"C:\Documents and
Settings\luke\My Documents";
            openFileDialog1.Title = "Select A File";
            openFileDialog1.ShowDialog();
            OutFile_tb.Text = openFileDialog1.FileName;
        }

        private void Send_btn_Click(object sender, EventArgs e)
        {
            int sendTextLength;
            sendTextLength = Input_tb.Text.Length;
            statusRTDX.Text = "Sending Message...";
            Output_tb.Text= "";
            Output_tb.Enabled = false;
            Input_tb.Text.Trim(); //removes spaces before and after the
message
            //If the message is less then 255 character and if there is
a message then
            if(sendTextLength <= 255)
            {
            if (Input_tb.Text != "")
            {
                //turning the text into a byte array
                TextMessage = StrToByteArray(Input_tb.Text);
                byte[] ByteMessage = new byte[TextMessage.Length + 1];
                //A byte array must be created and the last byte in the
array must be a 0
                //because the RTDX link needs a null value at the end
of the message
                ByteMessage[ByteMessage.Length - 1] = 0;
                //Mapping the Text byte array to the new byte array
that will be sent
                for (int i = 0; i < TextMessage.Length; ++i)
                {
                    ByteMessage[i] = TextMessage[i];
                }
                //Writing the byte array to the RTDX channel
                if (0 != rtdxWriteCH.Write(ByteMessage, out sBytes))
                {
                    MessageBox.Show(this, "Failed to write to the
RTDX.", "Write Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                    this.Close();
                }
                statusRTDX.Text = "Message Sent Successfully!";

            }
            else
            {
                MessageBox.Show("No Text was entered to send");
                return;
            }
            }
            else
            {
                MessageBox.Show("The Message is to long to send");
                return;
            }
```

```
            //Form Formatting after message is sent
            Send_btn.Enabled = false;
            Recieve_btn.Enabled = true;
            Output_rd.Checked = true;
            Input_rb.Checked = false;
            Output_tb.Enabled = false;
        }
        //This Function creates a Byte Array from a String
        public static byte[] StrToByteArray(string str)
        {
            System.Text.ASCIIEncoding encoding = new
System.Text.ASCIIEncoding();
            return encoding.GetBytes(str);
        }

        private void Recieve_btn_Click(object sender, EventArgs e)
        {
            //After the receive button is pressed the timer control
starts
            statusRTDX.Text = "Receiving Message....";
            Wait4RetMess.Start();
         }

        private void btnConnect_Click(object sender, EventArgs e)
        {
            Send_btn.Enabled = true;
            btnConnect.Enabled = false;
            const string board = "C6713 DSK"; //name of TI board
            const string processor = "CPU_1"; //name of the processor
on board
            statusRTDX.Text = "Connecting.....";
            //initializing communication with board and proccesor
            rtdx.SetProcessor(board, processor);

            if (0 != rtdx.EnableRtdx()) //enable rtdx
            {
                MessageBox.Show(this, "Failed to enable RTDX.", "Init
Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                this.Close();
            }
            if (0 != rtdxReadCH.Open("rad_out", "r")) //open a channel
for reading from the board
            {
                MessageBox.Show(this, "Failed to open RTDX read
channel.", "Con Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                this.Close();
            }
            if (0 != rtdxWriteCH.Open("rad_in", "w")) //open a channel
for Writing to the board
            {
                MessageBox.Show(this, "Failed to open RTDX write
channel.", "Con Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                this.Close();
            }
            if (0 != rtdxReadCH.EnableChannel("rad_out"))  //enable a
channel
            {
```

```
                MessageBox.Show(this, "Failed to enable RTDX read
channel.", "Con Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                this.Close();
            }
            if (0 != rtdxWriteCH.EnableChannel("rad_in"))  //enable a
channel
            {
                MessageBox.Show(this, "Failed to enable RTDX write
channel.", "Con Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                this.Close();
            }

            btnDisconnect.Enabled = true;
            statusRTDX.Text = "Connected!";
        }

        private void btnDisconnect_Click(object sender, EventArgs e)
        {
            Send_btn.Enabled = false;
            statusRTDX.Text = "Disconnecting...";
            btnDisconnect.Enabled = false;

            if (0 != rtdxReadCH.DisableChannel("rad_in"))//disabling
channel
            {
                MessageBox.Show(this, "Failed to disable RTDX read
channel.", "Dis Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                this.Close();
            }
            if (0 != rtdxReadCH.Close())//closing channel
            {
                MessageBox.Show(this, "Failed to close RTDX read
channel.", "Dis Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                this.Close();
            }

            if (0 != rtdxWriteCH.DisableChannel("rad_out"))
            {
                MessageBox.Show(this, "Failed to disable RTDX write
channel.", "Dis Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                this.Close();
            }
            if (0 != rtdxWriteCH.Close())
            {
                MessageBox.Show(this, "Failed to close RTDX write
channel.", "Dis Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                this.Close();
            }

            if (0 != rtdx.DisableRtdx())
            {
                MessageBox.Show(this, "Failed to disable RTDX.", "Dis
Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
                this.Close();
            }
            btnConnect.Enabled = true;
            statusRTDX.Text = "Disconnected";
```

```
            sendcount = 0;
            Input_tb.Text = "";
            Output_tb.Text = "";
        }
        //The timer control created runs every 10sec and checks for a
message
        //that is ready to be sent from the board through RTDX
        private void Wait4RetMess_Tick(object sender, EventArgs e)
        {
            Wait4RetMess.Stop();
            object pArr;
            //Checking for messages ready to receive
            rtdxReadCH.GetNumMsgs(out nMessages);
            string oMessage;  //string that will be sent to the output
text box

            //if the number of messages on the channel is greater than
            //the number sent back from the channel then there is a new
            //message
            if (nMessages > sendcount)
            {
                //data is returned in pArr
                rtdxReadCH.ReadSAI1(out pArr);
                //pArr is changes to a byte array
                byte[] rbMessage = (byte[])pArr;
                System.Text.ASCIIEncoding enc = new
System.Text.ASCIIEncoding();
                //the byte array is changed back into text
                oMessage = enc.GetString(rbMessage);
                Output_tb.Text = oMessage;
                sendcount++;
            }
            else
            {
                //if the the counts are equal then the timer starts
again
                Wait4RetMess.Start();
                return;
            }
            //Form formatting
            statusRTDX.Text = "Message Received Successfully!";
            Send_btn.Enabled = true;
            Recieve_btn.Enabled = false;
            Output_rd.Checked = false;
            Input_rb.Checked = true;
        }
    }
}
```

Appendix 4 – Matlab Programs

i. Soft_radio_init.m

```
% soft_radio_init.m
% Software Radio Initialization
% Bradley University Senior Capstone Project 2006
% Authors: Luke Vercimak and Karl Weyeneth

% Constants for radio
FFTsize = 64;
Fs = 96000;
Ts = 1/Fs;
Fmod = Fs/4;

% Windowing Calculations for OFDM packet
k = -24:-17;
coef = 0.5*(1-cos(pi.*(k./125+0.192)./0.064));
w = [coef ones(1, 80) coef(8:-1:1)];

% Preamble Short Sequence
short_seq = (sqrt(13/6))*[0, 0, 1+j, 0, 0, 0, -1-j, 0, 0, 0, 1+j, 0, 0,
0, -1-j, 0, 0, 0, -1-j, 0, 0, 0, 1+j, 0, 0, 0, 0, 0, 0, 0, -1-j, 0, 0,
0, -1-j, 0, 0, 0, 1+j, 0, 0, 0, 1+j, 0, 0, 0, 1+j, 0, 0, 0, 1+j, 0,0];
short_seq = [zeros(1,6), short_seq, zeros(1,5)];
short_seq = [short_seq(33:64),short_seq(1:32)];
short_seq_t = ifft(short_seq);
short_seq_r = [short_seq_t, short_seq_t, short_seq_t(1:33)];

% window the preamble short sequence
short_seq_r(1) = short_seq_r(1).*0.5;
short_seq_r(161) = short_seq_r(161).*0.5;

% Assemble the whole preamble
preamble = [];
for i = 1:10,
      preamble = [preamble, short_seq_r];
end

% Add some random noise to the beginning of the preamble so things line
up
l=214;
pre_sd_real = std(real(preamble));
pre_sd_imag = std(imag(preamble));
preamble_whole=[pre_sd_real*randn(1,l)+j*pre_sd_imag*randn(1,l),
preamble];

% Channel Control Variables
pnoise = 0.0005;
freq_off = 10;
noiseseed = 8;
delay1 = 55;
atten1 = 0.1;
delay2inc = 2;
```

```
atten2 = 0.78;
delay3inc = 2;
atten3 = 0.25;

% Read in image for input for testing the radio
test_img = imread('sim_br_hall.jpg');
test_img = test_img(:,:,1);
data = uint8(reshape(test_img, 256*256,1));
a.time = []; %datat'f;
a.signals(1).values = data;
a.signals(1).dimensions = 1;
```