A Hybrid DSP and FPGA System for Software Defined Radio Applications

Volodymyr S. Podosinov

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science
In
Electrical Engineering

Majid Manteghi, Chair
William A. Davis
Patrick R. Schaumont

7th April 2011
Blacksburg, VA

Keywords: software defined radio, sdr, fpga, dsp, bit error rate testing, C64x+

A Hybrid DSP and FPGA System for Software Defined Radio Applications

Volodymyr S. Podosinov

ABSTRACT

Modern devices provide a multitude of services that use radio frequencies in continual smaller packages. This size leads to an antenna used to transmit and receive information being usually very inefficient and a lot of power is wasted just to be able to transmit a signal. To mitigate this problem a new antenna was introduced by Dr. Manteghi that is capable of working efficiently across a large band. The antenna achieves this large band by doing quick frequency hopping across multiple channels. In order to test the performance of this antenna against more common antennas, a software radio was needed, such that tested antennas can be analyzed using multiple modulations.

This paper presents a software defined radio system that was designed for the purpose of testing the bit-error rate of digital modulations schemes using described and other antennas. The designed system consists of a DSP, an FPGA, and commercially available modules. The combination allows the system to be flexible with high performance, while being affordable. Commercial modules are available for multiple frequency bands and capable of fast frequency switching required to test the antenna. The DSP board contains additional peripherals that allows for more complex projects in the future. The block structure of the system is also very educational as each stage of transmission and reception can be tested and observed.

The full system has been constructed and tested using simulated and real signals. A code was developed for communication between commercial modules and the DSP, bit error rate testing, data transmission, signal generation, and signal reception. A graphical user interface (GUI) was developed to help user with information display and system control. This thesis describes the software-defined-radio design in detail and shows test results at the end.

Acknowledgements

Through my endeavors I have received help from many people. I would like to first thank Dr. Majid Manteghi, my advisor, for sponsoring my master's thesis project. I also thank him for supervising the progression of this project and ensuring its success.

I also would like to thank people of Mobile Portable Research Group (MPRG), specifically Dr. Michael Buehrer and his students for providing advice on the design of the system, and providing source code samples.

Thanks to Dr. Patrick Schaumont for assisting me with Verilog coding, and also contributing advice on device selection.

Thank you to all the members of the TI E2E (Texas Instruments, Engineer-to-engineer) online community, specifically the members associated with the Digital Signal Processors (DSP) forums, for your knowledge and help.

Overall I would like to say thank you to all the students and faculty in the Virginia Tech Antenna Group (VTAG) lab, and all my friends who offered suggestions, and kept me motivated throughout the duration of my thesis.

Finally, I would like to thank my fiancé Kimberly Virag, for her unconditional support throughout the process of completing my master's thesis.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Motivation for the project

The original idea from which this project started was to build a bit-error-rate testing platform for the compact, tunable antenna shown in Figure 1.1 and described in more detail in [1]. Modern devices, commercial or military, span multiple frequency bands and either require multiple antennas to operate efficiently or more commonly use one wide-band antenna, which operates inefficiently. The antenna presented in [1] has a narrow bandwidth at one particular frequency, but is very efficient. At the same time, using a control voltage it is able to readjust its center frequency and still be efficient and narrowband.

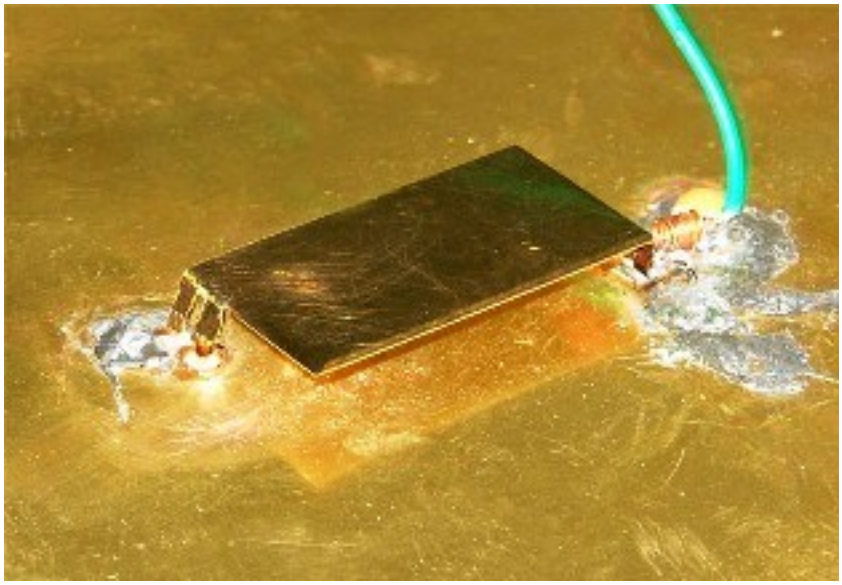

Figure 1.1: Tunable Planar Inverted-F Antenna

Using this antenna, a device can be implemented that works as shown in Figure 1.2. Each frequency would be used to send and transmit a signal with a particular bandwidth and modulation. In essence it would be similar to using orthogonal frequency division multiplexing (OFDM). A tunable antenna would work well with software-defined radios as each tunable channel might use different modulation and protocol.

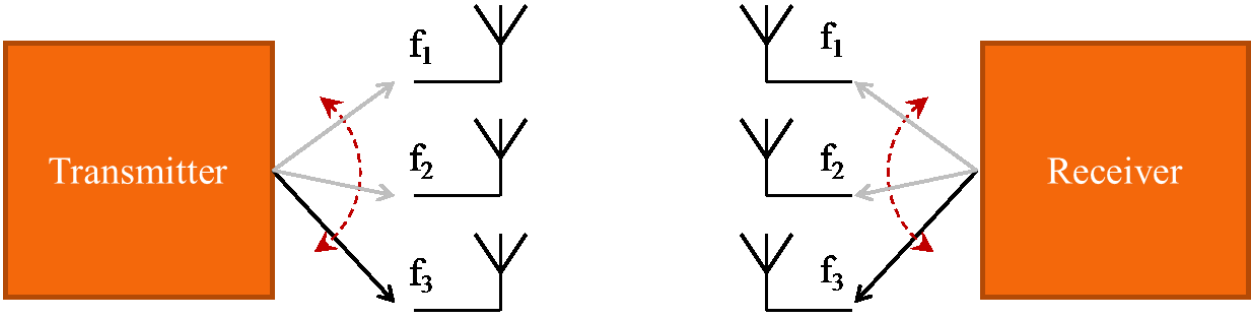

Figure 1.2: Sending and receiving signals using multi-band antenna

When the project started the main requirements were to be able to perform bit-error-rate testing, support multiple modulations (to see how each modulation behaves with this antenna), work with up to 10 MHz of spectrum at a time, and have the ability for fast frequency re-tuning such that operation in Figure 1.2 could be accomplished. The software-defined-radio platform also needed to have a reasonable price.

1.2 Software Defined Radio

The best way to define a software-defined radio is as a radio where radio components, such as mixer, detector, and filter, are implemented in the software. In software, a mixer will become a multiplier, and a filter will become a set of multiply-and-accumulate (MAC) operations. Other operations can be converted as well. An ideal software-defined radio is shown in Figure 1.3. A processor, which can be a DSP, FPGA, or a general purpose processor, takes digital data directly from an analog-to-digital converter (ADC) and sends data directly to a digital-to-analog (DAC) converter. The ADC and DAC use signal directly from the antenna with the minimal amount of the analog circuitry.

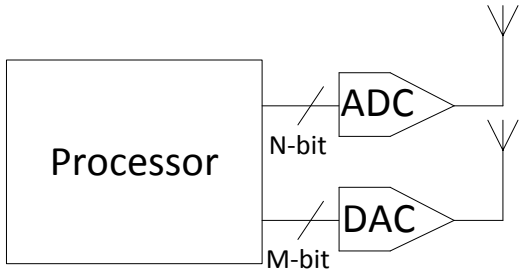

Figure 1.3: Ideal software defined radio

Currently, an ideal software defined radio is almost impossible as a lot of the signals are located high in the frequency spectrum and are very weak at the receiver, which would require a very fast ADC and DAC with a lot of bits. Fast ADC and DAC modules use a lot of energy and cost a lot of money and usually can digitize only a few bits. Processing the signal at such a high data rate also requires a fast processor.

There are two more practical approaches. One is to use some IF frequency where the signal would be first down-converted or up-converted before being digitized, as shown in Figure 1.4. This process is called heterodyning. Using system in Figure 1.4, the ADC and DAC need to be faster than twice the bandwidth of the signal, called Nyquist frequency, but not twice the RF frequency. This is because actual baseband information needs to be sampled, rather than whole RF spectrum. The structure shown in Figure 1.4 is being used for this project.

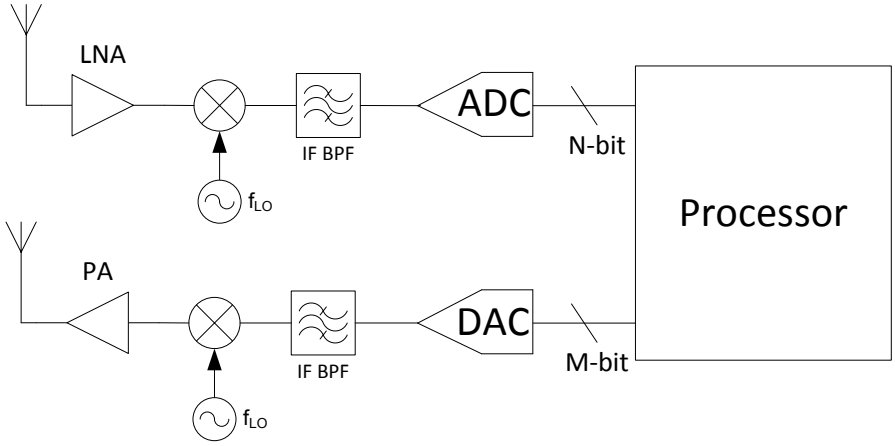

Figure 1.4: Heterodyne software defined radio

Another way to sample data at lower than Nyquist frequency is to use subsampling. Subsampling is sampling at least at twice the bandwidth of the signal, but not at twice the maximum signal frequency. Doing so causes the spectrum to be replicated at multiples of sampling frequency. Subsampling is almost the same as IF sampling, except sampled signal will be replicated at some frequency in digital domain. The ADC needs to support a larger bandwidth than its sampling rate in this case. Example of subsampling is having an ADC with the bandwidth of 100 MHz, and a maximum sampling rate of 10 MHz. Using this example a signal that is located up to 100 MHz in frequency with up to 5MHz bandwidth can be sampled directly, without use of down-conversion. After sampling a replica of the signal will be located at the 10 MHz intervals.

The advantage of the software radio is its flexibility. One of the best examples is in cellular communications. Two of the common types of cellular networks are GSM and CDMA. If the user wants to switch from a GSM network to a CDMA network they need to buy a new headset. With software defined radio, all they would need to do is to reprogram the headset to operate on a different network. Currently the disadvantage of the software defined radio is the power consumption. The ADC and DAC both consume more power as the sampling rate increases, and the processor also consumes more power with a faster clock rate. With advancing technologies, eventually this problem should be resolved.

1.3 Alternative platforms

At the time when the project started there were only couple software defined radio platforms on the market. The most common platform used was the USRP platform from Ettus Research. USRP platform consist of high speed ADC and DAC that pass data through the FPGA that contains decimation and interpolation filters. After that data is transmitted to and from the PC via the USB port. The rest of the work is done on the PC. There is a large community support for the USRP with the main one being GnuRadio. Towards the end of the project Ettus Research developed USRP2 platform, and currently they are in a process of phasing out USRP2 platform with more advance versions of their products. USRP platform costs $700 and USRP2 platform costs $1400. Daughterboard for all USRP platforms cost from $50 to $500 depending on the frequency range covered. Main idea behind USRP platforms is to digitize the signal and transfer it to PC for further processing.

Another software defined radio product is called Wireless Open-Access Research Platform (WARP) from Rice University. It is based on Virtex-II and Virtex-4 FPGA from Xilinx. Rice University WARP platform is an embedded solution, with available free source code. All of the operations are done on the FPGA. WARP platform is very expensive though, with single transceiver kit costing $8500.

One of the other commercial companies that make software defined radios is Lyrtech. Lyrtech specializes in hybrid software defined radio platforms such as this project. Their devices are also based around FPGA and DSP working together. Similarly to the WARP platform, Lyrtech devices are relatively expensive starting around $2500. Lyrtech platforms can also be

used as stand-alone devices without the use of PC. Lyrtech does not provide a free source code for their products.

For this project, some of the components used were from the company ComBlock. Although ComBlock does not make a complete software defined radio, they make rapid prototyping modules that a software defined radio can be made from. A fully functional software defined radio can be built from multiple ComBlock blocks. The software for the blocks is not freely available, but can be purchased from ComBlock as well. Without paying for the software a simple software defined radio from ComBlock can be built for $1500 or so. Any additional component or features, such as error coding or spreading module would cost additional money. There is also an option to buy software and deploy it on some other platform.

There are other software defined radios on the market from companies such as FlexRadio, but they are targeted toward specific application and therefore cannot be used for any universal application.

From all the products presented, WARP platform and Lyrtech platform were too expensive to base the project on. ComBlock company was not discovered until after the project was started and DSP board was already purchased. Using ComBlocks as a solution though would increase price if any other features would need to be implemented, and some features could not be implemented at all. USRP platform was a good starting point, as it had a freely available source code, and was relatively cheap. USRP had couple of disadvantages. First, USRP did not have an ability to synchronize frequency switching with antenna. Second, although it was not part of the requirement, USRP cannot be used as stand-alone device without the PC. It is possible to use embedded computer with USRP however. Third, USRP data stream is processed by the PC processor which does not have optimized instructions for signal processing like DSP, and USRP's internal FPGA is hard to modify. Overall all of these disadvantages can be overcome by using embedded board with USRP, such as Beagle Board for example.

1.4 Structure of the transceiver and thesis organization

Completely finished software defined radio transceiver is shown in Figure 1.5 and its block diagram is shown in Figure 1.6. Quadrature up-converter and down-converter modules are interchangeable for different frequency bands. Currently project is built with the L-band

modules. Up-converter frequency range is from 950 – 1450 MHz, and down-converter range is from 900 – 1575 MHz. Up-converter and down-converter have an on-board 10 MHz synthesizer reference, but there is also an optional input for the external oscillator. Maximum output power from up-converter is around 0 dBm, while maximum input power to the receiver should not exceed -5 dBm. The transceiver is built around Texas Instruments TMS320C6455 DSP, which uses a newer 64x+ architecture DSP compared to previous devices. DSP is located on the development board from Spectrum Digital. The development board, abbreviated DSK, has 3 peripheral connectors, 16-bit 96 kHz audio ADC and DAC, 1 Gbit Ethernet, and on-board 4 Mbit Flash memory. Additional features can be referenced from [5]. Quadrature down-converter is connected to the FPGA such that some pre-processing can be done before data is sent to the DSP, as the sampling rate is too high.

Figure 1.5: Photograph of the completely assembled transceiver

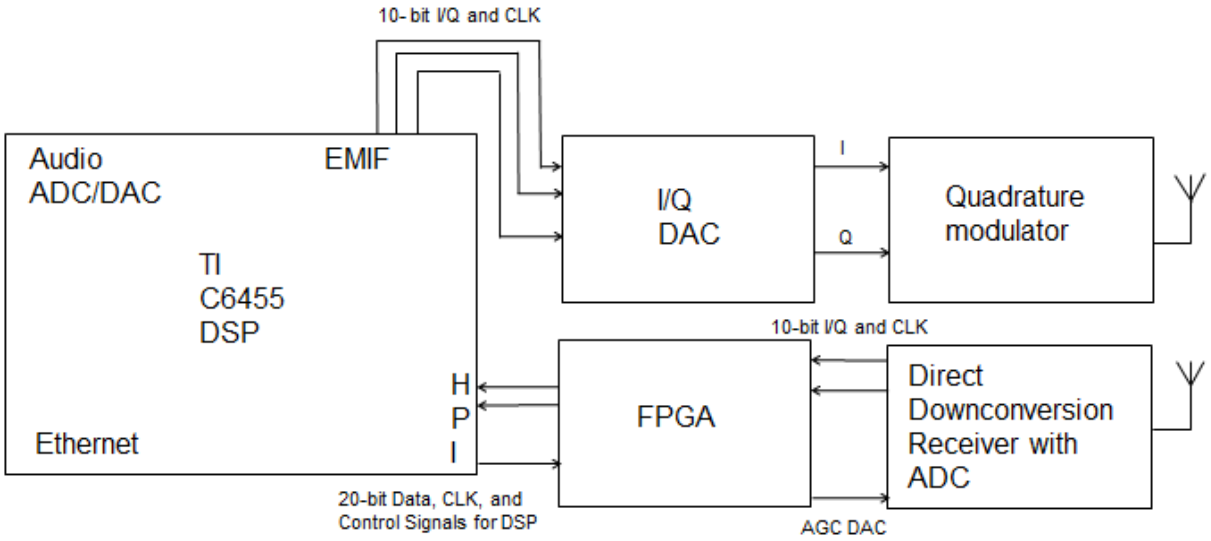

Figure 1.6: Block diagram of the transceiver

All system is powered from the single power supply that DSK uses. A cable (now shown) connects DSK to the other modules and provides them with power. A maximum current consumption of the modules should not exceed 1A.

The thesis is divided into 4 main chapters that follow transceiver's structure. Chapter 2 describes hardware structure of the transmitter and all the connections made. Chapter 3 describes hardware structure of the receiver including FPGA pre-processing modules. Chapter 4 focuses on all DSP software, and is divided into transmitter, receiver, and additional software sections. Chapter 5 shows all the testing performed with the transceiver and results that were obtained. Appendix at the end of the thesis contains some of the source code used in the project. The rest of the code is provided separately in a file.

1.5 Software Used for the Project

For this project multiple software packages were used. One of the main software packages used was MATLAB 2009b. MATLAB was used to design loop filter for the decision directed phase locked loop described in 4.3.2. It was used to test ideas before implementing them in the C or Verilog code, and it was used to generate waveform data to be stored inside of the FPGA. MATLAB's main packages that were used during design were Fixed Point toolbox, Control Systems toolbox, Filter Design Toolbox, and Communications Toolbox. Communication toolbox "modem" object was used to determine gray coding pattern, and to determine how FSK

baseband is mapped. Fixed Point toolbox was used to convert decimal number representation to the fractional integer notation used in the FPGA and the DSP. Control Systems toolbox was used to design a PI controller for the phase locked loop.

All of the Verilog and VHDL code that was written for the FPGA was simulated using Mentor Graphics ModelSim SE 6.6b, and synthesized using Xilinx ISE 11.5. Xilinx Core Generator that comes with the ISE was used to generate some of the modules in the FPGA. All Xilinx ISE products can be downloaded for free from their website. ModelSim SE is a full Verilog and VHDL simulator version. A free ModelSim Xilinx edition can be downloaded from the Xilinx website as well. Main method of programming an FPGA was through a short USB cable using ComBlock Control Center. For faster programming of an FPGA a JTAG programmer can be used. JTAG programmer can be purchased or built relatively cheaply using parallel port. Even though JTAG programming is quicker, FPGA does not retain bit stream after power is turned off, and therefore JTAG should be used for testing only.

C code development for DSP was done using Texas Instrument Code Composer Studio version 3.3. Code Composer Studio came for free with the DSP board, but usually needs to be purchased. Texas Instrument recently released a new version of Code Composer Studio based on the Eclipse IDE that can be downloaded for free from their website. Additional libraries and packages were used from Texas Instruments that are described in more detail in 4.1.2.

For graphical user interface programming Microsoft Visual Studio 2010 was used, but a smaller and free version of Microsoft Visual Basic Express can be used instead.

All ComBlock blocks can be configured and monitored through the ComBlock Control Center that can be downloaded for free on their website. If the user wishes to write their own configuration software ComBlock API reference can be used.

DAC output measurements were done using the Tektronix TDS 420A four channel digital oscilloscope and spectrum output was captured using Rohde & Schwarz FSU50 Spectrum Analyzer.

1.6 Theory of Digital Communications

Even though constructed software defined radio transceiver can be programmed to do analog modulations such as variations of AM, and FM, it was designed with digital modulations in

mind. Digital modulation has an advantage over analog modulation in couple areas. Digital modulation has a predetermined number of possible symbols, not all real numbers. As the base of any data is bits, operations such as encryption, forward error correction, and other can be performed easily. For this project it was also important to measure performance change with the use of antenna. Even though analog modulations have figure of merit, digital modulations are more easily evaluated. Digital modulations however, require correct bit timing and correct phase orientation, which can be disadvantageous. Overall, digital modulations are used in more and more areas of wireless communications today.

1.6.1 Modulation

Modulations is defined according to [3] as the "process by which some characteristic of a carrier wave is varied in accordance with an information-bearing signal." In order to modulate a signal, a carrier wave which is usually a sinusoid at the RF frequency, changes amplitude, phase, frequency, or combination of any three of them. The information signal, located at DC (zero-frequency) in frequency spectrum, is called a *baseband* signal. When the information signal is modulated onto the carrier it becomes *bandpass* signal. *Bandpass* signal is also any signal which is not located at DC in frequency spectrum.

Bandpass signal after modulation can be represented in three forms. Phase and amplitude form is

$$s(t) = A(t) * \cos\big(2\pi f_c t + \theta(t)\big) \tag{1.1}$$

In equation (1.1) $A(t)$ is the real-valued amplitude that changes with time, f_c is the carrier frequency, and $\theta(t)$ is the real-valued phase that changes with time due to information. This representation is an easy way to see what electromagnetic wave "looks like." In-phase and Quadrature form of the signal is

$$s(t) = x(t) * \cos(2\pi f_c t) - y(t) * \sin(2\pi f_c t) \tag{1.2}$$

In this form there is no phase term, as the sum of cosine and sine carriers creates different phase. $x(t)$ and $y(t)$ are real-value baseband signals. The final signal representation is preferred for mathematical calculations and is complex envelope representation:

$$s(t) = Re\{g(t)e^{2\pi f_c t}\} \tag{1.3}$$

with $g(t)$ being a complex envelope of the information carrying signal. All three forms of the expressions for bandpass signal can be converted to the other forms.

Expression (1.1) can be converted to the modulator structure shown in Figure 1.7.

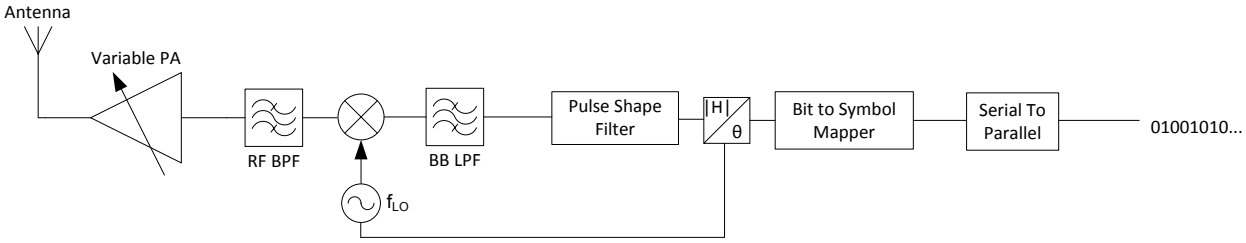

Figure 1.7: Direct modulator/up-converter

A stream of bits is mapped to a changing amplitude and phase or frequency. After that magnitude of the symbol is passed through the pulse shape filter, while phase changes local oscillator phase. Before being amplified, signal is filtered to remove any unwanted spurious emissions and mixing products. The modulator is direct, because baseband signal is directly mapped to RF carrier. As it can be seen in Figure 1.7 this type of modulator might not be easy to build, as phase will be hard to change instantaneously on the local oscillator. It is more convenient to modulate signals using modulator based on expression (1.2), which is shown in Figure 1.8. This type of modulator is called direct quadrature modulator. It still converts baseband signal directly to RF, but it does so using in-phase and quadrature components.

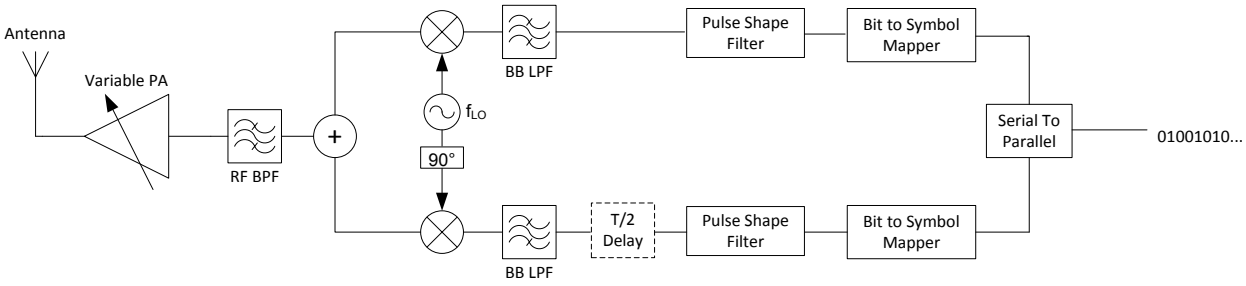

Figure 1.8: Quadrature direct modulator/up-converter

In-phase and quadrature components are real and imaginary parts of the complex envelope $g(t)$ from expression (1.3). Each part is filtered using pulse shape filter, and for some modulations, quadrature branch is delayed by half a symbol. After that in-phase branch is multiplied by cosine, and quadrature branch is multiplied by sine. Another form of the modulator is the quadrature heterodyne modulator, which is shown in Figure 1.9

Sometimes the carrier frequency is too high to do a direct conversion of baseband signal to the RF, and an additional intermediate stage is introduced. Another reason heterodyning is performed is when modulation frequency is not constant. Then baseband signal is up-converted to intermediate frequency first, where a fixed filter is present, and then up-converted once again to tunable RF frequency, but RF filter is no longer there.

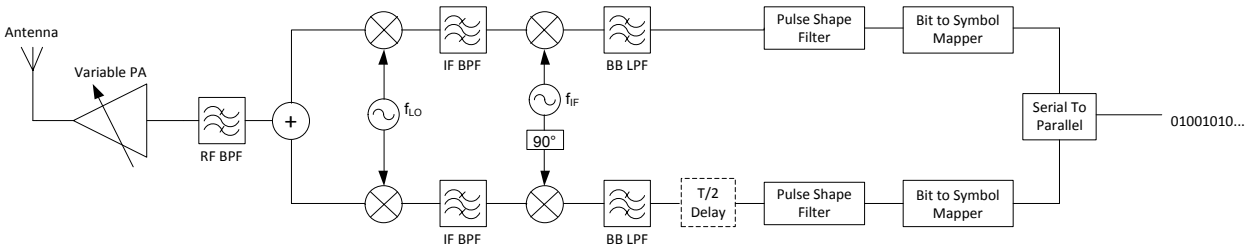

Figure 1.9: Quadrature heterodyne modulator/up-converter

Digital modulations can be broken into *linear* modulations, and *non-linear* modulations. Modulations where amplitude and phase carries information, such as PSK, PAM, ASK, QAM are linear [2]. Modulations that modulate frequency, such as FSK, MSK, are non-linear. A reader can reference [2] to read detailed description of linear and non-linear modulations. Linear modulation PSK, stands for the Phase Shift Keying. PSK modulation carries information using phase of the carrier sinusoid. PSK signals can be represented in the form:

$$s(t) = \cos\left(2\pi f_c t + \frac{2\pi N(t)}{M}\right) \qquad N = 0 \dots M-1 \qquad M \equiv number\ of\ levels$$

Or using in-phase and quadrature form:

$$s(t) = x(t)\cos(2\pi f_c t) - y(t)\sin(2\pi f_c t) \qquad x(t)^2 + y(t)^2 = 1$$

$x(t)$ and $y(t)$ need to make equally spaced symbols with energy equal to 1. For 8-PSK modulation, for example, the following pairs are used:

$$\{x(t), y(t)\} = \left\{(1,0);\ \left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right);\ (0,1);\ \left(\frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right);(-1,0);\left(\frac{-1}{\sqrt{2}}, \frac{-1}{\sqrt{2}}\right);(0,-1);\ \left(\frac{1}{\sqrt{2}}, \frac{-1}{\sqrt{2}}\right)\right\}.$$

As it can be seen from expression, sinusoidal carrier only changes phase, which means unless pulse shaping is used, sinusoid always has the same amplitude. This is an important feature for the non-linear power amplifiers on the transmit side. Non-linear power amplifiers are more efficient, but distort amplitude. Therefore for non-amplitude dependent modulations non-linear amplifier can be used without introducing errors to modulation.

4-PSK modulation is referred to as QPSK (Quadrature Phase Shift Keying), and 4-PSK modulation, where quadrature channel is shifted by half a symbol is called O-QPSK (Offset QPSK).

Linear modulation ASK stands for Amplitude Shift Keying, and signal is represented in the form:

$$s(t) = \mathrm{A(t)} * \cos(2\pi f_c t) \qquad A = [0, \dots M-1]$$

As it can be seen, there is no quadrature channel, and there is a DC offset due to use of unipolar signaling. ASK modulation does not have a symbol energy equal to 1, which means linear amplifiers are preferred.

QAM is the Quadrature Amplitude Modulation. Information is modulated using both, amplitude and the phase. Signal is easier to represent in the quadrature form:

$$s(t) = x(t)\cos(2\pi f_c t) - y(t)\sin(2\pi f_c t) \qquad x(t) \ni \{\pm 1, \pm 3, \dots\} \quad y(t) \ni \{\pm 1, \pm 3\}$$

Symbol energy is no longer equal to 1, like in PSK modulation. Most of the QAM modulations are square modulations, such that if constellation is rotated 90° it will look the same. Because QAM modulation also does not have symbol energy equal to 1 on average, linear amplifier is preferred.

Although [2] calls PAM, Pulse Amplitude Modulation an ASK modulation, it is defined in this project differently. PAM modulation just like ASK modulation is located in in-phase channel only, but the baseband pulses used are bipolar, which means there is no DC offset, and phase changes between 0° and 180°

$$s(t) = A(t) * \cos(2\pi f_c t) \qquad A(t) \ni \{\pm 1, \pm 3, \dots\}$$

Similar to QAM, PAM does not have average symbol energy of 1, and needs more linear amplifier. A PAM modulation implemented in thesis is 8-PAM modulation similar to the one used in ATSC standard.

APSK or Amplitude Phase Shift Keying modulation can be thought of as QAM modulation, except APSK modulation forms a multi-level circle constellation, instead of square constellation. In this project APSK modulation was implemented according to standard in [19] using a 9/10 code rate.

A non-linear FSK or Frequency Shift Keying modulation uses multiple carrier frequencies to transmit each symbol. Frequencies need to be spaced close enough to take little bandwidth, but at the same time be orthogonal to each other. Minimum frequency separation for coherent reception is 1/2 of a symbol rate, and for non-coherent reception 1 symbol rate. For this project a non-coherent separation was used to make sure that both methods can be used for reception. FSK signals can be generated by just switching the frequency generator between two values, but in this project signals need to be generated at baseband, which is non-intuitive. Below is the I and Q representation of how it is done:

$$s(t) = x(t)\cos(2\pi f_c t) - y(t)\sin(2\pi f_c t)$$

$$x(t) = \begin{cases} \cos(\theta + 2\pi\Delta f t)|_0^{T_s} & b = 1 \\ \cos(\theta - 2\pi\Delta f t)|_0^{T_s} & b = 0 \end{cases}$$

$$y(t) = \begin{cases} \sin(\theta + 2\pi\Delta f t)|_0^{T_s} & b = 1 \\ \sin(\theta - 2\pi\Delta f t)|_0^{T_s} & b = 0 \end{cases}$$

Using these equations each bit is mapped to a sinusoid. It is important to note, that θ is a continuous value. After each bit the value is calculated as $\theta = 2\pi\Delta f t_{T_s}$. For 4-FSK modulation, equations above can still be used, but 2 more symbols are added using additional frequency separation:

$$x(t) = \begin{cases} \cos(\theta - 2\pi\Delta f t)|_0^{T_s} & b = 0 \\ \cos(\theta + 2\pi\Delta f t)|_0^{T_s} & b = 1 \\ \cos(\theta - 2\pi 2\Delta f t)|_0^{T_s} & b = 2 \\ \cos(\theta + 2\pi 2\Delta f t)|_0^{T_s} & b = 3 \end{cases}$$

$$y(t) = \begin{cases} \sin(\theta - 2\pi\Delta f t)|_0^{T_s} & b = 0 \\ \sin(\theta + 2\pi\Delta f t)|_0^{T_s} & b = 1 \\ \sin(\theta - 2\pi 2\Delta f t)|_0^{T_s} & b = 2 \\ \sin(\theta + 2\pi 2\Delta f t)|_0^{T_s} & b = 3 \end{cases}$$

Advantage of the FSK modulation is that as number of symbols increases, it becomes more energy efficient (less errors), but at the expense of ever increasing bandwidth. It also has an average energy of 1, which means non-linear amplifier can be used, and it can be detected non-

coherently, compared to PSK. Coherent and non-coherent reception will be described in the next section.

The last non-linear modulation implemented in the project was an MSK or Minimum Shift Keying modulation. MSK modulation is usually described as an FSK modulation with a minimum spacing, but a better way to look at it is as O-QPSK modulation with half-sine pulse shaping. This means an expression for O-QPSK can be used, except a half-sine pulse is used instead of the square wave. MSK modulation, similar to FSK modulation, has an average energy of 1, which means it is a good modulation for non-linear amplifiers, but compared to FSK it needs to be received coherently, as information is stored in phase as well. Because MSK uses a half-sine pulse shape, spectrum is also better compared to a square wave.

In this project all the described modulations were implemented. Square pulses were used for all modulations except FSK and MSK modulations where the sinusoids were stored in the look-up-table. Next sections below will describe how each of the modulations is detected, and what is there theoretical performance.

1.6.2 Reception

Signal reception is much more difficult compared to the signal modulation, because lots of variables are unknown, such as exact received carrier phase and frequency. Similarly to the modulator, there are two ways to convert the RF signal to the baseband: using a direct

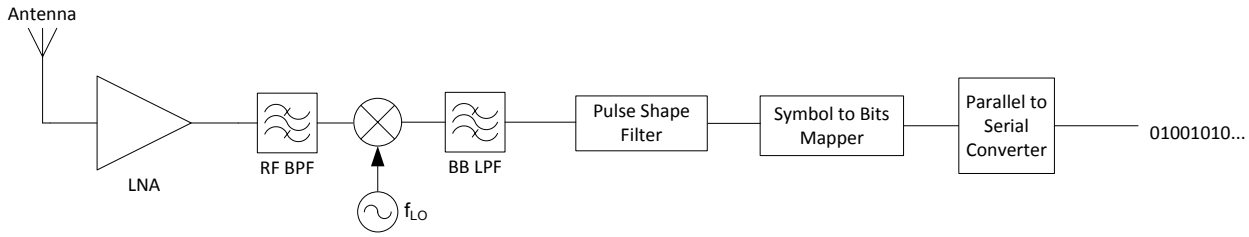

Figure 1.10: Direct conversion receiver/down-converter

down-conversion or using a heterodyne. A direct down-conversion receiver is shown in Figure 1.10. LNA is the low noise amplifier. For the best performance received signal pulse shape filter needs to be matched filter, such that:

$$g(t) = g(-t) \qquad Time\ Domain$$

$$G(f) = G^*(f) \qquad Frequency\ Domain$$

The receiver in Figure 1.10 has the disadvantage that it only works with the real signal, while it is better to work with the complex signal format. It is possible to convert real format to a complex format, but it requires additional filter in digital domain. However, this type of receiver has fewer parts, which usually means less power consumption, and lower cost. Another type of receiver is shown in Figure 1.11, which is direct quadrature down-converter. Compared to direct conversion down-converter in Figure 1.10, this receiver splits signal into in-phase and quadrature parts, which are easily converted to the complex number. An additional half-symbol delay can be added to make it possible reception of O-QPSK and MSK signals, but phase and frequency has to be synchronized. The disadvantage of this receiver structure is that if the local oscillator mixes with itself, the output contains additional DC offset at baseband. This problem can be solved with the quadrature heterodyne down-converter shown in Figure 1.12. Heterodyne down-converter also needs to be used if the local oscillator cannot cover the full RF range. The disadvantage of the heterodyne receiver is the cost.

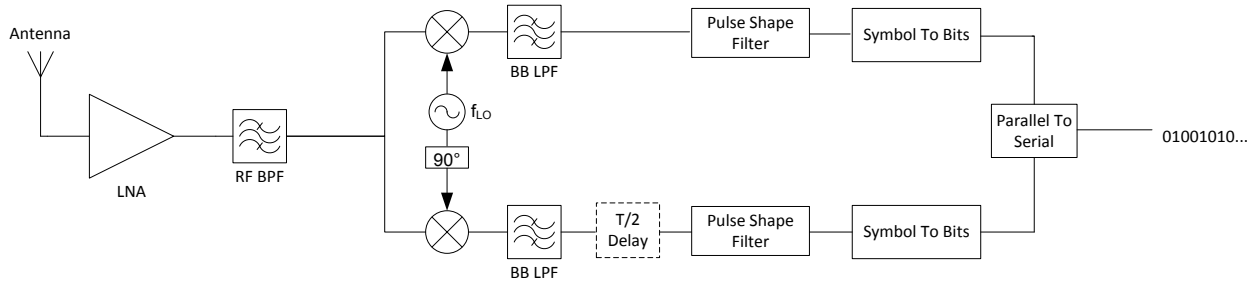

Figure 1.11: Direct quadrature receiver/down-converter

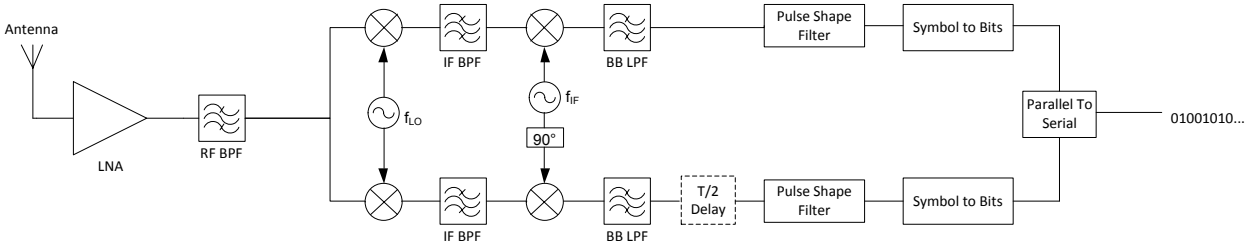

Figure 1.12: Heterodyne quadrature receiver/down-converter

It is also possible as described in section 1.2 to use subsampling. In this case most of the RF components (mixers and oscillators) can be removed, reducing the cost, but ADC needs to be able to support the bandwidth up to location of desired information. Otherwise structure in Figure 1.11 still needs to be used to down-convert signal to the required frequency range.

Whatever the structure of the receiver is used, signal needs to be demodulated *coherently* or *non-coherently*. *Coherent* demodulation means that local oscillator is completely synchronized to the carrier wave. *Non-coherent* demodulation means that local oscillator is not synchronized. Coherent demodulation is always better than non-coherent, but requires a more complex receiver with phase locked loop. All modulations carrying information in phase (PSK, QAM, MSK) need to use coherent demodulator. FSK and ASK modulations can be demodulated non-coherently, at the cost of performance drop. PSK and QAM modulations can be modified to use differential coding, and also demodulated non-coherently.

All linear modulations can be demodulated easily using receiver structure in Figure 1.11 and Figure 1.12. After the signal is converted to complex format and synchronized, each symbol is recovered based on minimum distance to the expected set of symbols. The process is called correlation. Symbol with highest correlation is most likely transmitted. MSK modulation is O-QPSK modulation with half-sine pulse shaping, which means as long as incoming signal is synchronized, it can use receiver structure in Figure 1.11 and Figure 1.12, with half-sine pulse shape filter. For FSK modulation, receiver needs frequency oscillator for each symbol transmitted, thus making it more complex. There are also some non-optimal non-coherent methods.

Realistic receiver, after down-converting, needs to correct phase and frequency offset caused by oscillator difference and also sample at correct timing. These additional functions are shown in Figure 1.13.

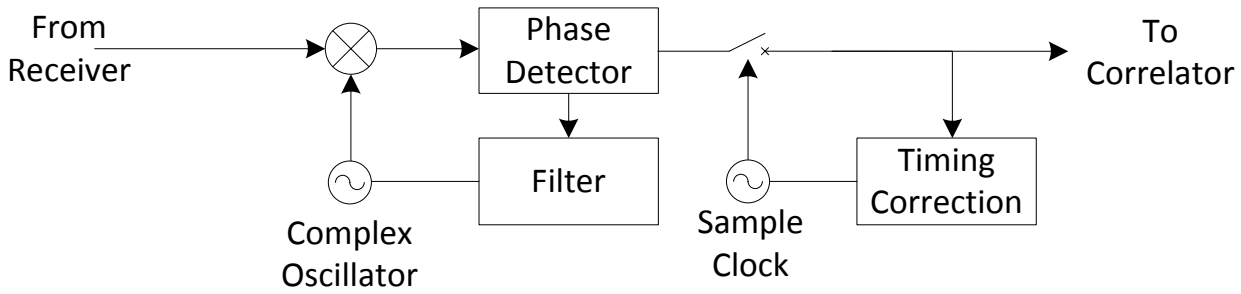

Figure 1.13: Receiver functions after down-converter

All of the circuitry in Figure 1.13 can be implemented in analog or digital domain. For a software radio these components need to be in digital domain to be adaptable to any modulation.

1.6.3 Performance Metric for Digital Modulations

Digital modulations use bit error probability as a performance metric. Given value of energy per bit, written as E_b/N_0, probability of error can be calculated for different modulations. Modulation performance usually depends on how closely spaced symbols are in the constellation. Most of the pre-derived expressions for error probability that can be found in [2] assume that received signal has an additive white Gaussian noise, symbols are received coherently, and probability of bit being either 1 or 0 is equal. It also assumes that bits in multi-dimensional modulations are gray coded. Gray coding means that closest neighbors of any particular symbol do not differ by more than one bit from each other. For example, if the symbol is mapped with bit value "100", than closest neighbors can have value "000", "101", "110", etc. Probability of error is usually expressed in terms of the Q function whose definition is shown in expression (1.4)

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty e^{-\frac{u^2}{2}} du \tag{1.4}$$

The error probability for the BPSK, MSK, O-QPSK and QPSK modulation can be calculated using following equation:

$$P_b = Q\left(\sqrt{\frac{2E_b}{N_0}}\right)$$

For any other PSK modulation probability of bit error can be found using Union Bound, and is

$$P_b \approx \frac{2}{\log_2 M} Q\left(\sqrt{\frac{2*\log_2 ME_b}{N_0}} \sin\left(\frac{\pi}{M}\right)\right)$$

where M is the number of symbols in a constellation. For BASK and BFSK modulations bit error probability is

$$P_b = Q\left(\sqrt{\frac{E_b}{N_0}}\right)$$

which means that BASK and BFSK performance is 3 dB lower than BPSK performance. For 16-QAM modulation an exact bit error probability can also be found in [2], and is equal to

$$P_b = \frac{3}{4} Q\left(\sqrt{\frac{4E_b}{5N_0}}\right) - \frac{9}{12} Q\left(\sqrt{\frac{4E_b}{5N_0}}\right)^2$$

For any other modulation expression can be found in [2], but it can also be derived using Union Bound. Explanation of how 4-ASK error performance is derived using Union Bound is provided in Appendix. 4-ASK modulation bit error probability is

$$P_b \approx \frac{3}{4} Q\left(\sqrt{\frac{2E_b}{7N_0}}\right)$$

For M-FSK modulation the bit error probability can be found using Union Bound as well, and is equal to

$$P_b \approx \frac{(M-1)}{2} Q\left(\sqrt{\frac{\log_2 ME_b}{N_0}}\right)$$

M-FSK modulation improves in performance as the size of the constellation gets larger, at the expense of the bandwidth. 8-PAM modulation bit error probability can be found derived in [2] or can be derived using Union Bound as well and is equal to

$$P_b \approx \frac{7}{12} Q\left(\sqrt{\frac{2E_b}{7N_0}}\right)$$

The 16-APSK modulation that is part of the DVB-S2 standard from [19] does not have a non-coded bit error probability derived anywhere. Union Bound was used to find the bit error probability expression, but experimentally it was found that it was incorrect. Overall 16-APSK performance is close to 16-QAM performance when symbols are equally spaced apart, but in DVB-S2 standards they are optimized for Trellis coding. Therefore an experimental Table 1.1 was created which is shown below.

E_b / N_0 (dB)	Bit Error Probability
3	0.08292
4	0.06361
5	0.04621
6	0.0313
7	0.01954
8	0.01106
9	0.005476
10	0.002353
11	0.0008326
12	0.0002429

Table 1.1: 16-APSK experimental bit error probability

2. Transmitter Hardware

Transmitter hardware consists of three major blocks: digital signal processor (DSP), digital-to-analog converter (DAC), and an up-converter modulator. DSP takes an array of data bits from either a BER sequence or from the data user wants to transmit and then maps each bit, or a group of bits onto the constellation point. After digital data is mapped to symbols, DAC converts symbols to analog voltages that are multiplied by a carrier in the modulator block. The main challenges were mapping symbols correctly and keeping up with the transmission data rate. Due to the non-standard interface, and the lack of COTS solutions, another challenge was creation of an interface between the DSP and the DAC that will be described later in this chapter. The overall block diagram of the transmitter hardware is shown in Figure 2.1

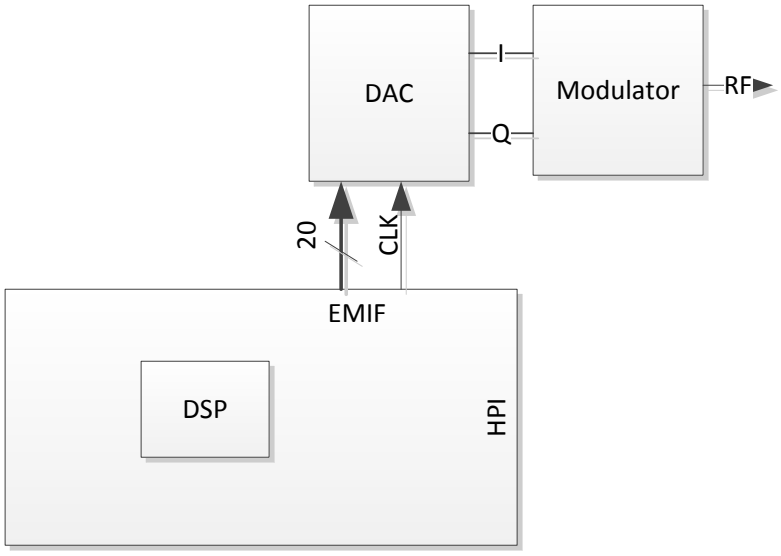

Figure 2.1: Block diagram of the transmitter hardware connections

2.1 Digital to Analog Converter

Digital-to-analog converter (DAC) converts a bit stream into a variable voltage or current. There are different types of DACs, which use different methods of converting digital signal to the analog signal. This is outside of the scope of this paper, as the selection of the DAC was solely based on the ease of integration of the DAC with the DSP board. The DAC that was chosen for this project is a rapid prototyping module from ComBlock. The module is COM-2001 whose description can be found in [7]. COM-2001 uses Analog Devices AD9760 DACs, with a

maximum sampling rate of 125 MSamples/s. Two DACs on the COM-2001 board use 10-bit parallel data input. DACs output is passed through a 6-pole Butterworth filter with the 3 dB cutoff frequency of 13 MHz. This means that although the output of the DAC can have a sampling rate of 125 MSamp/sec, the maximum data bandwidth is around 10 MHz. Output of the DAC is in the unsigned format with the inverted logic, with the following mapping:

$$0 = +0.75V$$

$$1023 = -0.75V$$

Although the manual [7] specifies DAC output to be $1V_{pp}$ with 0.85V DC bias, experimentally it was found that a better expression for the DAC output is

$$V_o = 1.56 - 1.46 * \left(\frac{D_{IN}}{1023}\right) V \tag{2.1}$$

where D_{IN} is the digital input word. The maximum output using expression (2.1) is $1.5V_{pp}$ The results can be confirmed with images taken at the DAC output in section 5.2

The DAC module is powered by a single +5 V DC power source, and provides a 40-pin 2 mm pin spacing connector. The analog output is through 2 SMA connectors. The 40-pin connector provides digital grounds that can be tied to the DSP digital ground to have the same voltage reference. COM-2001 module can work with +3.3V or +5V digital inputs.

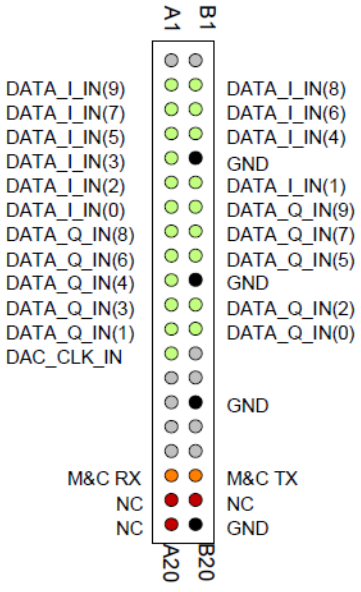

Figure 2.2: Input connector of the COM-2001 DAC [7] [Fair Use]

In addition to the digital inputs for the data, there are also pins for the serial communication between other ComBlock modules. Figure 2.2 shows all the inputs and outputs of the 40-pin connector to the DAC.

2.2 DAC to DSP interface

2.2.1 Interface wiring

Because DSP and COM-2001 modules are from different companies, they do not share an easy interface connector. Most ComBlock modules have a 40-pin 2 mm pin spacing connector. For the ease of creating an interface cable between DSP and the DAC, and between DSP and the receiver module, it was beneficial to use a pre-made cable, or a crimp-on connector and a ribbon cable. Even though it is fairly easy to find a crimp on connector for the 40-pin 2 mm spacing, the ribbon cables are very expensive. In addition, all interfaces on the DSP are output through the 80-pin 0.050 x 0.050 inches low profile connector from Samtec. At the time of creating the interface board, there was no crimp on mating connector for DSP available. The only available connectors were surface mount, and a through-hole. This meant that a small interface PCB would need to be created, which would take a lot of time, and money. In order to save time and money, interface needed to be created using some kind of commercial of the shelf parts (COTS). A very common cable used inside of the PCs is the Parallel ATA/IDE cable, which is used to connect older hard drives and CD-ROM drivers to the motherboard. The IDE cable can handle speeds of up to 100 MHz, which fit the project. IDE cables have 40-pin 0.1 inch spacing connector, while ComBlocks as mentioned before have 2 mm spacing connector. However, laptop PCs that use IDE bus, use 44-pin 2 mm spacing connector, and there are cables and adapters readily available that transform 40-pin 0.1 inches to 44-pin 2 mm spacing connector. These cables cost under $10, and have 4 extra pins on the side for hard drive power. Those pins were removed as they were not needed.

In order to make a connection between a ribbon cable and a DSP, a wiring board was purchased from the same company that made the DSP board. The board is presented in Figure 2.3 below. The board had mating connectors to the DSP's interface. The board cost was $100. Each of the wiring board's connectors had a copper pad, and a through-hole connection available. This made possible soldering of ribbon cable to the wiring board. In addition to the expansion connectors, wiring board also contains pads for the serial and parallel port output. If

the user wants to, they can easily implement communication using serial port though the McBSP on the same board.

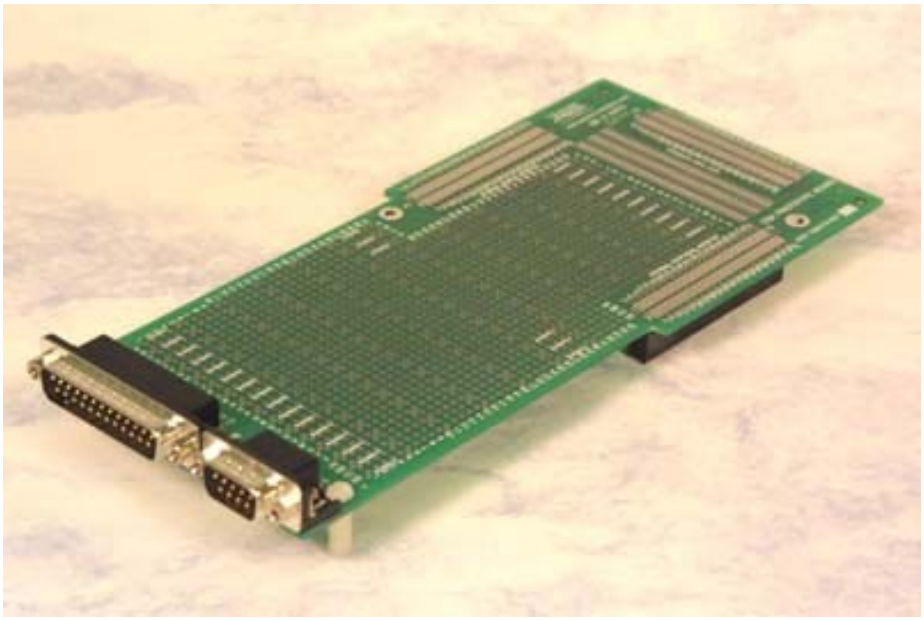

Figure 2.3: Wiring and prototyping board for the TMS320C6455 DSK [34] [Fair Use]

Before the signals can travel from the DSP to the board, a small modification was needed. In order for the DSP board to transmit to the expansion connectors, pin 75 on the peripheral expansion connector J3 described in [5] and a ground pin needed to be soldered together. Pin 75 indicated to the DSP board that a daughter-card is present.

2.2.2 Data Exchange

TMS320C6455 DSK provides 3 external interface connectors. External memory interface (EMIF), Host Port Interface (HPI), also shared by PCI, and other peripheral outputs connector such as voltages for the daughter board, timers, and McBSP. COM-2001 DAC accepts data in parallel format and requires outside clock to latch data on the clock's rising edge. Both, HPI and EMIF interfaces are parallel data interfaces [13, 16]. EMIF port is made to access external synchronous or asynchronous memory, and DSP has full control over it. HPI port, on the other hand is designed to be accessed by an external system, and DSP has almost no control over it. Given these conditions, it is beneficial to use EMIF interface to communicate data with DAC. Pinout of the EMIF expansion connector on the DSK is provided in Table 2.1, and pinout of the DAC is shown in Figure 2.2.

Pin	Signal	I/O	Description	Pin	Signal	I/O	Description
1	5V	Vcc	5V voltage supply pin	2	5V	Vcc	5V voltage supply pin
3	AEA19	O	EMIF address pin 21	4	AEA18	O	EMIF address pin 20
5	AEA17	O	EMIF address pin 19	6	AEA16	O	EMIF address pin 18
7	AEA15	O	EMIF address pin 17	8	AEA14	O	EMIF address pin 16
9	AEA13	O	EMIF address pin 15	10	AEA12	O	EMIF address pin 14
11	GND	Vss	System ground	12	GND	Vss	System ground
13	AEA11	O	EMIF address pin 13	14	AEA10	O	EMIF address pin 12
15	AEA9	O	EMIF address pin 11	16	AEA8	O	EMIF address pin 10
17	AEA7	O	EMIF address pin 9	18	AEA6	O	EMIF address pin 8
19	AEA5	O	EMIF address pin 7	20	AEA4	O	EMIF address pin 6
21	5V	Vcc	5V voltage supply pin	22	5V	Vcc	5V voltage supply pin
23	AEA3	O	EMIF address pin 5	24	AEA2	O	EMIF address pin 4
25	AEA1	O	EMIF address pin 3	26	AEA0	O	EMIF address pin 2
27	ABE3#	O	EMIF byte enable 3	28	ABE2#	O	EMIF byte enable 2
29	ABE1#	O	EMIF byte enable 1	30	ABE0#	O	EMIF byte enable 0
31	GND	Vss	System ground	32	GND	Vss	System ground
33	AED31	I/O	EMIF data pin 31	34	AED30	I/O	EMIF data pin 30
35	AED29	I/O	EMIF data pin 29	36	AED28	I/O	EMIF data pin 28
37	AED27	I/O	EMIF data pin 27	38	AED26	I/O	EMIF data pin 26
39	AED25	I/O	EMIF data pin 25	40	AED24	I/O	EMIF data pin 24
41	3.3V	Vcc	3.3V voltage supply pin	42	3.3V	Vcc	3.3V voltage supply pin
43	AED23	I/O	EMIF data pin 23	44	AED22	I/O	EMIF data pin 22
45	AED21	I/O	EMIF data pin 21	46	AED20	I/O	EMIF data pin 20
47	AED19	I/O	EMIF data pin 19	48	AED18	I/O	EMIF data pin 18
49	AED17	I/O	EMIF data pin 17	50	AED16	I/O	EMIF data pin 16
51	GND	Vss	System ground	52	GND	Vss	System ground
53	AED15	I/O	EMIF data pin 15	54	AED14	I/O	EMIF data pin 14
55	AED13	I/O	EMIF data pin 13	56	AED12	I/O	EMIF data pin 12
57	AED11	I/O	EMIF data pin 11	58	AED10	I/O	EMIF data pin 10
59	AED9	I/O	EMIF data pin 9	60	AED8	I/O	EMIF data pin 8
61	GND	Vss	System ground	62	GND	Vss	System ground
63	AED7	I/O	EMIF data pin 7	64	AED6	I/O	EMIF data pin 6
65	AED5	I/O	EMIF data pin 5	66	AED4	I/O	EMIF data pin 4
67	AED3	I/O	EMIF data pin 3	68	AED2	I/O	EMIF data pin 2
69	AED1	I/O	EMIF data pin 1	70	AED0	I/O	EMIF data pin 0
71	GND	Vss	System ground	72	GND	Vss	System ground
73	AARE#	O	EMIF async read enable	74	AAWE#	O	EMIF async write enable
75	AAOE#	O	EMIF async output enable	76	AARDY	I	EMIF asynchronous ready
77	ACE3#	O	Chip enable 3	78	ACE2#	O	Chip enable 2
79	GND	Vss	System ground	80	GND	Vss	System ground

Table 2.1: External memory (EMIF) expansion connector pinout [5] [Fair Use]

Only data needs to be written to the DAC, therefore all the address pins available on the EMIF interface are irrelevant. To make it easier writing integers for the modulating signal, it is beneficial to make I channel be in the upper 16 bits, and Q channel be in the lower 16 bits.

However, data width is only 10 bits for both I and Q channels on the DAC. Therefore there is no need to use all 16 bits. Figure 2.4, shows how DAC bits inside of the DSP memory are mapped. This mapping implies that based on Table 2.1 pins AED0 – AED9 are used, and pins AED16 – AED25 are used. Pins AED10 - AED15 and AED26 – AED31 are left unconnected.

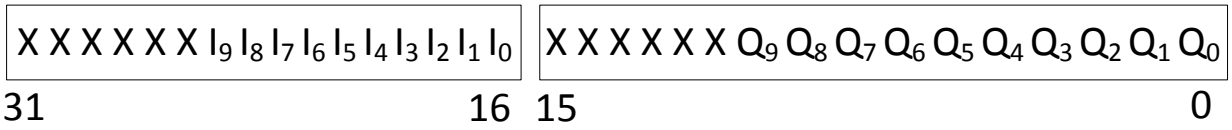

Figure 2.4: Bit mapping of I and Q symbols in 32-bit word

To make sure that signals traveling over the line experience the least amount of noise, all ground pins on the DAC modules are connected to the ground pins on the DSP connector. Finally, a clock needs to be provided for the DAC. Clock can be generated using on-board timer, or using external memory interface. Using on-board timer, a clock of up to 100 MHz can be generated, however, memory interface is not synchronized in any way to the timer, and some kind of synchronization would needs to be implemented. A FIFO implemented using a discrete IC or a programmable logic, such as FPGA, can be used to implement synchronization. This method, however, requires additional components and increases overall project cost. Another method of providing clock to the DAC is using asynchronous memory $\overline{AAWE}$ output pin, which is "Write Enable" pin. The pin is at the position 74 on the EMIF external connector as shown in Table 2.1 Maximum clock rate that can be provided by this pin is 30 MHz. Although this clock is much slower than the clock that can be generated using timer, it is synchronous with the memory accesses, and because of that does not require any additional components to synchronize writes to the DAC. Table 2.2 summarizes all the connections and corresponding pin numbers on the DAC module and on the DSP that are made.

DSP Pins	DAC Pins	Pin Designation
11	B5	GND-GND
12	B10	GND-GND
31	B15	GND-GND
32	B20	GND-GND
50	A7	AED16 – DATA_I_IN(0)
49	B6	AED17 – DATA_I_IN(1)
48	A6	AED18 – DATA_I_IN(2)
47	A5	AED19 – DATA_I_IN(3)
46	B4	AED20 – DATA_I_IN(4)

45	A4	AED21 – DATA_I_IN(5)
44	B3	AED22 – DATA_I_IN(6)
43	A3	AED23 – DATA_I_IN(7)
40	B2	AED24 – DATA_I_IN(8)
39	A2	AED25 – DATA_I_IN(9)
70	B12	AED0 – DATA_Q_IN(0)
69	A12	AED1 – DATA_Q_IN(1)
68	B11	AED2 – DATA_Q_IN(2)
67	A11	AED3 – DATA_Q_IN(3)
66	A10	AED4 – DATA_Q_IN(4)
65	B9	AED5 – DATA_Q_IN(5)
64	A9	AED6 – DATA_Q_IN(6)
63	B8	AED7 – DATA_Q_IN(7)
60	A8	AED8 – DATA_Q_IN(8)
59	B7	AED9 – DATA_Q_IN(9)
74	A13	AAWE – DAC_CLK_IN

Note: A1-A20 = Bottom Row
B1-A20 = Upper Row

Table 2.2: Pin numbers and connections between DAC and the EMIF port on DSP

2.2.3 EMIF port configuration

In order to configure the DAC clock, the memory interface needs to be configured prior to the main program execution. The $\overline{AAWE}$ toggle rate, which is DAC clock rate are controlled by the following asynchronous memory timing write parameters: SETUP, STROBE, HOLD, and TURN-AROUND (TA). These parameters, except TA, are shown in the asynchronous memory write cycle diagram in Figure 2.5 found in [13]. All of the timings during the write cycle are based off EMIF clock ECLKOUT, which is either provided externally, or from the internal system clock SYSCLK4 [23]. Minimum values for the SETUP, STROBE, and HOLD are 1 ECLKOUT cycle. TURN-AROUND minimum value is 2 ECLKOUT cycles. Therefore maximum $\overline{AAWE}$ toggle rate that can be achieved is ECLKOUT/5. The way TMS320C6455 DSK is configured, ECLKOUT = 96 MHz. Re-soldering some components on the DSK, ECLKOUT source can be changed from externally provided 96 MHz clock to internally provided SYSCLK4 [5]. However maximum clock rate for SYSCLK4 is 166 MHz, which makes maximum $\overline{AAWE}$ maximum toggle, rate 33.2 MHz. Given that default value for SYSCLK4 is

DSP clock divided by 8 [23], SYSCLK4 = 1200 MHz / 8 = 150 MHz. This maximum value is what gives the calculated maximum DAC clock of 30 MHz, mentioned in the previous section.

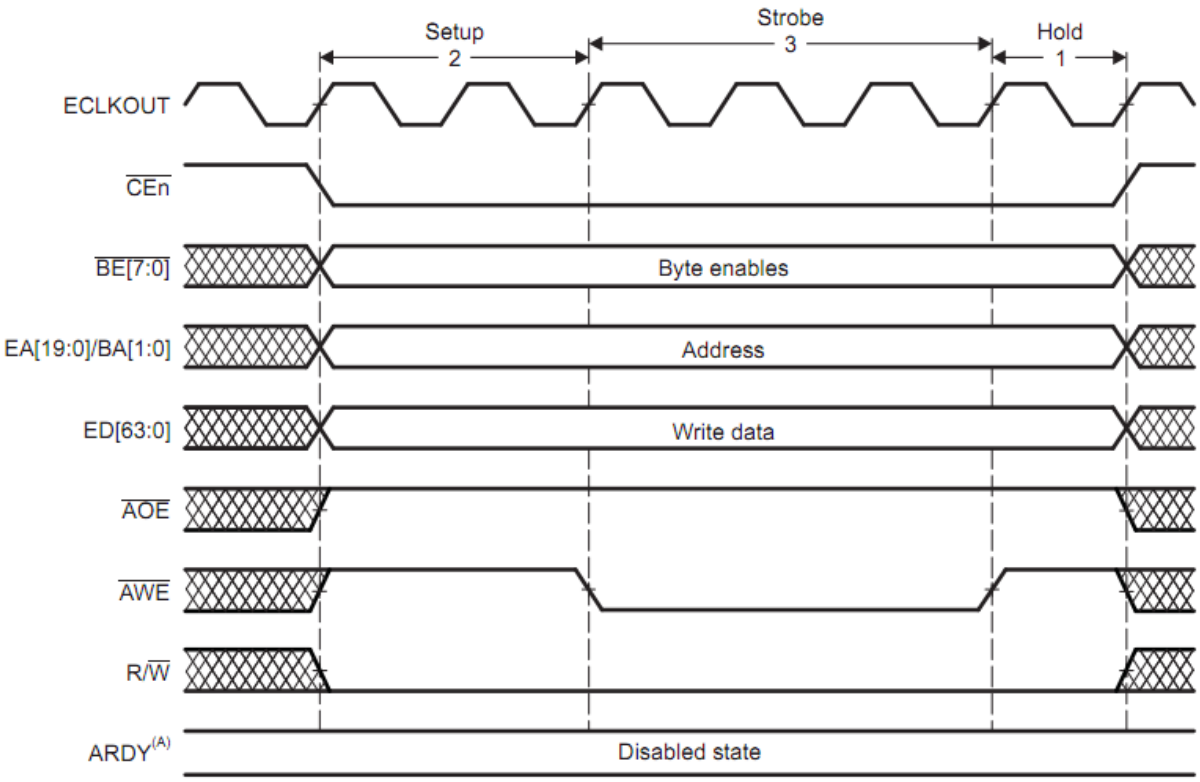

Figure 2.5: Timing Diagram for the EMIF interface in the writing mode [13] [Fair Use]

Figure 2.5 shows more signals than are presented in Table 2.2, confusing the reader, but the EMIF connector provided on the TMS320C6455 DSK provides inputs and outputs only for signals in Table 2.1. An ECLKOUT output is available on the other peripheral connector J3.

Even though it is not necessary, clock signal for the DAC was made with 50% duty cycle. Following Figure 2.5 it can be seen that in order to make DAC clock have 50% duty cycle, STROBE needs to be as long as SETUP, HOLD, and TA together. Given those parameters, SETUP = 1, HOLD = 1, TA = 2, and STROBE = 4. With these settings DAC clock is set at ECLKOUT/8 rate, which is equal to 12 MHz. Given a clock rate of 12 MHz, a maximum symbol rate of 6 MHz, can be achieved from the output of the DAC. However, this rate can only be achieved using square pulses, with 2 samples per symbol. If MSK, FSK, or any other modulation with pulse shaping is to be implemented, symbol rate needs to be reduced to 2 MHz or less to avoid having an incorrect output spectrum.

2.3 Modulator

In order to be able to transmit signal at the RF frequency, baseband signal that comes from DAC needs to pass through modulator. Modulator types and structures are described in section 1.6.1. For this project ComBlock COM-4002D direct up-conversion modulator was used description of which can be found in [9]. Modulator converts baseband signal to the frequencies between 950 and 1450 MHz. Modulator also contains output power measurement option. The output power can be read from ComBlock Control Center as hexadecimal number. After converting hexadecimal number to decimal, the following expression gives approximate output power:

$$P_o = -\frac{1}{26.02} * D_{IN}(dec) + 9.316\ dBm$$

Modulator used in this project has a structure shown in Figure 2.6. In-phase and quadrature channels on the input of the modulator are in unipolar format. According to the expression (2.1) the value is between 0.08V and 1.56V. Modulator converts a unipolar signal to bipolar signal by subtracting an average value from it, based on schematic [22]. Average value is subtracted from input, by a means of a differential amplifier as shown in Figure 2.6 and can be found in [22].

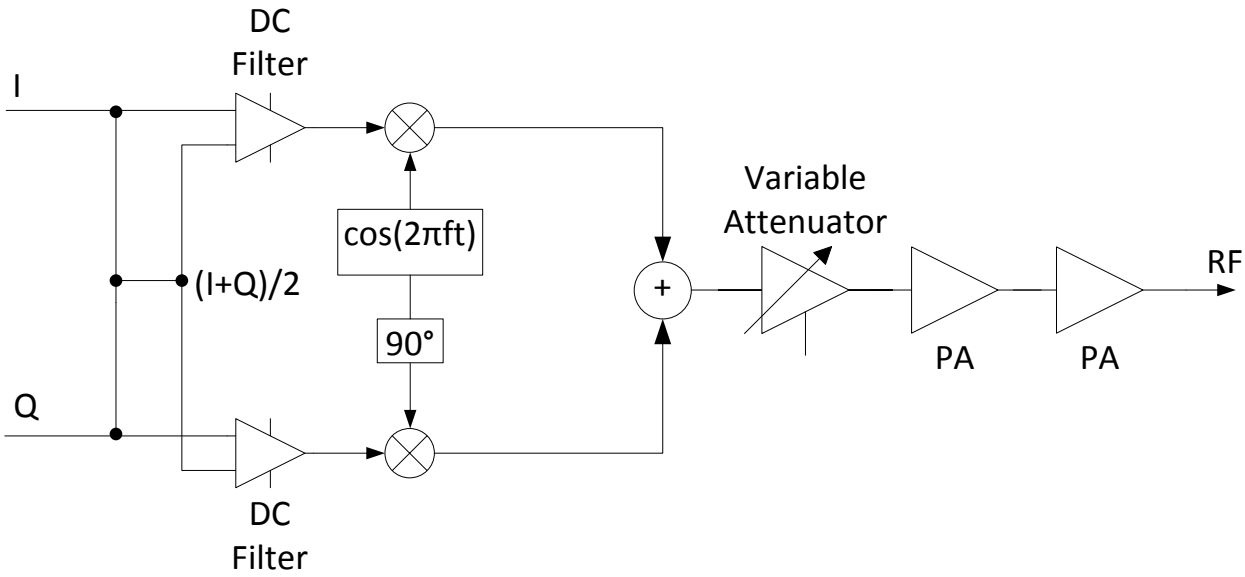

Figure 2.6: Modulator structure used in this project

This type of DC offset removal works well for the inherently bipolar modulated signals such as PSK, QAM, MSK, and FSK. When the signal is modulated using unipolar scheme such as

ASK, the received signal has a wrong DC offset if it is not modified. For example, theoretically mapping of symbols for the BASK should be as follows:

$$(I, Q): D_{IN} = (512, 512) => 0$$

$$(I, Q): D_{IN} = (0, \quad 512) => 1$$

Experimentally it was found that in order to get the correct signal output the average value at the input of the modulator needs to be 512. Because of that, the following mapping works correctly for BASK:

$$(I, Q): D_{IN} = (512, 512) => 0$$

$$(I, Q): D_{IN} = (0, \ 1023) => 1$$

Similarly, values for in-phase and quadrature channels needed to be adjusted for 4-ASK as well.

After signal is modulated, it passes through a variable attenuator, and a two stage power amplifier. Each amplifier has a gain of 14 dB, and a 1 dB compression point at +7 dBm. The scheme is different from the one shown in Figure 1.8. Using variable attenuator allows keeping amplifier always working at the same power. Controlling variable attenuator is much easier, than controlling amplification of the power amplifier, which requires changing power supply voltage.

3. Receiver Hardware

Receiver hardware is a little more complex than transmitter hardware, due to the fact that samples from the ADC cannot directly be fed into DSP. As was mentioned in the previous chapter, DSP contains two ports, and an HPI port was chosen to communicate with the ADC. Because HPI port is designed to be controlled by the outside device, original design when ADC was directly connected to an HPI port failed. Sample rate from the ADC is also too high to be processed directly without any buffering. Due to these reasons it was necessary to insert an additional device between ADC and HPI port that would take care of pre-processing signal and controlling an HPI interface. A Field Programmable Gate Array (FPGA) was chosen to do that job. An FPGA, and ADC blocks were both purchased from the ComBlock. Because they were from the same company, they easily interfaced between each other, and did not require any additional work. An FPGA to DSP interface was made using the same board as in Figure 2.3, and using the same cabling method as was described in Chapter 2.2.1. Overall block diagram of the receiver hardware is presented in Figure 3.1. The received signal is directly down-converted in the receiver block module using quadrature down-converter, filtered and sampled at 40 Msamples/s. After sampling, the signal is pre-processed by the FPGA, which buffers samples and sends them to the DSP for final processing and decoding.

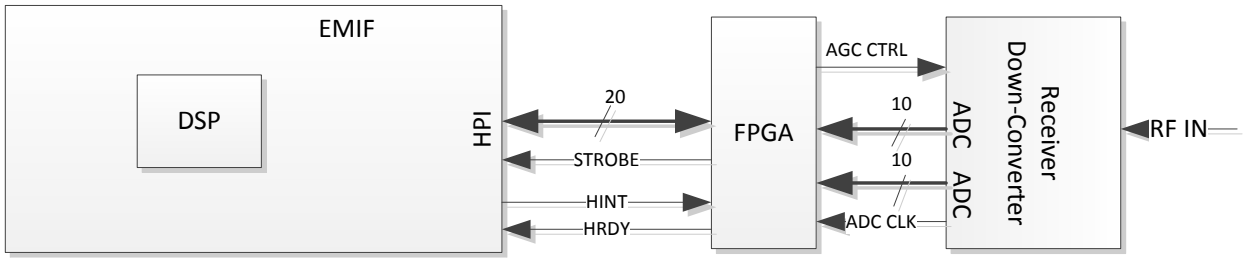

Figure 3.1: Receiver hardware block diagram

3.1 Quadrature down-converter

The first stage or the receiver is the direct quadrature down-converter, which has a structure similar to the one shown in Figure 1.11. Due to DC offset, which came from local oscillator leakage, a software based heterodyning was introduced. To achieve software heterodyning, a receiving frequency was set below the transmitting frequency, known as low-side heterodyning. Low-side heterodyning makes sure spectrum is not inverted. A ComBlock COM-3002B

quadrature down-converter was used in this project. The block diagram of it is shown in Figure 3.2 Frequency synthesizer has a minimum step size of 25 kHz, and frequency selection can be triggered using outside pin. The input frequency range is from 900 to 1575 MHz. Signal amplification is adjusted using voltage at the AGC input pin. Before being sampled by the ADC, down-converted signal is filtered using 4th order elliptic filter with 3 dB bandwidth of 20 MHz.

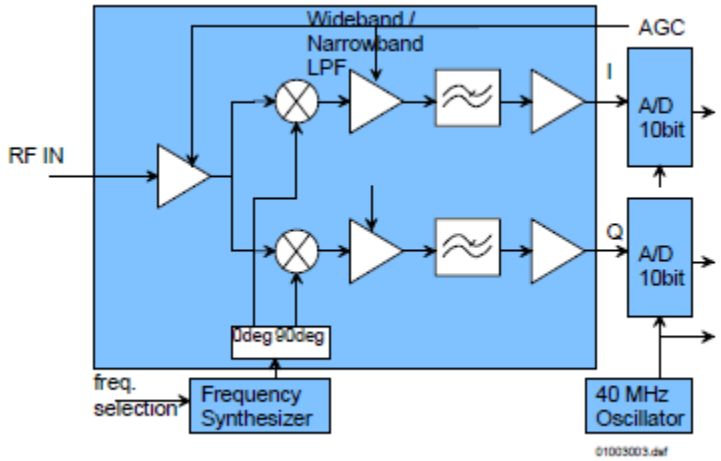

Figure 3.2: Direct down-converter used in the project [8] [Fair Use]

The output of the COM-3002B module is the 40 pin 2 mm connector which interfaces directly to the FPGA module following it. The pinout of the COM-3002B module is shown in Figure 3.3. The output clock is used to latch in ADC samples at the rising edge.

Due to the fact that receiver and transmitter modules are separated physically on the board, there was a need to connect them together such that all of them can be configured from the ComBlock Control Center, using same interface. To do this connector J6 on the COM-3002B and connector J1 on COM-4002D were connected together using a small ribbon cable, made by cutting a 44-pin 2 mm spacing connector into 4x2 pin blocks. M&C RX and M&C TX pairs were connected together. By connecting all blocks serially, it is now possible to configure all 4 of them using connection to just one of them. PLL_STOBE is also shared on connector J1 on COM-4002D and connector J6 on COM-3002B. This connection will allow synchronous frequency switching on transmitter and receiver as required for antenna testing described in the Introduction.

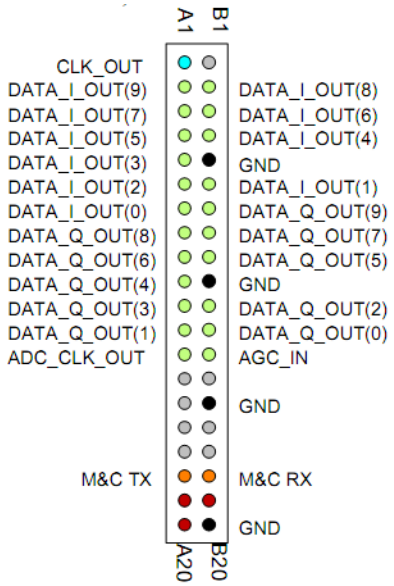

Figure 3.3: Output of the COM-3002B receiver module [8] [Fair Use]

3.2 FPGA Pre-Processing

Sampling rate of 40 MSamp/s coming from the ADC is too high to directly process it by the DSP, and additionally DSP's HPI interface is designed to have an external device for the control. In order to solve these problems, an FPGA was inserted between the receiver block, and the DSP. FPGA performs couple of the operations before the signal gets sent to the DSP for the demodulation: gain control, sample rate reduction, timing error estimation, matched filtering, and communication with the DSP. The overall internal block diagram of the FPGA is shown in Figure 3.4. Each of the blocks is explained in detail further in the chapter. In addition to the blocks shown in Figure 3.4 FPGA contains blocks that are used to communicate between the PC and the FPGA, and to capture signals inside of the FPGA for user analysis. However, these blocks are not shown as they are not part of the signal processing.

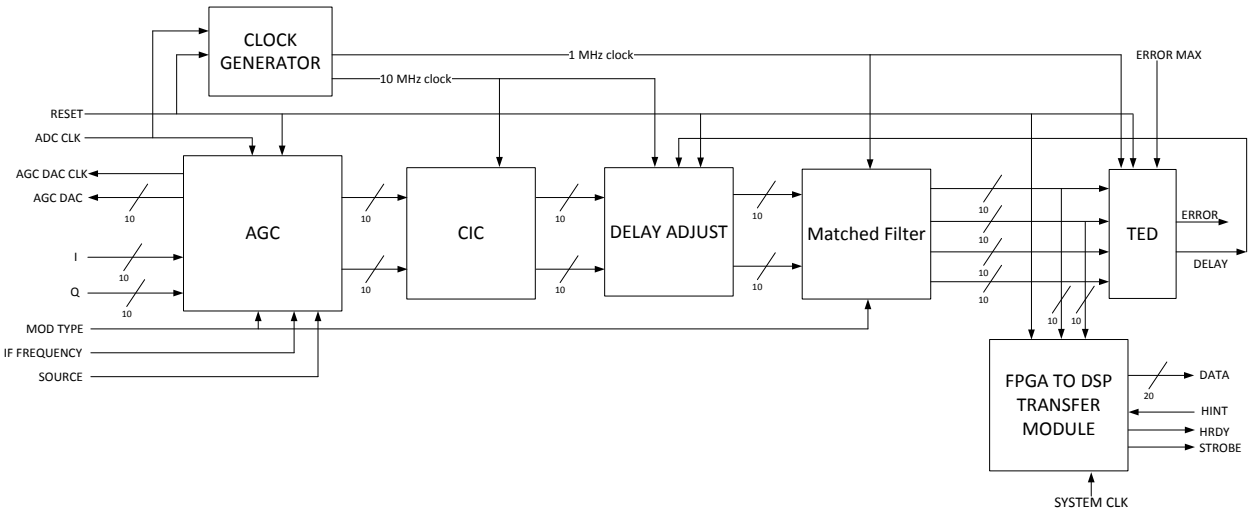

Figure 3.4: Block diagram of the signal processing modules inside FPGA

3.2.1 AGC Block

Automatic Gain Control (AGC) is the process of controlling the gain of the signal at the input of the ADCs, such that their full resolution is used. AGC block inside of the FPGA performs gain adjustments and also removes any DC offset from the input signal. The detailed block diagram of the structure of the AGC block is shown in Figure 3.5.

The quadrature branch of the input signal needs to be inverted if the signal comes directly from the ADC, or kept the same if the signal comes from the memory. The signal inside of the memory is stored in the complex envelope form:

$$s(t) = g(t)e^{j2\pi f_{IF}t} = [i(t) + j * q(t)] * e^{j2\pi f_{IF}t} \tag{3.1}$$

However, signal from the ADC comes in the form (ignoring amplitude factor):

$$r(t) = \begin{Bmatrix} i(t) * \cos(2\pi f_{IF}t) + q(t) * \sin(2\pi f_{IF}t) \\ -i(t) * \sin(2\pi f_{IF}t) + q(t) * \cos(2\pi f_{IF}t) \end{Bmatrix}$$

Converting this to the complex form (ignoring amplitude factor):

$$r(t) = i(t) + j * q(t) = [i(t) + q(t)] * e^{-2\pi f_{IF}t}$$

As it can be seen in the above expression, if this input is multiplied by the NCO sinusoid in the form:

$$v(t) = e^{-j2\pi f_{IF}t} = \cos(2\pi f_{IF}t) - j * \sin(2\pi f_{IF}t)$$

instead of moving data to the baseband, it will be moved to the 2 times the IF frequency. One solution to this is to have NCO output in the conjugate form and store signal in the memory in the same form as the input from the ADC. However in most literature signal is assumed to be in the form in equation (3.1), and therefore it was chosen to keep input in the same form.

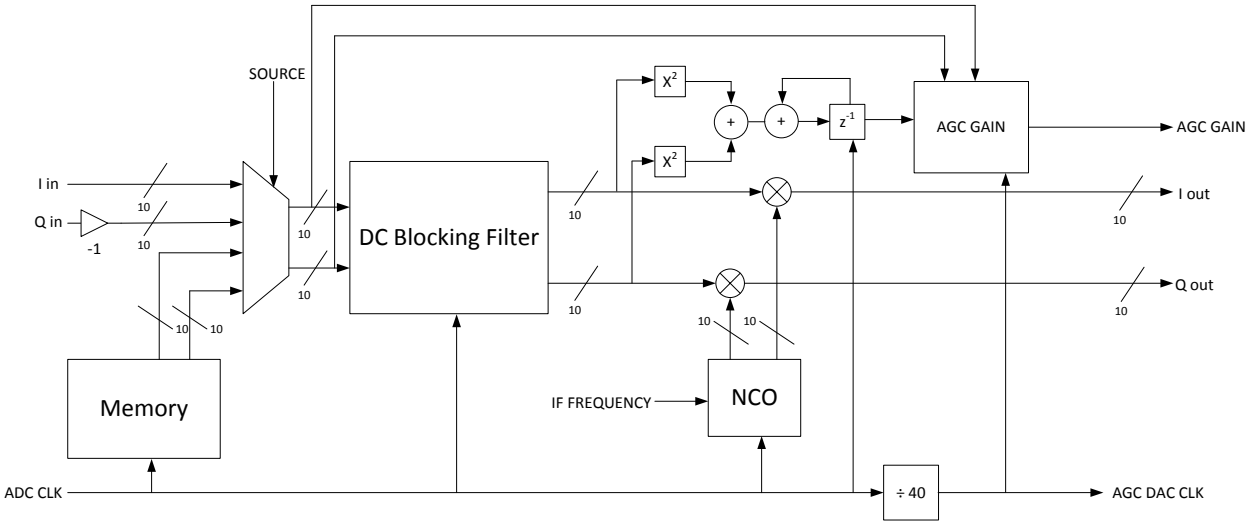

Figure 3.5: Detailed AGC block diagram

When input comes directly from the ADC there is an unknown timing offset that changes over time and degrades receiver performance. In order to do theoretic noise tests that take timing offset out of the performance results, and test algorithm in DSP, input from the memory was introduced. User can use REG1 in ComBlock Control Center, to select which input to use. Memory contains one block of 63 symbols that transmitter generates, and simulates all the properties of the input waveform. Waveform is generated in the MATLAB, and converted to the COE file that Xilinx Core Generator can use for the block memory generation. Because waveform stored in the memory needs to simulate input, MATLAB generated signal is multiplied by IF frequency, with the additional phase and frequency offset, and then shifted to be between 0 and 1023. Frequency offset needs to be as follows:

$$f_{off} = \left(\frac{1e6}{63}\right) * n$$

where n is an integer. The reason this needs to be done is because memory output is periodic, and there should be a complete number of sine cycles at the end of the waveform, otherwise there will be error jumps between successive periods. The MATLAB script to generate a waveform for the FPGA is provided in the Appendix B.

Input signal, either from the memory or directly from the receiver is represented in straight binary. This means that input values are between 0 and 1023, but the values need to be signed between -512, and +511. A simple subtraction

$$s(t) = r(t) - 512$$

only works when signal does not have any DC offset, but true received signal has unknown DC offset due to oscillator leakage, which gets larger with higher gain of the input signal. To avoid this problem, a digital IF frequency is introduced, and the signal is passed through DC blocking filter. DC filter response is shown in Figure 3.6.

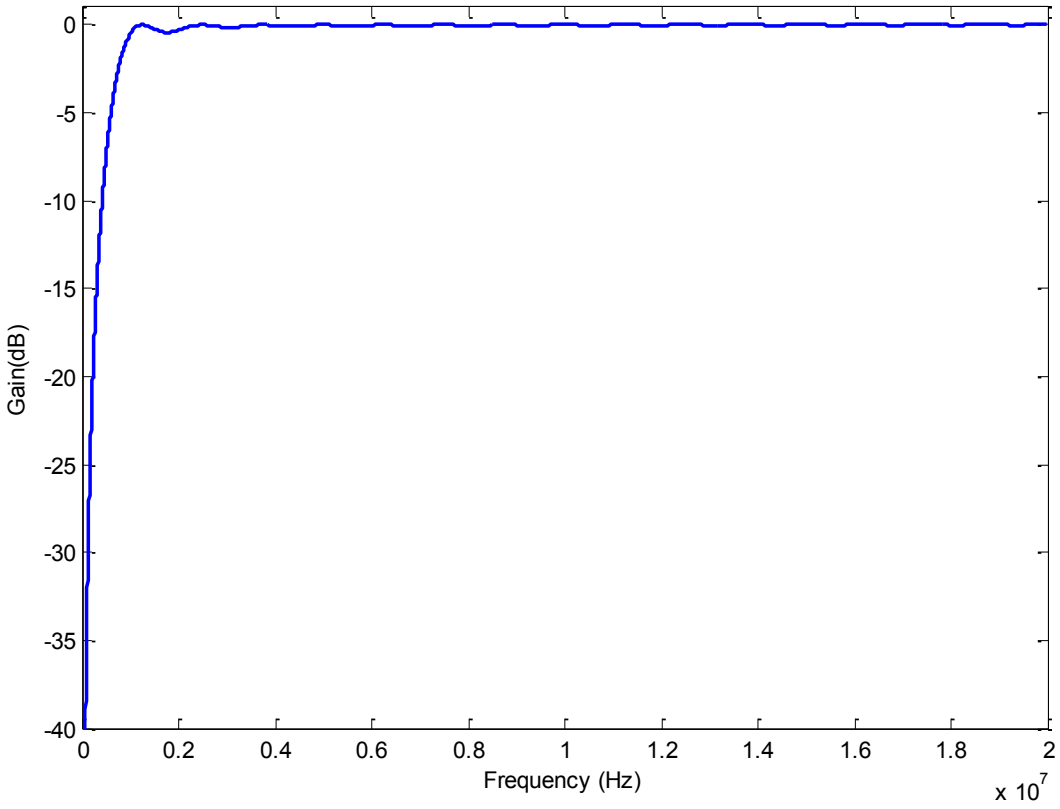

Figure 3.6: DC blocking filter response

DC blocking filter is implemented using a moving average filter and a delay line. Block diagram of the filter is shown in Figure 3.7. Full filter description is available in [4]. Filter phase response needs to be linear, just as with the FIR filter, because square input pulses will get corrupted. DC filter cutoff frequency depends on the filter delay amount. It is possible to make a DC filter with the very narrow stop-band, however, amount of resources required is so large, that moving signal to IF frequency and making filter more relaxed is a better solution.

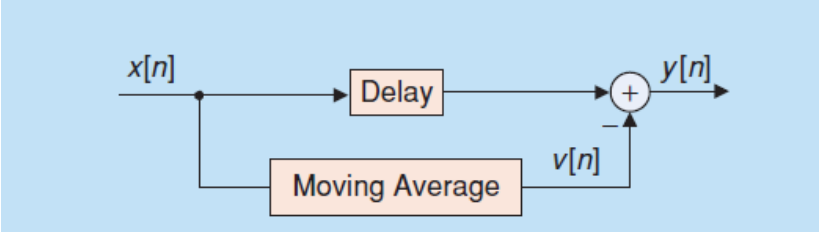

Figure 3.7: DC blocking filter block diagram [4] [Fair Use]

DC blocking filter's pass-band ripple is -0.42 dB, which can be reduced to -0.04 dB if desired by placing two Moving Average filters in series, and doubling the delay line length. For this project the length of the delay for the moving average filter was chosen to be 32, which gives it a -3 dB cutoff frequency of about 71 kHz. Given these parameters IF frequency needs to be chosen such that most of the power is located in the flat pass-band of the filter.

After signal passes through the DC blocking filter it is centered on zero, and has a signed two's complement representation. The signal is split into two paths, one path passes through a complex mixer, where signal is multiplied by the complex conjugate of the IF frequency. IF NCO has a resolution of about 610 Hz, which is the smallest frequency change a user can have. This is a much better resolution than 25 kHz, that the receiver synthesizer has, allowing better synchronization if needed. NCO frequency can be adjusted using REG3 and REG4 in ComBlock Control Center. The second path of the signal goes through the calculations to maximize signal input at the ADC. AGC block consists of two parts: a signal limiter, and an energy averager. Amplifier inside of the receiver is controlled by the voltage that comes from the DAC located on the FPGA module. DAC can work at the maximum frequency of about 1.7 MHz with a 10-bit resolution. Maximum gain is achieved with the input value of 0, and minimum gain is achieved with the input value of 1023. Frequency of the AGC voltage update was chosen to be 1 MHz, because of the ease of generating that frequency. The 1 MHz AGC DAC clock is generated from the 40 MHz ADC sampling clock.

Amplifier inside of the receiver has a dynamic range of 70 dB, from -40 to +30 dB gain range. AGC block in the FPGA controls the gain logarithmically similarly to the binary search algorithm. Below is the AGC's algorithm pseudo code:

```
MIN_GAIN = 1023
MAX_GAIN = 0
If gain is too low
   NEW_GAIN = OLD_GAIN - (MAX_GAIN + OLD_GAIN)/2
If gain is too high
   NEW_GAIN = OLD_GAIN + (MIN_GAIN - OLD_GAIN)/2
```

Using this algorithm, gain is slowly increased towards maximum or decreased towards a minimum if it is already close to it; otherwise gain is changed more quickly. Signal limiter of the AGC looks at the samples before they reach the DC blocking filter. Signal limiter has a priority of the gain adjustment over averager, because signal limiter only reduces gain. This is done to make sure that incoming samples never go over the ADC limit. Energy averager looks at the average energy of 2048 samples. With the current implementation this is equivalent to about 51 symbols. Energy averager compares calculated energy against the look-up-table of expected average energies based on the selected modulation, and adjusts gain accordingly. Table 3.1 below summarizes type of modulation, selection number, and expected average energy. Modulation can be changed using REG0 in the ComBlock Control Center.

Modulation	Average Energy Expected (integer/decimal)
BASK(0)	163 / 0.31835937
BPSK(1)	325 / 0.63476562
BFSK(2)	325 / 0.63476562
MSK(3)	325 / 0.63476562
QPSK(4)	325 / 0.63476562
O-QPSK(5)	325 / 0.63476562
4-ASK(6)	126 / 0.24609375
4-FSK(7)	325 / 0.63476562
8-PSK(8)	325 / 0.63476562
8-PAM(9)	139 / 0.27148437
16-QAM(10)	181 / 0.35351562
16-APSK(11)	256 / 0.50000000

Table 3.1: AGC look-up-table of expected energy values

Note that for constant modulus modulations such as PSKs, FSKs, and MSK, expected average energy is not 1. This is because the overall gain of the system as it will be described later in the chapter is 1.25, and if the input is not adjusted accordingly the output will overflow

3.2.2 Fixed Point Arithmetic

Throughout the system, even though the signal has an integer value, all the computations are done assuming fixed point notation, where 10-bit signed integer represents a fraction with 1 sign bit, and 9 fractional bits like shown in Figure 3.8

S	.	F	F	F	F	F	F	F	F	F

Figure 3.8: 10 bit fixed point representation

Using this representation, a maximum positive value that can be achieved is

$$2^9 - 1 = 511 = 0.998046875$$

and the maximum negative value is

$$-2^9 = -512 = -1.0$$

If more accuracy is needed, more bits are introduced to the fractional part. If more range is needed, imaginary binary point Figure 3.8 is moved to the right. Figure 3.9 shows a fixed point representation where the range is larger due to the extra integer bit.

S	I	.	F	F	F	F	F	F	F	F	F

Figure 3.9: 11-bit fixed point notation, with an integer part

To convert decimal number to a fixed point number, a decimal number needs to be multiplied by the maximum value that represents fraction and rounded:

$$0.2 * 512 = 102.4 = 102$$

$$1.4 * 512 = 716.8 = 717$$

To get a decimal number back from the integer, a reverse operation needs to be performed, and integer needs to be divided by maximum integer that represents fraction:

$$\frac{102}{512} = 0.19921875$$

$$\frac{717}{512} = 1.400390625$$

Fixed point notation is a common method that is used to represent a signal in DSPs and FPGAs. Compared to the floating point notation, which has a dynamic range for equivalent word length, fixed point notation does not require any special units to perform calculations, and arithmetic is simple integer arithmetic. Fixed point notation allows achieving a much higher computational speed compared to floating point arithmetic and because of that a preferred method for embedded signal processing.

Just like regular integer arithmetic, there are some issues that need to be taken into account. Addition and subtraction of 2 integers, representing fixed point numbers can cause overflow, which if not taken care of, will produce a wrong result:

$$200 + 50 => 0.390625 + 0.09765625 = 0.48828125 \quad (OK)$$

$$350 + 200 => 0.68359375 + 0.390625 = -0.92578125 \; (Overflow)$$

To mitigate this problem, more bits need to be used like in Figure 3.9, binary point needs to be moved to the right, or the values needs to be smaller.

To multiply two fixed point numbers, a regular integer multiplication is used with addition of shifting to the right (dividing). Example below shows what is happening when two fixed point numbers are multiplied (5 bit integers are used):

```
        10010 (-0.875)
        01011 (+0.6875)
-------------------------------------
     111110010 (extend sign bit)
  +   11110010
  +    0000000
  +     110010
  +      00000
-------------------------------------
    1101110110 (2 sign bits + 8 fractional bits)
```

Multiplication result has 2 sign bits, and 8 fractional bits. However, the original numbers had 1 sign bit and 4 fractional bits. To return result into the same form, product is shifted number of fractional bits to the right (shift 4 bits), and 1 sign bit is removed. The end result is 10111 or -0.5625, which is rounded version of actual result -0.6015625. Therefore after 2 fixed point numbers are multiplied, the result is shifted to the right, and first bit is removed.

When representation in Figure 3.8 is used for the fixed point number, multiplying two positive numbers never produces an overflow. Multiplying two negative numbers can produce an overflow, because +1.0 is not a valid positive fixed point number using notation in Figure 3.8. Using notation in Figure 3.9 an integer value will get larger when two numbers are multiplied, and because of that product needs to have longer bit length.

Just like with the regular integers, when two numbers with different integer lengths are added or subtracted, sign extension is performed. However, if two numbers with different fractional bit

lengths are added or subtracted, than bits are added and set to zero to the right of the shorter fraction. For example:

$$A = 0.11011 = 0.84375 \text{ (5 fractional bits)}$$
$$B = 0.1101100000 = 0.84375 \text{ (10 fractional bits)}$$

Throughout this paper and in all comments in the code, fixed point integers are referred using Texas Instruments Q-notation. When the number is Q0.9 it is 10-bits long, signed, has 9 fractional bits, and 0 integer bits. If the number is Q2.13 it is signed 16-bit long integer, with 13 fractional bits and 2 integer bits.

3.2.3 CIC Decimation

When signal is sampled, it is generally desired to have sampling rate higher than the Nyquist frequency, which is two times the maximum frequency component in a signal. This allows the analog anti-aliasing filters to be more relaxed. When signal gets converted to digital domain, however, sampling rate should be relatively low. The amount of power and resources required grows exponentially with the clock frequency, and therefore sampling clock frequency needs to be reduced. This process is called down-sampling, and needs to be performed carefully to avoid aliasing of the higher frequency components into the desired band. There are two methods to do down-sampling: using an FIR filter followed by decimation, or using the CIC filter with the embedded decimation. CIC stands for Cascaded Integrator Comb, and has the structure presented in Figure 3.10 and has a transfer function shown by equation (3.2) where *R* is the down-sampling rate, *M* is the comb delay, which is typically 1 or 2, and *N* is the number of stages. As it can be seen, compared to FIR filters, CIC filters do not have any multipliers, and therefore a preferred method of down-sampling in FPGAs. The drawback of the CIC filters is non-flat pass-band response that can be corrected by FIR filter. If the input signal bandwidth is within 1-2 dB of the CIC filter cutoff frequency, correctional FIR filter can be avoided.

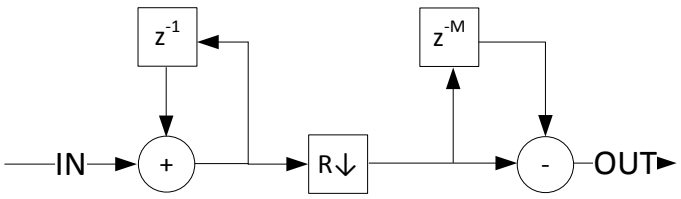

Figure 3.10: Block diagram of the CIC filter

$$H(z) = \left(\frac{1-z^{-RM}}{1-z^{-1}}\right)^N \tag{3.2}$$

The CIC filter has a gain of

$$G = (RM)^N$$

which means the output needs to be truncated if same bit length output is desired. Because of the gain, the internal registers inside of the CIC need to be able to accommodate the largest number. Expression below calculates the maximum internal register length required [10]:

$$B_{out} = \lceil N * \log_2 RM + B_{in} \rceil \tag{3.3}$$

To make CIC filter pass band smaller more stages can be cascaded together or the comb delay M can be switched from 1 to 2. Comb delay is responsible for the locations of the nulls. For this project N = 1 stage CIC was used with R = 4, and M = 1. These parameters give the frequency response shown in Figure 3.11. Given these parameters and using equation (3.3) with $B_{in} = 10$, the maximum register length needs to be$B_{out} = 12$.

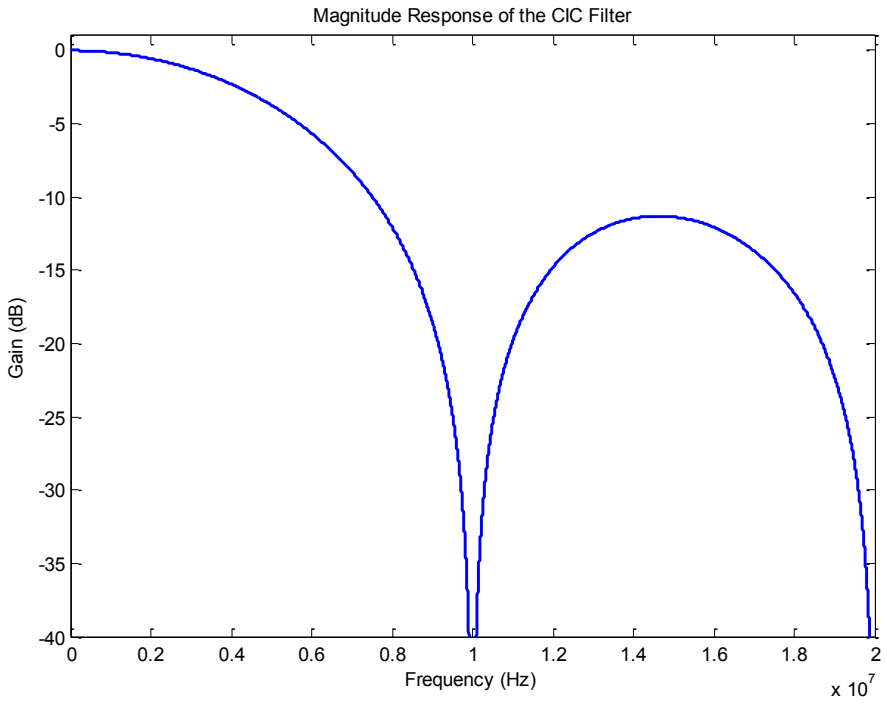

Figure 3.11: CIC filter response

Decimation of 4 was chosen to reduce number of samples per symbol from 40 to 10, which was the best number of samples for good matched filtering. When the number of samples was chosen to be 5 or 4 (R = 8 and R = 10) instead, matched filter outputs did not have a full amplitude, due to quantizing effects. As it is seen in Figure 3.11 there is a null located at the frequency of 10 MHz, which makes it an ideal place for signal's IF frequency. When IF frequency is equal to 10 MHz, all residual IF carrier will be filtered out after passing through CIC filter.

3.2.4 Matched Filtering

In order to achieve the best performance at the receiver, a matched filter needs to be used, as discussed in 1.6.2. If the transmitting pulses are symmetrical, than the matched filter is the exact replica of the transmitting pulse. In this project square pulses were used for transmission, except MSK modulation which uses half-sine shaped pulses. Square pulses were chosen due to the fact that it was easy to implement transmitter and the receiver using them. For the square pulses, the matched filter can be implemented using an integrate-and-dump method, or using an FIR filter with all coefficients being the same. An FIR filter in the FPGA would require a lot of resources, and for the square pulses, multiplications are done by the same value. An integrate and dump method on the other hand is simple to implement, but without modification it cannot output the mid-symbol points required for the timing detector that follows it. For this reason and FIR-like structure was made for the matched filter, and is shown in Figure 3.12.

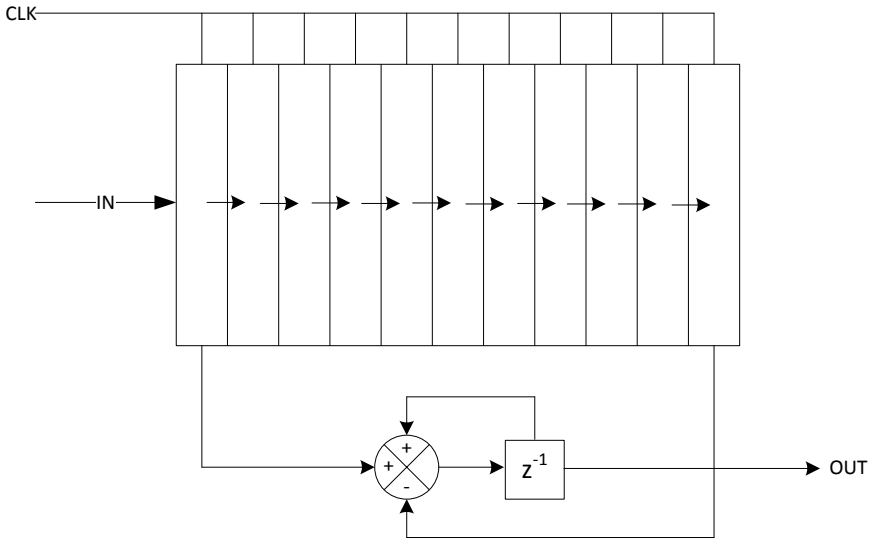

Figure 3.12: Structure of the implemented matched filter

Matched filter is implemented as a FIFO of N+1 length, where N is the number of samples in a symbol after decimation. In this project the sample rate after decimation was 10, which made the FIFO length of 11. Implemented matched filter integrates 10 samples in a row, and divides the result by 8. This is the reason why a gain of 1.25 is produced. A division by 8 in the FPGA is just 3 bit shifts to the right, and much easier to implement than an integer division. Because matched filter acts as a sliding window, as soon as 11^{th} sample is added, the first sample needs to be subtracted. In order to get samples at half-symbol interval for the timing error detector, samples are latched in at 2 different points. When internal sample counter equals to 5, half-symbol integration is latched in, when the counter equals to 0 (start of a new integration), full symbol integration is latched in. Matched filter uses one clock, generated by the clock generator that has a symbols' frequency of 1 MHz. Matched filter has a square pulse frequency response shown in Figure 3.13. Note an overall power gain of 1.94 dB.

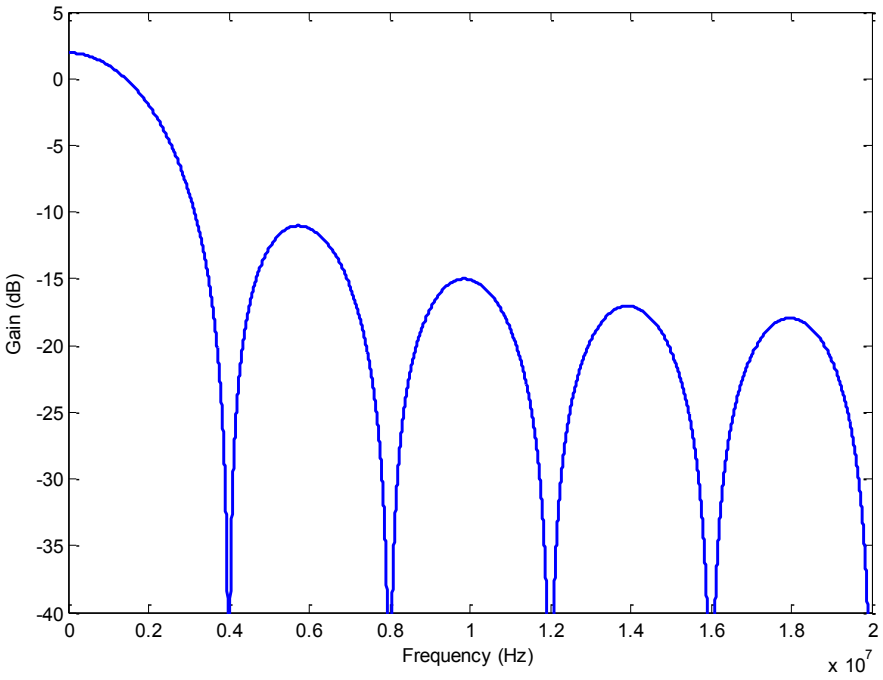

Figure 3.13: Frequency response of the matched filter

3.2.5 Timing Error Detector and Delay Adjustment

In addition to the frequency and phase offset that is present due to receiving and transmitting oscillator differences, there is also an error in sampling clock. Sampling clock error causes additional performance drop, because samples from the matched filter do not have maximum energy. This is the reason a block memory was introduces in AGC described in chapter 3.2.1.

Block memory stores uniformly spaced samples, which do not have a timing error problem. For the real life input from the ADC, however, timing error causes square pulses to move in time, and sampling instant needs to be adjusted continuously.

One of the methods to adjust timing is to calculate error using digital samples past the ADC, and adjust sampling clock before the ADC. For this project, such configuration was not possible. In order to adjust timing, a delay module was put after the CIC filter, and before the matched filer. Delay module is N samples long FIFO, where N is the number of samples per symbols. Timing error detector module determines if the timing delay needs to be increased, or decreased, and sample is taken from the correct position in the FIFO. Because the length of the FIFO is N samples long, N-1 delays can be achieved, with Nth delay being prompt. Delay adjustment was inserted after the CIC for two reasons: resource savings, and practicality. If the delay adjustment module was to be inserted after the AGC, and before the CIC, N = 40, which is much larger FIFO, and more FPGA fabric would be needed. Additionally, when samples pass CIC, some of them are removed, and adjusting delay by 1/40 of a sample, will lead to no change after the CIC. Because of that, it is impractical and wasteful to make a much finer delay adjustment, when much smaller amount is actually detectable.

In order to determine the actual timing error, a timing error detector module is used, and inserted after the matched filter. Timing error detectors can be decision and non-decision directed. For the current project structure, decision directed timing error detectors would be much harder to implement, and therefore non-decision directed approach was used. There are multiple types of the non-decision directed timing error detectors. Experimentally it was found that the most practical timing error detector was Early-Late Gate detector whose description can be found in [2, 28], while error expression for it can be found in [28].

Early-Late Gate detector calculates error based on the sample derivative. Detector's work is best described by the Figure 3.14. When the timing is correct, nT and $(n+1)T$ samples will have maximum energy, and difference between half-symbol points $nT + \frac{T}{2}$ and $nT - \frac{T}{2}$ will be equal to zero. When the timing is off, derivative will either have positive or negative value, which will indicate if the delay needs to be reduced or increased. Instantaneous error will not necessarily always be zero, because of random symbol transitions, but on average the error will be zero. The Early-Late Gate detector works with any type of modulation.

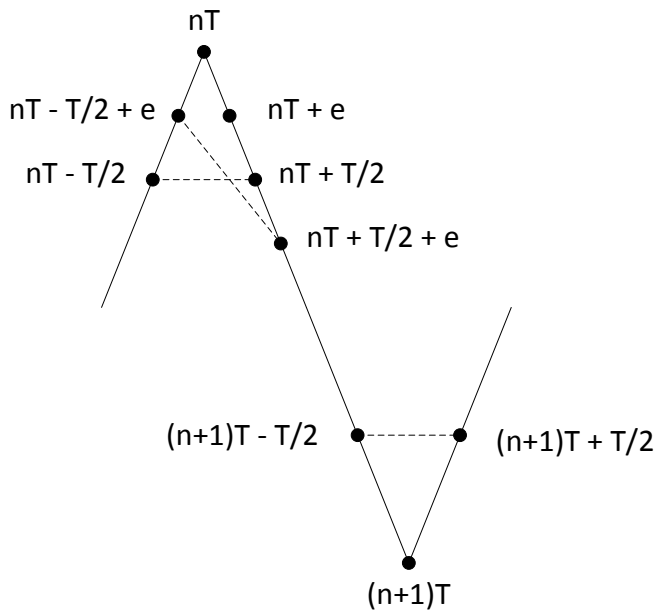

Figure 3.14: Early-Late gate principle

Expression (3.4), and (3.5) from [28] show required operations for the error calculations. As it can be seen, in-phase and quadrature branches are calculated independently. Also, each branch requires one multiplier.

$$e_I = I(nT) * \left(I\left(nT + \frac{T}{2}\right) - I\left(nT - \frac{T}{2}\right) \right) \tag{3.4}$$

$$e_Q = Q(nT) * \left(Q\left(nT + \frac{T}{2}\right) - Q\left(nT - \frac{T}{2}\right) \right) \tag{3.5}$$

$$e = e + K * \left(e_I + e_Q\right) \tag{3.6}$$

After each branch is calculated, total error is calculated using a simple proportional-integral filter with reset shown in expression (3.6). Total error is averaged out over predetermined interval, and final value is tested at the end of the interval. After that, filter is reset. The value of K was chosen to be $1/D$, where D is the integration interval length in symbols. This proportional-integral filter works as an averager. At the end of the averaging interval total error is compared to the predefined limits, and delay is incremented for negative error, and decremented for positive error. When delay is outside of the FIFO limits, it wraps back to 0 or 9. In order to satisfy timing requirements during synthesis of the code for the FPGA, timing error limits were set internally, and can be selected using REG5 from the ComBlock Control Center.

Selection number	Error Limit
0	-40 to +40
1	-30 to +30
2	-20 to +20
3	-15 to +15

Table 3.2: REG5 values and error limits

Experimentally it was determined that selection 0 works best with the PSK, QAM, and PAM type modulations, while selection 2 and 3 works best with the BASK, and 4-ASK respectively.

3.2.6 System Clocking

There are two main clocks that are being input to the FPGA. They are a 60 MHz USB clock, and a 40 MHz clock coming from the ADC. A 60 MHz USB clock is used to clock the USB communication circuit, and also gets multiplied inside of the FPGA to a higher frequency using DCM (Digital Clock Manager). After USB clock is multiplied inside of the DCM, it has a frequency of 80 MHz, and becomes a system clock throughout FPGA. A 40 MHz ADC clock is used to latch in samples at the ADC, and passes through the DCM, where its phase is shifted by 10 ns. This is done such that there is sufficient timing at the ADC for the sample to settle before they are latched into the registers of the FPGA.

An 80 MHz system clock is used for the operation of peripheral components on the FPGA. It is responsible for clocking ComScope (an internal signal analyzer), an AGC DAC controller, a DSP communication controller, and a USB interface. A 40 MHz ADC clock is used throughout the signal processing path. A clock generator module divides 40 MHz clock into the 10 MHz CIC clock, and a 1 MHz symbol clock. A 10 MHz CIC clock is used to latch in samples after they pass CIC filter, and 1 MHz symbol clock is used for the matched filter and timing error detector. All signal processing modules on the FPGA have an input and output registers to reduce timing requirements.

3.2.7 DSP Communication Controller

DSP communication controller is one of the most important blocks in the FPGA, as it is responsible for data transfer between FPGA registers and a DSP for further processing. Host Port Interface (HPI) port on the DSP side is designed for external control, and therefore works almost

solely from the commands sent to it from FPGA. HPI in addition to the multiplexed address/data pins also provides couple control pins that can be used to trigger FPGA. Those pins are HINT, and HRDY. Both HRDY and HINT pins are active low, which means they are "on" when they equal to 0. HRDY pin informs FPGA if the HPI port is ready to read or receive data, while HINT pin tells FPGA if DSP has requested an interrupt. A detailed description of HPI port can be found in [16].

Simplified block diagram of the DSP communication controller is shown in Figure 3.15. A more detailed description of the operation of the HPI port will be described in the following chapters. HPI port needs an external strobe to latch in and out commands, address, and data, which is generated using system clock. The strobe has a rate of 5 MHz, but the data input is only at 1 MHz. In addition, data input is not continuous and some strobe cycles are used for the configuration and communication commands.

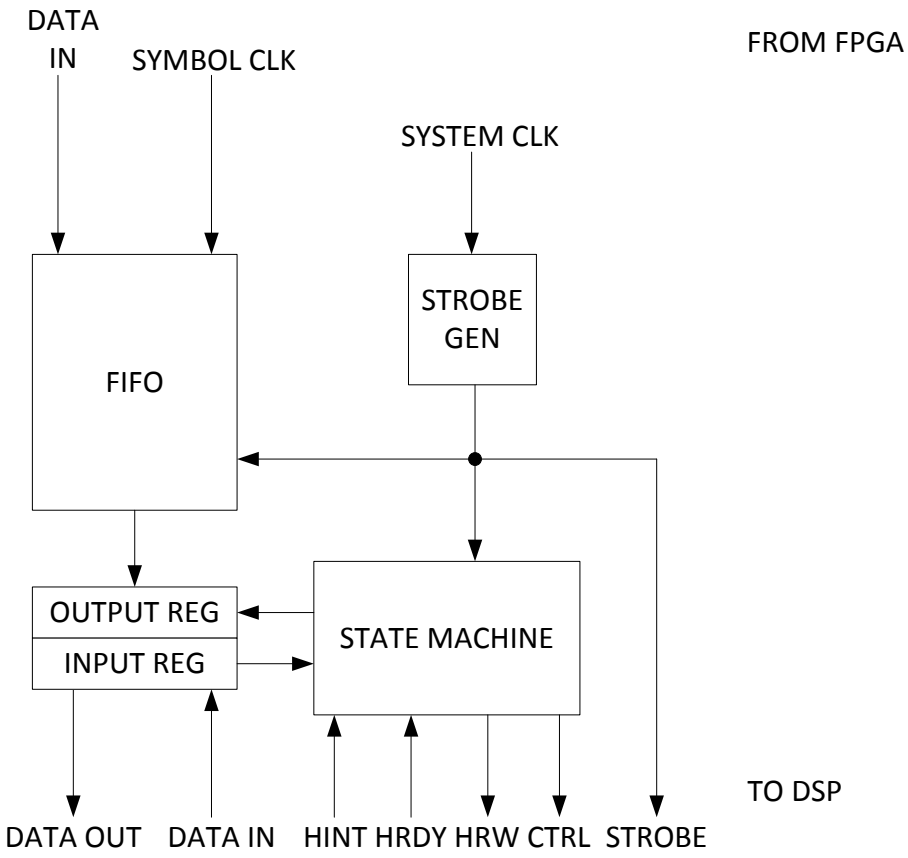

Figure 3.15: Simplified diagram of the DSP communication controller

Due to this reason a FIFO is inserted between incoming and outgoing data. FIFO has two clock domains, where data is input at the symbol clock rate, while data is read at the strobe rate.

This allows use of the same clock for the state machine, and data transfer. Because incoming data rate is slower than outgoing data rate, FIFO is always available for writing, making sure continuous samples are not lost, but reading is controlled. Data is read from the DSP and written to the DSP using the same register, which means it needed to be bidirectional. HINT and HRDY pins are used as one of the inputs to the state machine, while HRW and HCTRL pins are used for the HPI port control.

HPI port write timing diagram is shown in Figure 3.21. As it can be seen, the control command is latched in at the falling edge of the strobe, while actual data from the port is latch in at the rising edge. Because FPGA internally generates inverted strobe, state machine changes control commands on the rising edge, and latches data on the falling edge.

Due to the dual edge clocking, DSP communication controller finite state machine is a hybrid state machine between Mealy and Moore machines. Mealy finite state machine outputs change as a function of inputs, and a state. A simple example of Mealy finite state machine, with the traditional notation is shown in Figure 3.16

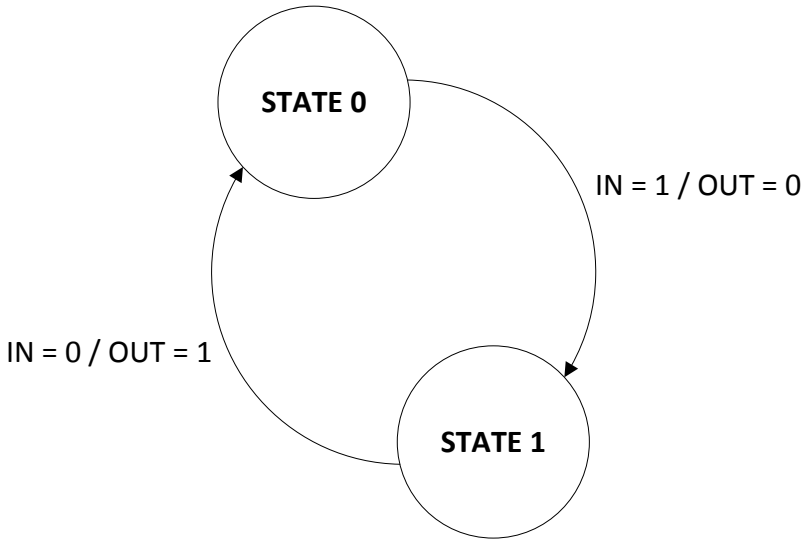

Figure 3.16: An example of Mealy FSM

A Moore finite state machine only changes states as a function of inputs and outputs only depend on the state. A simple Moore finite state machine is shown in Figure 3.17 with the traditional notation. A DSP communication controller finite state machine is shown in Figure 3.18. There are a total of 6 states, and state transitions happen at the rising strobe edge (a falling HPI strobe edge). As the states change, control inputs to the DSP are set at the CTRL pins. After

state changes, the data is latched in for the currently set control mode at the falling strobe edge (a rising HPI strobe edge).

When DSP first boots up, or when the user previously turned off the DSP communication controller, state machine starts in the START state. A state machine stays in the START state until interrupt is received from the DSP, and the writing function is disabled.

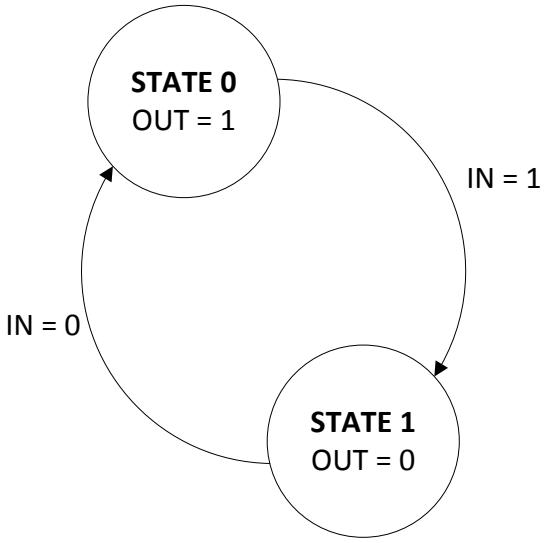

Figure 3.17: An example of Moore FSM

As soon as DSP triggers an interrupt by writing to the control register (operation described later in details), HINT pin is set high, which causes state machine transition to the INTACK state. INTACK state is needed to acknowledge an interrupt from the DSP, and turn on the writing mode. A state machine stays in the INTACK state until HINT pin goes high. When HINT pin goes high, a state transition occurs to the WADDR state, and a write address is latched into the DSP. Data transfer is performed using double buffering, and one of the two addresses is written to the DSP. Controller writes address to the DSP once, and waits until DSP is ready to start receiving data. Usually, the process happens instantaneously. As soon as the address is latched in, data starts to be written from the FIFO in WDATA state using auto-incrementing address until buffer on the DSP is full. The buffer size was chosen to be 256 words long. When the write buffer is full, state machine transitions to the CPUWAIT state, and writes to the DSP control register, triggering interrupt on the DSP.

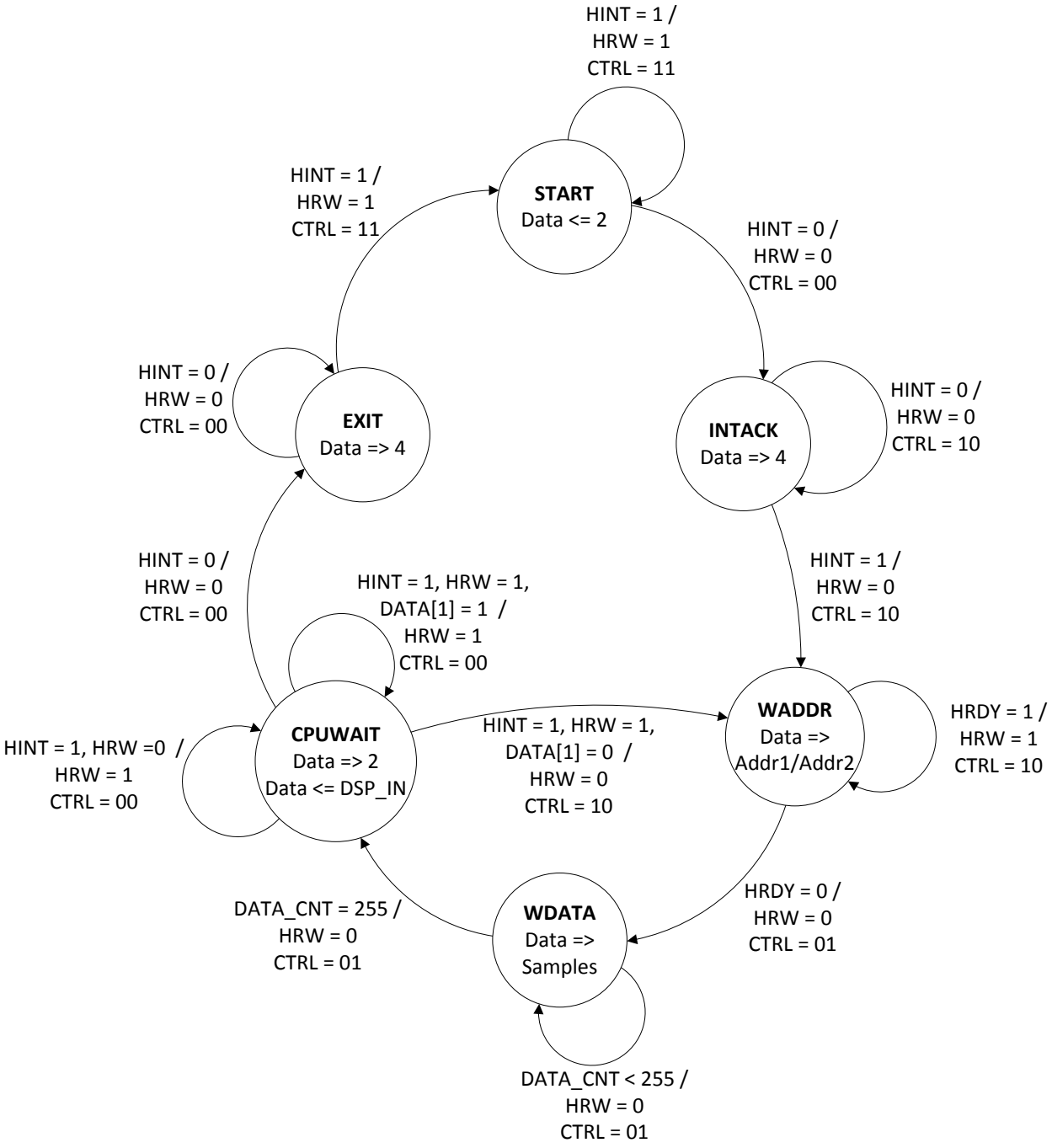

Figure 3.18: Finite state machine for the FPGA-DSP communication

In order to continue writing data, DSP needs to acknowledge an interrupt, by clearing interrupt bit. State machine stays in the CPUWAIT state until DSP interrupt bit reads 0. One cycle is used to transition state machine from the writing state, to the reading state. When DSP acknowledges an interrupt, state machine returns to the WADDR state and writes a second address of the two to the DSP, after which the data writing process continues. If at any point user wants to stop DSP

communication controller, an interrupt needs to be triggered by the DSP. This causes state machine to transition from CPUWAIT state, to the EXIT state, where interrupt is acknowledged similarly to the INTACK state. After that state machine transitions back to the START state, and the process can be restarted if a new interrupt is triggered by the DSP.

3.3 FPGA to DSP Interface

3.3.1 Interface Electrical Wiring

Similarly to the receiver hardware, an interface between FPGA and HPI port was needed. Just like many other ComBlocks, FPGA output was a 40 pin 2 mm spacing connector. In order to connect it to the DSP, a 40 pin IDE cable was used together with the laptop IDE adapter. The laptop IDE adapter converts regular 40-pin 0.1 inch spacing connector to the 44 pin 2 mm spacing connector. The other side of the IDE cable was cut, and using the board in Figure 2.3 an electrical connection was made to the HPI port. Table 3.3 shows the mapping of the DSP pins to the FPGA pins. The data mapping was made the same way as in the receiver, and is shown in Figure 2.1. Pinout of the HPI connector on the DSP is shown in Table 3.4. Connector on the FPGA side has been mapped differently from the default configuration.

DSP Pins	FPGA pins	Pin Designation
12	B1	HD0
9	A1	HD1
14	B2	HD2
11	A2	HD3
16	B3	HD4
13	A3	HD5
18	B4	HD6
15	A4	HD7
19	A5	HD8
22	B6	HD9
21		HD10 – SW
24		HD11 – SW
23		HD12 – SW
26		HD13 – SW
25		HD14 – SW
28		HD15 – SW
48	A6	HD16

51	B7	HD17
50	A7	HD18
53	B8	HD19
52	A8	HD20
55	B9	HD21
54	A9	HD22
57	A10	HD23
60	B11	HD24
63	A11	HD25
62		HD26 – SW
65		HD27 – SW
64		HD28 – SW
67		HD29 – SW
66		HD30 – SW
69		HD31 – SW
33	B12	HDS1#
29		HDS2# - GND
37		HCS# - GND
49	A14	HR/W
36	A12	HCNTL0
41	B13	HCNTL1
44	B14	HINT#
45	A13	HRDY#
32		HAS# - VCC +3.3
17	B5	GND – GND
27	B10	GND – GND
34	B15	GND – GND

Table 3.3: FPGA to DSP HPI port pin mapping

Default configuration for the FPGA maps pin 1 on the connector J8 to the position A1, pin 2 to the position B1, pin 3 to the position A2, and so on. Using this mapping for the FPGA output was very inconvenient when the code was written, and because of that mapping was changed such that pin 1 is located at position B1, pin 2 is located at position A1, pin 3 is located at position B2, and so on. Grounds on the connector J8 were connected to the grounds on the HPI connector.

Internal strobe distribution and control for the HPI port can be seen in Figure 3.19. HPI connector provides access to the two strobes such that writing and reading can be done independently. For this project this was not needed, and HDS2 input was grounded. Using this configuration HDS1 input needs to be active high. HCS input of the HPI connector provides means to turn on or off port reading and writing completely, but the data latching does not occur

unless strobe is present. Because FPGA controls the strobe manually, there is no need for the HCS input when HPI port is not used. HRDY output signals the FPGA when the port cannot accept more data for the writing, or does not have data available for the reading. As the HCS input is always grounded, meaning the HPI is always enabled, HRDY pin output depends only on condition of the writing and reading queues. Host Address Strobe or HAS input of the HPI port is useful when there are no dedicated output pins on the host device for control pins HCNTL0 and HCNTL1. As FPGA is fully configurable, this is not an issue, and HAS input has been tied to the 3.3V pin.

Pin	Signal	I/O	Description	Pin	Signal	I/O	Description
1	PCI_EN	I	PCI enable	2	Resv	N/C	Resv
3	GND	Vss	System ground	4	HPI_RS#	I	HPI reset
5	Resv	N/C	Resv	6	Resv	N/C	Resv
7	GND	Vss	System ground	8	GND	Vss	System ground
9	HD1	I/O	Host data 1	10	PCBE0#	I/O	PCI command/byte ena 0
11	HD3	I/O	Host data 3	12	HD0	I/O	Host data 0
13	HD5	I/O	Host data 5	14	HD2	I/O	Host data 2
15	HD7	I/O	Host data 7	16	HD4	I/O	Host data 4
17	GND		System ground	18	HD6	I/O	Host data 6
19	HD8	I/O	Host data 8	20	GND	Vss	System ground
21	HD10	I/O	Host data 10	22	HD9	I/O	Host data 9
23	HD12	I/O	Host data 12	24	HD11	I/O	Host data 11
25	HD14	I/O	Host data 14	26	HD13	I/O	Host data 13
27	GND	Vss	System ground	28	HD15	I/O	Host data 15
29	HDS2#	I/O	Host data strobe 2	30	GND	Vss	System ground
31	GND	Vss	System ground	32	HAS#	I/O	Host address strobe
33	HDS1#	I/O	Host data strobe 1	34	GND	Vss	System ground
35	GND	Vss	System ground	36	HCNTL0	I/O	Host Control 0
37	HCS#	I/O	Host chip select	38	GND	Vss	System ground
39	GND	Vss	System ground	40	PTRDY#	I/O	PCI target ready
41	HCNTL1	I/O	Host Control 1	42	GND	Vss	System ground
43	GND	Vss	System ground	44	HINT#	I/O	Host Interrupt
45	HRDY#	I/O	Host ready	46	GND	Vss	System ground
47	GND	Vss	System ground	48	HD16	I/O	Host data 16
49	HR/W#	I/O	Host Read/Write	50	HD18	I/O	Host data 18
51	HD17	I/O	Host data 17	52	HD20	I/O	Host data 20
53	HD19	I/O	Host data 19	54	HD22	I/O	Host data 22
55	HD21	I/O	Host data 21	56	GND	Vss	System ground
57	HD23	I/O	Host data 23	58	Resv		Resv
59	Resv		Resv	60	HD24	I/O	Host data 24
61	GND	Vss	System ground	62	HD26	I/O	Host data 26
63	HD25	I/O	Host data 25	64	HD28	I/O	Host data 28
65	HD27	I/O	Host data 27	66	HD30	I/O	Host data 30
67	HD29	I/O	Host data 29	68	PGNT#	I	PCI bus grant
69	HD31	I/O	Host data 31	70	GND	Vss	System ground
71	GND	Vss	System ground	72	Resv		Resv
73	Resv		Resv	74	GND	Vss	System ground
75	GND	Vss	System ground	76	Resv		Resv
77	Resv		Resv	78	GND	Vss	System ground
79	GND	Vss	System ground	80	N/C	-	No connect

Table 3.4: Host Port Interface (HPI) external connector pinout [5] [Fair Use]

Because FPGA does not have enough pins to control all 32 bits of the shared address/ data lines, and control pins, switches were introduced. The default values of the HPI pins HD31 to HD26, and HD15 to HD10 when nothing is connected to them is logic 1. All the switches that are connected to the pins HD31 to HD26, and HD15 to HD10 are grounded. When the switches are in the "on" position, the HPI pins are grounded and have a logic value of 0. When the switches are in the "off" position the pins are floating and have a logic value of 1. Using the switches the user can set the desired address correctly. When more than one address needs to be used, like in this project, two addresses should not be separated by more than 1024 words (4096 bytes); otherwise user would need to change the switches as well.

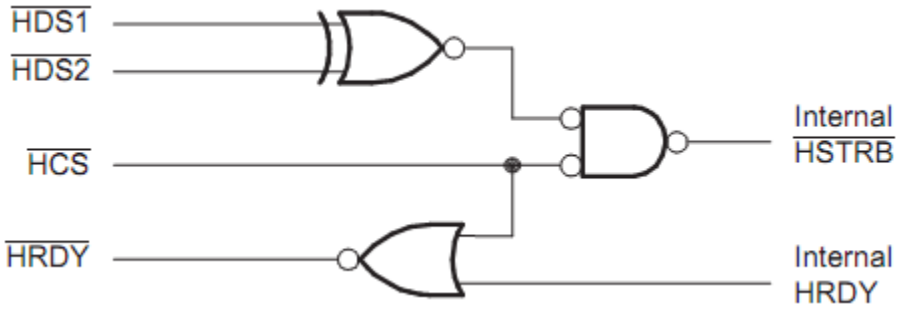

Figure 3.19: HPI strobe control logic [16] [Fair Use]

3.3.2 HPI Port Internal Workings

HPI port is very different in control and configuration to the EMIF port on the transmitter side. The most important difference is that HPI port is designed to be controlled by the outside device. In this project's case it's an FPGA. Second important difference is that due to the fact that DSP cannot write almost anything to the HPI configuration register, port configuration needs to be done by the external device as well. Finally, data and address lines of the HPI port are multiplexed, and address for the HPI port is written in the word format, rather than in the byte format that is usually used. In order to write data to the address 0x800000 for example, HPI address would need to be set to 0x200000.

Due to the fact that data, address, and configuration register are all written using the same data lines, control pins HCNTL0 and HCNTL1 are used for the mode selection. Table 3.5 shows possible values of the controls pins, and the mode that HPI is in. HPIC register is responsible for the HPI port's configuration, and contains some indication flags as well. HPIA register holds the

value of the address for the external device to read from or write to. If the auto-incrementing mode is used than HPIA address is automatically incremented every read or write cycle. HPID register stores the value of the data to be written to the address in HPIA from external device, or the data that was fetched from the memory to be read by the external device.

HCNTL1	HCNTL0	Description
0	0	HPIC access. The host requests to access the HPI control register (HPIC).
0	1	HPID access with autoincrementing. The host requests to access the HPI data register (HPID) and to have the appropriate HPI address register (HPIAR and/or HPIAW) automatically incremented by 1 after the access.
1	0	HPIA access. The host requests to access the appropriate HPI address register (HPIAR and/or HPIAW).
1	1	HPID access without autoincrementing. The host requests to access the HPI data register (HPID) but requests no automatic post-increment of the HPI address register.

Table 3.5: Access type based on control signal value [16] [Fair Use]

HPI data is written by the external device directly to the DSP's memory using HPI's own direct memory access (DMA). After HPI address is set in HPIA register, external device can either read/write one word at a time, or perform read/write with the address auto-incrementing following control pins values in Table 3.5. Data writing and reading process is shown in Figure 3.20. When the external device uses writing with auto-incrementing address mode, data is written to the input FIFO until FIFO is 4 words full, and then DMA transfers the data to the required address. When address auto-incrementing is not used, data written to the port is immediately transferred by the DMA logic to the memory. In addition to the writing FIFO, there is also a read FIFO, and a fetch command available in the control register. As there was no data reading from DSP's memory in this project, the process will not be described,

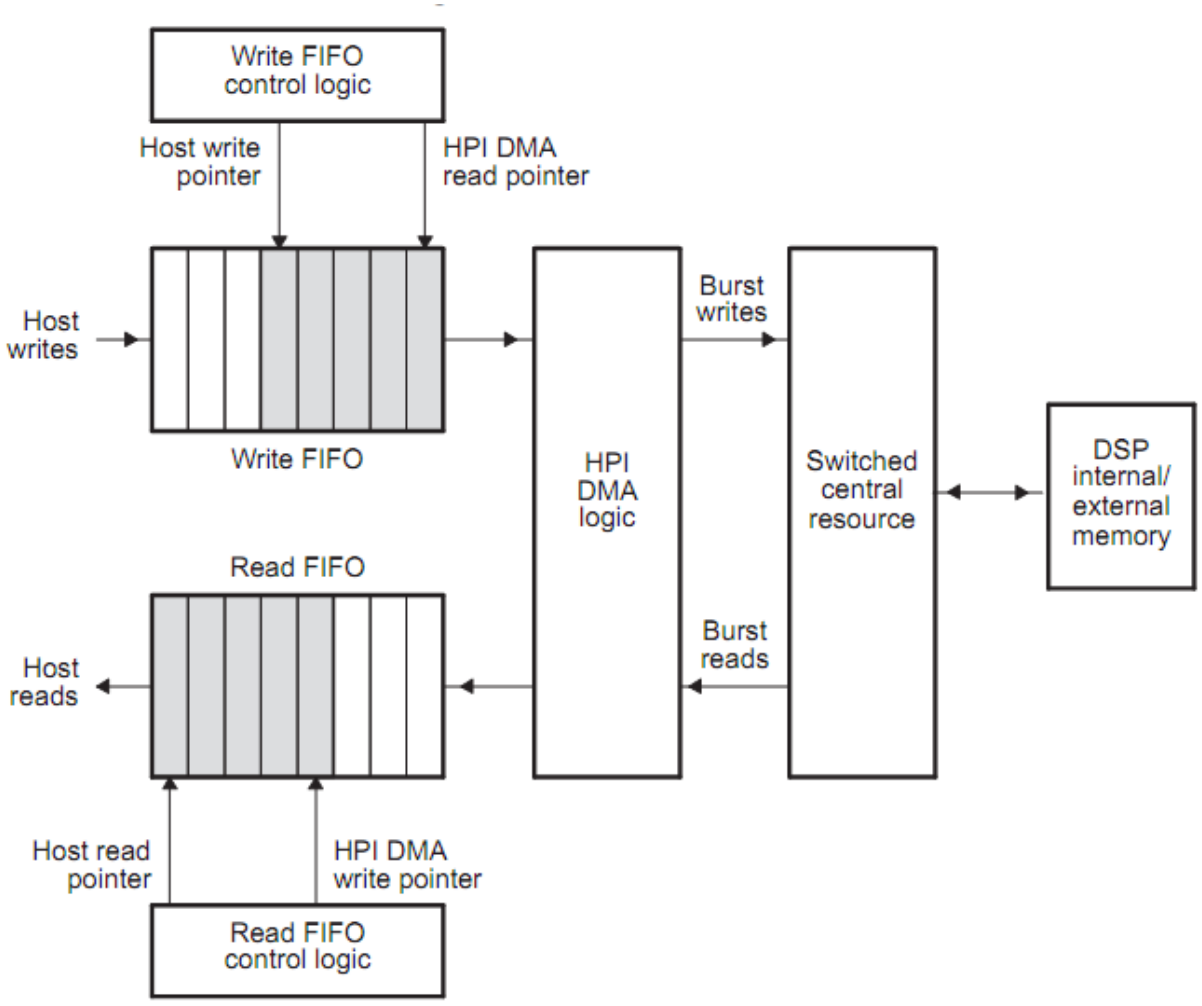

Figure 3.20: HPI interface DMA logic [16] [Fair Use]

and the reader can look at it in more detail in the HPI reference [16]. HPI writes are performed following timing diagram in Figure 3.21. As it can be seen control commands are latched in at the falling edge of the internal strobe, and data is latched in at the rising edge of the internal strobe. This is the reason for the hybrid nature of the finite state machine in the FPGA as was described in chapter 3.2.7. When the HPI port is busy with the fetching of the data from the memory for the read operation, or transferring data to the memory for the write operation, HRDY pin goes high indicating to the external device that operation cannot be performed. If the user does not want writes or reads to be stopped often, address auto-incrementing can be used, or the strobe can be made longer.

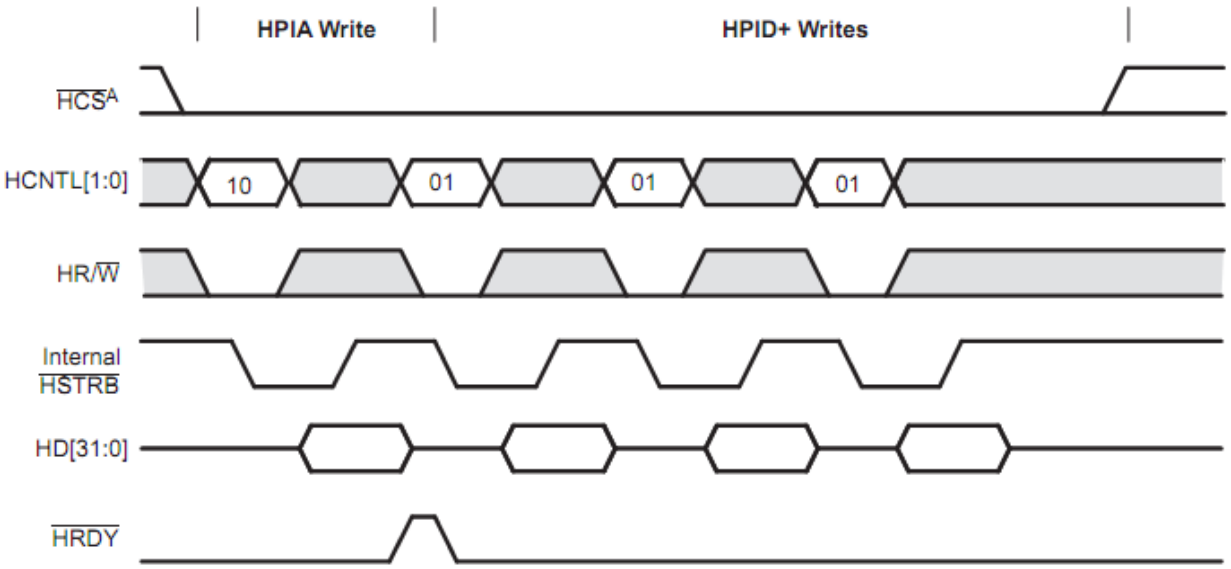

Figure 3.21: HPI interface write timing diagram [16] [Fair Use]

4. DSP Software

Digital Signal Processor it at the center of the whole transceiver. It is responsible for modulating and demodulating data, mapping bits to symbols, calculating bit error rates, and forming data frames for reception and transmission. DSP is also responsible for exchanging information with the PC. In addition to all of the above function, DSP board also contains audio codecs which can be used as another signal source. Timer outputs available, can be used to trigger frequency change at the receiving and transmitting modules. DSP works at the frequency of 1200 MHz and is capable of up to 8000 Million Instructions per Second (MIPS). This is achievable due to the DSP architecture which allows multiple instructions to be executed in parallel. An overall software flow diagram is shown in Figure 4.1

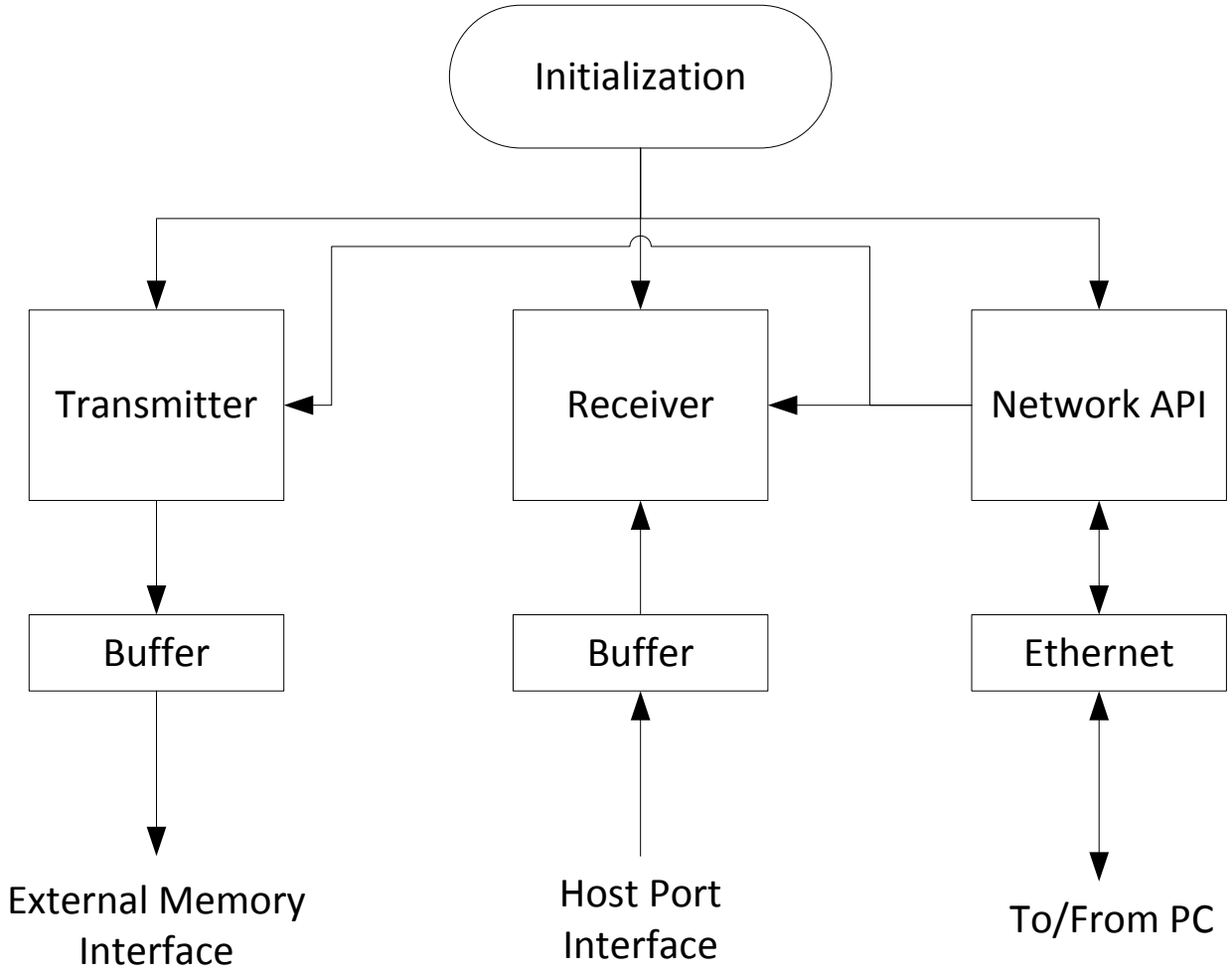

Figure 4.1: Software flow diagram

4.1 DSP Programming

There are two main methods of programming DSP or any other processor. One method is cyclic executive method, where everything happens in a large infinite loop, and processor periodically responds to interrupts. Another method is using scheduler, which usually comes with operating system (OS). The cyclic executive method is shown with the pseudo-code below:

```
Interrupt Handler 1 {
  Perform interrupt tasks 1;
  Return;
}

Interrupt Handler 2 {
  Perform interrupt tasks 2;
  Return;
}

main() {
  Initialize code;

  while(1) {
       Do processing;
       Function 1();
       Function 2();
  }
}
```

Cyclic executive method is pretty simple, and does not require any extra overhead. All the code that is being used is for performing particular function. There is no extra code that is idle, because it would stall the infinite loop. Cyclic executive method, however, starts to have problems as the size of the program increases. There are couple of major problems with the larger and more complex code. First problem is that any time new function or the code needs to be added, it needs to be placed in the loop correctly. If the added code is blocking for example, which means it's waiting for something, then code following it will stall. Second problem is interrupt handling. Suppose, interrupt 1 happens every 1 second and is quick to execute, and interrupt 2 happens every 2 seconds, but takes longer time to execute. There might be a situation where interrupt 2 is still executing while interrupt 1 happens. As the processor usually disables all interrupts while inside of the interrupt handler, interrupt 1 will be simply ignored. Finally there is problem with function execution. When function is executing, it can only be preempted by an interrupt, and there is no way to assign priority to functions.

A lot of the problems with the Cyclic executive method can be resolved if there is scheduler available. A scheduler is a code that's being constantly executed on the lower layer, and manages order and amount of time each function is executed based on the function's priority. A function that scheduler executes is referred to as thread. There are different types of threads, which will be described in the next chapter. Disadvantage of the scheduler code is a much larger overhead. Scheduler also needs to have some kind of clock reference, such that time of execution can be determined. Using scheduler, the pseudo-code would look something like this:

```
Interrupt Handler 1 {       Thread 1 {      Thread 3 {
  Call thread 1;             Do work;        Do    non-interrupt
  work;                      End;            End;
  Return;                   }               }
}

Interrupt Handler 2 {       Thread 2 {
  Call thread 2;             Do work;
  Return;                    End;
}                           }

main() {
  Initialization code;
  Run scheduler;
}
```

When interrupt happens with the scheduler, thread is created and interrupt exits. This way scheduler takes care of the code, and manages its execution. If another interrupt comes in while thread is being executed, a different thread is created, and now 2 threads are managed by scheduler. In addition to the threads created by interrupts, there are also independently executing threads. Threads can have functions in them, which are not controlled directly by the scheduler. Using scheduler also simplifies code expansion and modification. If the new functionality needs to be added, a new thread is created, and it does not matter when it is being executed, as long as there is no data sharing between threads. When program is written using scheduler, a programmer needs to keep in mind two concepts: there should be synchronization if needed, and threads need to be pre-emptible. Threads need to be able to be pre-emptible because otherwise scheduler would not be able to execute anything else. Scheduler makes thread execution look parallel, which means that if there is a shared variable, programmer needs to make sure it's

updated correctly. Synchronization and pre-emption can be done using multiple methods, but some of the common ones are semaphores and message queues.

4.1.1 Texas Instrument RTOS/BIOS

If the user wishes to write the program for the Texas Instrument DSP using scheduler, they can use Texas Instrument Real Time Operating System (RTOS) called BIOS. TI DSP BIOS is not the only available RTOS for the DSP, however it is provided with the Code Composer Studio and there are multiple drivers already written for peripheral control, which were used in the project. An introduction to the TI DSP BIOS can be found in [24].

BIOS can be configured multiple ways. Main way to configure BIOS is using the GUI, which in turn writes a text file that can be edited by user separately. A GUI for the BIOS configuration is show in Figure 4.2.

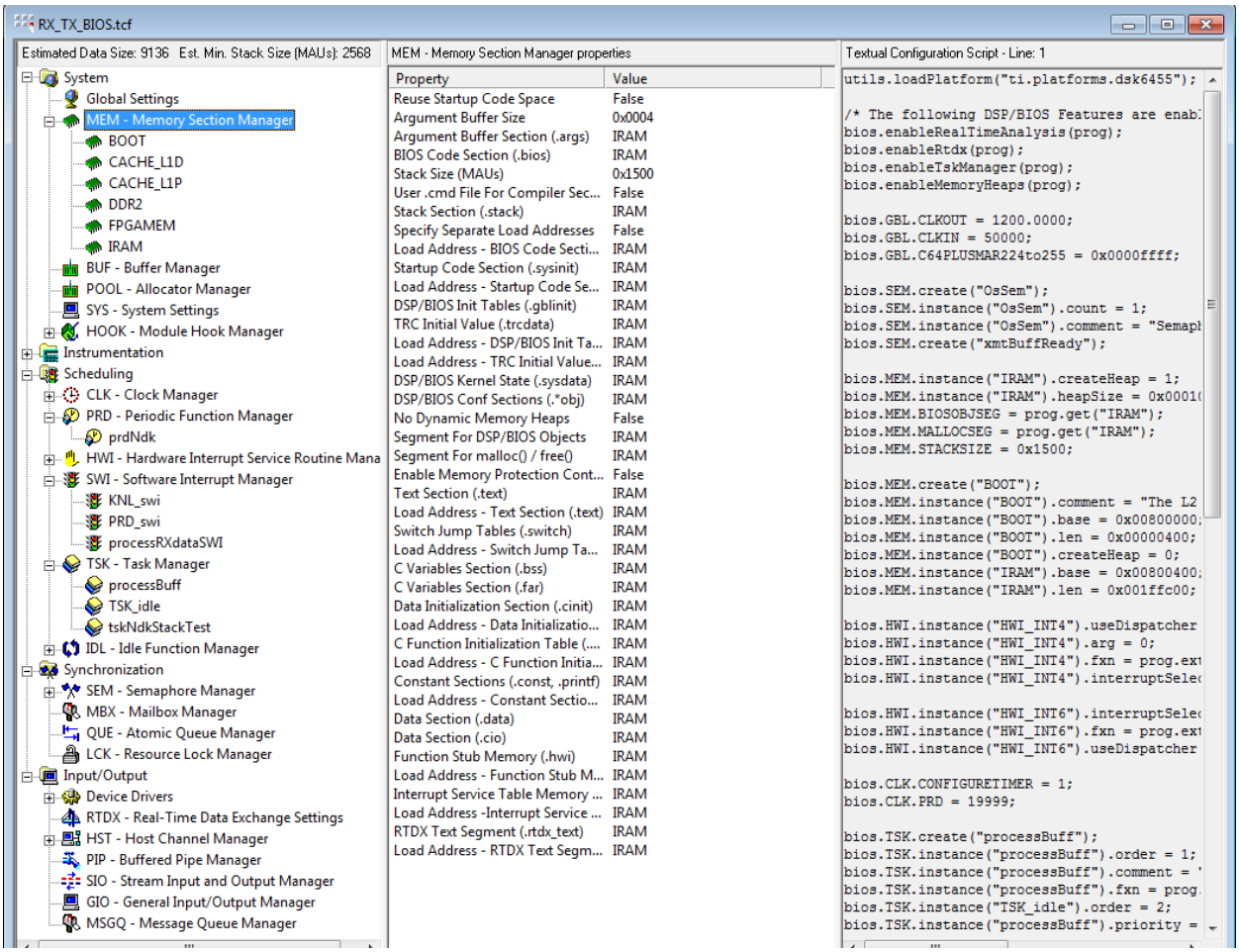

Figure 4.2: Sample of the BIOS configuration GUI

As it can be seen, right side shows what text file has in it, and left side shows different DSP parameters that can be created and configured. Middle part displays current configuration of the selected parameter.

When BIOS is created and compiled, linking file is automatically generated, and user does not need to write their own. Also, when interrupts are configured in the BIOS, the only operation that needs to be done in the main code is to turn interrupt on. If the user does not want to use BIOS, interrupt code and interrupt handler needs to be written separately.

BIOS is a complex software with a lot of capabilities, but only some of those capabilities were used in the project. A full description of the BIOS capabilities can be found in [24]. Reader should also refer to examples provided with the DSK and EDMA3 LLD installation for quick examples on how to use BIOS. A simple diagram describing BIOS structure can be seen in Figure 4.3. BIOS itself is the underlying real-time operating system, which controls five different types of threads: Hardware Interrupts, Software interrupts, Tasks, Periodic Functions, and an Idle Thread. Semaphores, Mutexes, Message Queues and Pipes are used for the data synchronization and exchange between different threads. Idle Thread is not shown in Figure 4.3 because it was not used for this project, but it can be used by user as well.

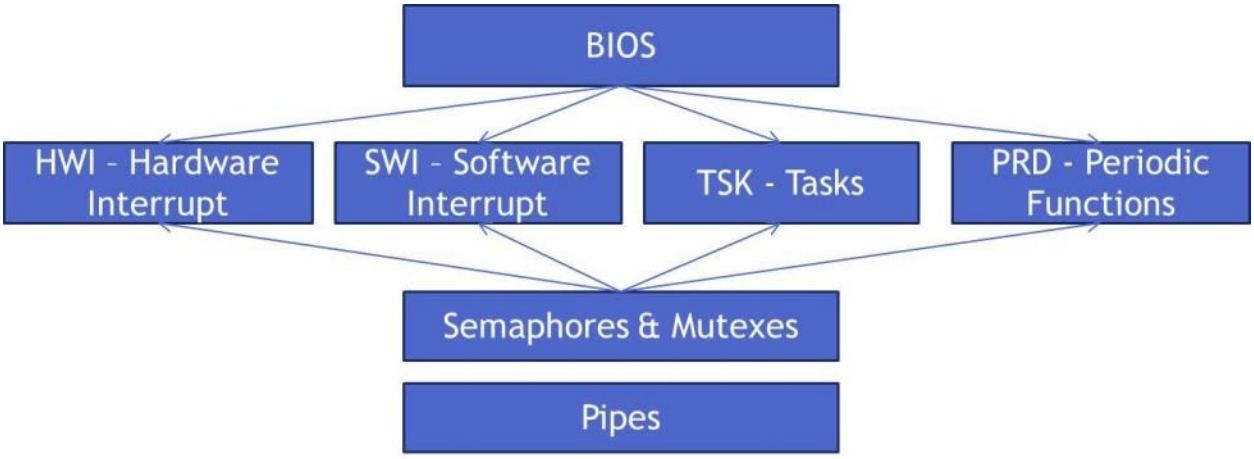

Figure 4.3: Simple BIOS structure diagram

Hardware Interrupts or HWI have the highest priority of all threads in BIOS. They are executed as soon as they are created and cannot be pre-empted using semaphores. *HWI* are triggered by some hardware event (timer, external pin, etc.) and should be used for a very quick critical processing, such that there is a minimal amount of time that other interrupts are disabled. *Software Interrupts or SWI* are the second level of interrupts. Just like *HWI* they are executed as

soon as they are called, and cannot be pre-empted using semaphores, but *SWI* have 15 different priority levels assigned by the user and the only user can start *SWI*. *Periodic Functions or PRD* are a special kind of threads that are triggered periodically based on the timer ticks, and are executed similarly to the *HWI*. Similar to the *HWI,* periodic functions have the highest priority, because the source of the interrupt is the hardware timer. *Tasks or TSK* are lower in priority compared to the *SWI* and *HWI,* and usually run in the infinite loop with pre-emption being done using semaphores. Each *TSK* does not share stack with the rest of the system, and also has multiple levels of priority. At the lowest level of priority is the *Idle Thread or IDL* which runs continuously and every other thread has priority over it. *IDL* thread should contain code that does not require attention and monitors background activity. One such monitoring activity is the useful features of the BIOS to display processor load to the user together with the execution graph, and a cycle counter. These features can be turned on and off from the *Global Settings* menu in the BIOS GUI. *Semaphores* are used for the thread synchronization and work at the very low programming lever. The following pseudo-code describes how each *semaphore* command works.

```
Semaphore POST {
  S = S + 1
}

Semaphore PEND {
     While(S == 0)
  Wait;

     S = S - 1;
}
```

As it can be seen from the pseudo-code when semaphore *PENDs,* program waits until second thread *POSTs* the semaphore signaling to the first thread that resource is ready to be used. Semaphores can be binary (only have a value of 0 or 1) or integer, where instead of incrementing *S,* an integer is added to it. In either case synchronization is achieved between threads sharing same resource.

4.1.2 BIOS Software Packages

As previously mentioned, one of the reasons that TI DSP BIOS was chosen was the availability of the free driver packages that supported the project's platform. The main package that comes

automatically with the Code Composer Studio installation or can be downloaded separately from the TI website is the Chip Support Library (CSL) reference provided in [12]. CSL has all the required functions to initialize and work with the peripherals on the DSP. CSL simplifies configuration and use of the peripherals by accessing required peripheral configuration registers and separating user involvement in explicitly configuring each register directly. CSL provides example programs for each peripheral with its installation which allows easier learning. For the project CSL functions for the EMIF and HPI configuration were used.

In addition to the Chip Support Library, project's DSP board also came with the Board Support Library (BSL) provided by the board's manufacturer, Spectrum Digital. BSL can be downloaded for free from Spectrum Digital website. BSL is responsible for easy control of the peripherals provided by the DSK, such as audio ADC and DAC, LEDs, switches, and for DSP start-up configuration. DSP start-up can be accomplished using GEL file. If the application needs to be bootable though, GEL file needs to be replaced with the start-up function, which BSL provides. DSP initialization function was used in the project from the BSL.

Even though CSL has functions to work with the Enhanced Direct Memory Access ver. 3.0 (EDMA3), a simpler BIOS package was chosen instead. EDMA3 BIOS package is called EMDA3 LLD (Low Lever Driver). It is free of charge and needs to be downloaded separately from the TI website. EDMA3 LLD similarly to the CSL accesses configuration registers, but compared to the CSL it is easier and quicker to set-up and use EDMA3. Also program is easier to read using EDMA3 LLD code. The disadvantage of the EDMA3 LLD is that BIOS needs to be used in the DSP program, which adds large overhead for smaller programs. Reader can refer to [15] for EDMA3 LLD examples and training.

Network interface is the hardest peripheral to configure, and there are no CSL examples on how to do it. The only available resource is the Network Development Kit (NDK) from the TI. Just like EDMA3 LLD, NDK is free of charge and can be downloaded from the TI website. Starting from the version 2.0, NDK no longer has time limitation for the free version. NDK requires use of the BIOS as well. Example programs included with the NDK jump start user quickly into writing simple network software. Reader can refer to [21] for more details about NDK.

The final package that was used for writing DSP software is the IQmath library which can be referenced from [18]. IQmath library is a separate package from the standard Code Composer

Studio installation that has optimized functions to work with the fixed point arithmetic. IQmath library works with signed integers 32-bit in length with varied fixed point length. Results of multiplications are stored in 64-bit long integers, eliminating the need to watch for the overflows. Like other packages used in the project, it is free of charge and can be downloaded from the TI website. In addition to the above four packages TI offers other free of charge packages on their website to work with the signal and image processing, among other things.

4.1.3 DSP Memory Allocation

TMS320C6455 DSP has a 32-bit addressing scheme, where all internal registers, memory, and external devices are mapped to the same address space. Memory mapping for the DSK is shown in Figure 4.4. A partial memory mapping of the internal memory indicated by the address space 0x00000000 to 0x00100000 in Figure 4.4 is shown in Table 4.1. TMS320C6455 DSP has 2 Mbytes of internal memory mapped to the address 0x00800000 to 0x009FFFFF.

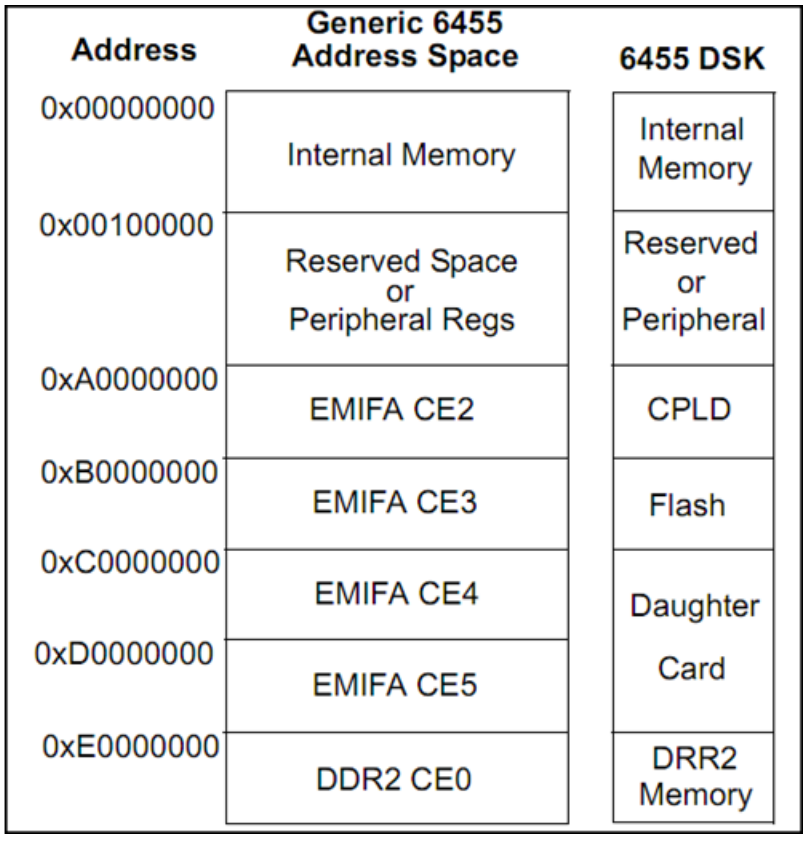

Figure 4.4: TMS320C6455 DSK Memory Map [5] [Fair Use]

Internal memory is where all of the program and data is stored. In the Table 4.1 internal memory is called Level 2 cache or Internal RAM. The internal RAM is divided into 3 more sections to

accommodate boot-loader and space for the data coming from the FPGA. The division and addressing of the internal RAM is shown in Figure 4.5.

By default when DSP comes out of reset, 1024 bytes of data is copied by EDMA from Flash memory into the internal RAM. Most of the DSP programs are larger than 1024 bytes, and a boot-loader needs to be used [20]. As EDMA automatically loads first 1024 bytes from Flash memory when DSP comes out of reset, first 1024 bytes of the internal memory and Flash are reserved for the boot-loader only. As previously mentioned in chapter 3.3.2 HPI port has its own DMA mechanism which directly copies received data to the specified address. If two buffers are created in memory using regular code, after compilation they are positioned anywhere where compiler found free space.

MEMORY BLOCK DESCRIPTION	BLOCK SIZE (BYTES)	HEX ADDRESS RANGE
Reserved	1024K	0000 0000 - 000F FFFF
Internal ROM	32K	0010 0000 - 0010 7FFF
Reserved	7M - 32K	0010 8000 - 007F FFFF
Internal RAM (L2) [L2 SRAM]	2M	0080 0000 - 009F FFFF
Reserved	4M	00A0 0000 - 00DF FFFF
L1P SRAM	32K	00E0 0000 - 00E0 7FFF
Reserved	1M - 32K	00E0 8000 - 00EF FFFF
L1D SRAM	32K	00F0 0000 - 00F0 7FFF
Reserved	1M - 32K	00F0 8000 - 00FF FFFF
Reserved	8M	0100 0000 - 017F FFFF
C64x+ Megamodule Registers	4M	0180 0000 - 01BF FFFF
Reserved	12.5M	01C0 0000 - 0287 FFFF
HPI Control Registers	256K	0288 0000 - 028B FFFF
McBSP 0 Registers	256K	028C 0000 - 028F FFFF
McBSP 1 Registers	256K	0290 0000 - 0293 FFFF
Timer 0 Registers	256K	0294 0000 - 0297 FFFF
Timer 1 Registers	128K	0298 0000 - 0299 FFFF

Table 4.1: Partial TMS320C6455 internal memory map [23] [Fair Use]

This is not a good option, because of the following reasons: addresses assigned will change as the program changes, two consecutive addresses might be separated in such a way that switches would need to be changed between two of them, and finally buffers might not be word-aligned as HPI requires. Because of these reasons, it is essential to assign space for the data received from the FPGA, such that it never changes location with each compilation, and address is always known. The assigned space is located after the boot-loader section and before the rest of the program. For this project it was decided to make space able to hold 2 buffers 256 words long. As each word is 4 bytes, the total space required is 2048 bytes. The rest of the internal RAM is left for the program and data sections.

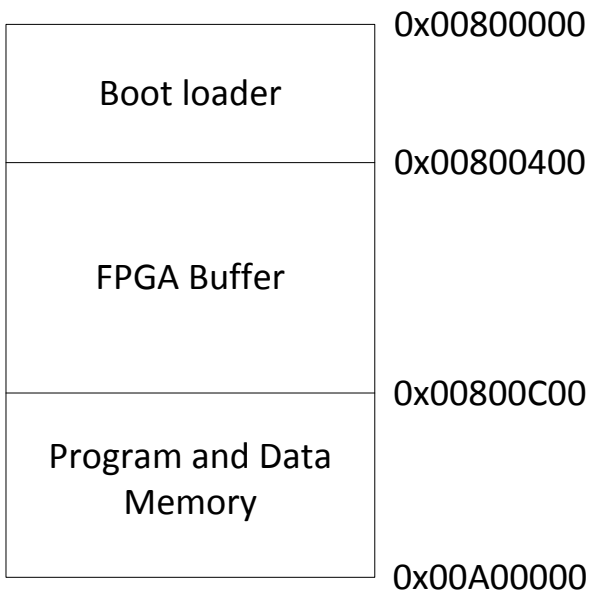

Figure 4.5: L2 memory division

In order for the BIOS to know where to put code and data, described sections need to be specified in the BIOS configuration GUI under *Memory Section Manager*. A separate user linker file needs to be created with boot-loader, and FPGA buffer memory section assigned accordingly. This needs to be done such that code and data are not written by compiler to those sections. User should refer to POST example from Spectrum Digital board on how to do it.

4.2 Transmitter Software

Transmitting part of the DSP software performs the following tasks: symbol mapping, and transmitting symbols to the DAC. Symbol mapping is done using binary sequences for the error calculations, or binary sequences containing data. After symbols are mapped, they are put into a buffer that is transmitted using EDMA to the DAC through the EMIF interface. Double buffering scheme is used, which referred to in a program as PING-PONG buffering. Using double buffering allows one buffer to be filled, while the other buffer is being transmitted to the DAC.

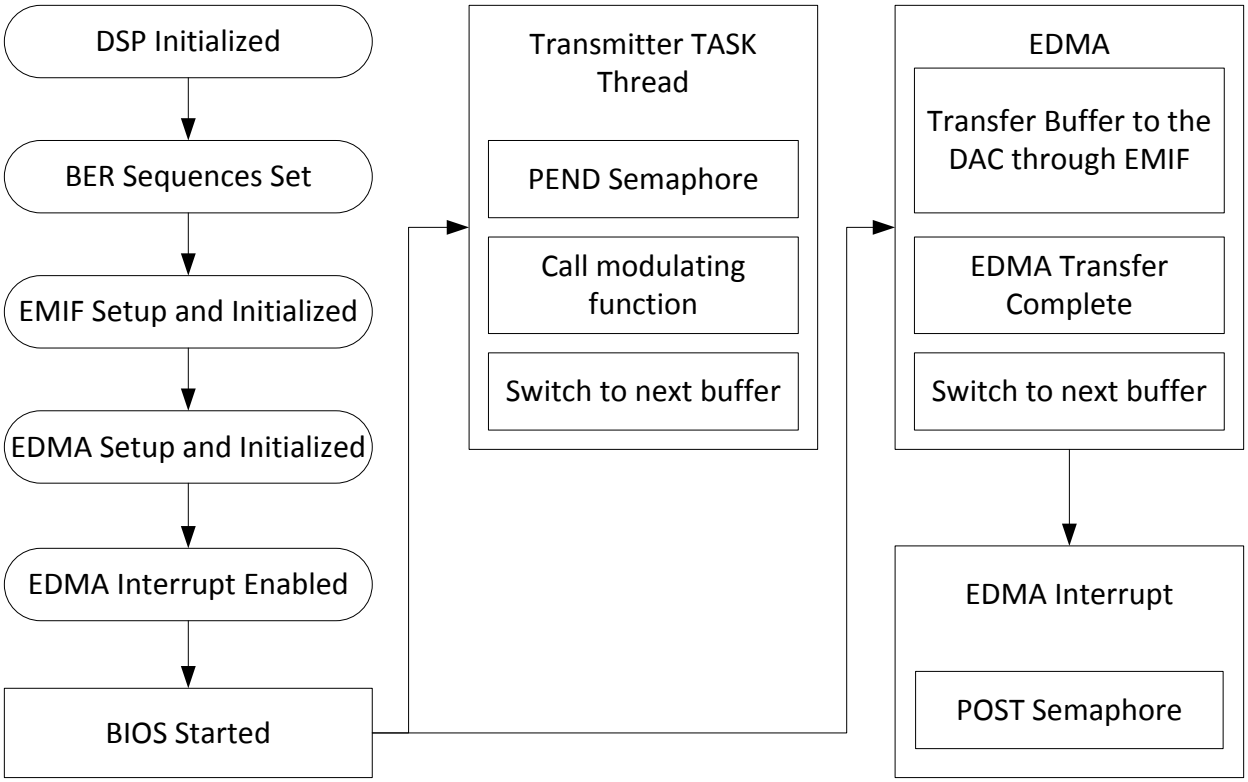

Figure 4.6: Software Flow Diagram for the Transmitter

Transmission of the data to the DAC is much slower compared to the processing rate of the DSP, and because of that it is possible to use this method. Overall software flow for the transmitter is shown in Figure 4.6.

After all of the peripherals required for the transmitter are initialized, BIOS is started and transmitter task is created. As was discussed in the chapter 4.1.1 tasks are lower priority threads that have their own stack. Transmitter could have been implemented using software interrupts as well. Tasks are created usually once with infinite loop inside of them, but they should be pre-emptible and that's why semaphore is present at the beginning of each iteration of the infinite loop. Semaphore is controlled using EDMA interrupt handler. As soon as the buffer transmission is done, interrupt is triggered, unblocking the task and switching buffers for the next transmission. Configuration of the EDMA will be described in the next chapter. Transmitter task calls a helper modulating function that fills buffer with the mapped symbols. At the end of each loop iteration inside of the task, buffers are flipped. This way buffer filled is always opposite of the buffer transmitted by the EDMA.

4.2.1 Enhanced Direct Memory Access (EDMA)

The easiest way to write to the DAC would be as follows:

```
unsigned int* DAC_ADDR = (unsigned int*)0x12345678;
while(1) {
  IQvalue = Calculate a new value;
  DAC_ADDR[0] = IQvalue;
}
```

Problem with this solution is that EMIF port, through which DAC is being written to, does not have a way to report back when the write was finished. Also this solution better fits a non-BIOS enabled program. Because there is no way for the EMIF port to report back, an infinite while loop cannot be used inside of the task thread as it cannot be pre-empted. Even if this solution is used in a non-BIOS program, writing to the EMIF port is much slower than execution speed of the DSP. This means that while value is being written to DAC, DSP is wasting cycles waiting to execute next instruction. To solve this problem a dedicated memory transfer transparent to the processor is needed, which direct memory access provides.

Texas Instrument's DSP direct memory transfer is called EDMA. EDMA is a separate unit on the DSP that performs transfers between external peripherals and internal RAM without involvement of the main processor. Detailed description of the DMA can be found in [14]. DMA consists of the 64 regular channels and 4 "quick" DMA (QDMA) channels. QDMA channels are quicker to configure, and need manual triggering. QDMA channels are more suitable for the memory to memory copying, rather than peripheral to memory copying. For this project a regular DMA channel was used. Each DMA or QMDA channel is configured using a set of parameters. Some of the important parameters are source/destination address, source and destination index, number of elements, synchronization type, and event trigger.

Event trigger is what causes DMA to start transferring data. QDMA is always triggered manually by user, but for the DMA events event trigger can be a timer interrupt, an external pin interrupt, or a manual trigger as well. Amount of data to be transferred is defined using 3 parameters ACNT (Element), BCNT (Frame), and CCNT (Block). ACNT is the number of elements in bytes; BCNT is the number of elements in a frame, and CCNT is the number of frames in a block. Figure 4.7 shows an example of memory organization. An array of 16 squares represents some section of a memory where each square is a byte for example. If 8 bytes need to be transferred they can be packed into a block of 2 frames, so the ACNT = 1 (1 byte per

element), BCNT = 4 (4 elements per frame), and CCNT = 2 (2 frames per block). This is what is shown in example in Figure 4.7. It is also possible to just make ACNT = 8 (1 element = 8 bytes), BCNT = 1, and CCNT = 1. Even though the size of the transfer does not change, its organization does. It makes a difference when it comes to the type of transfer required, and location of the transferred data.

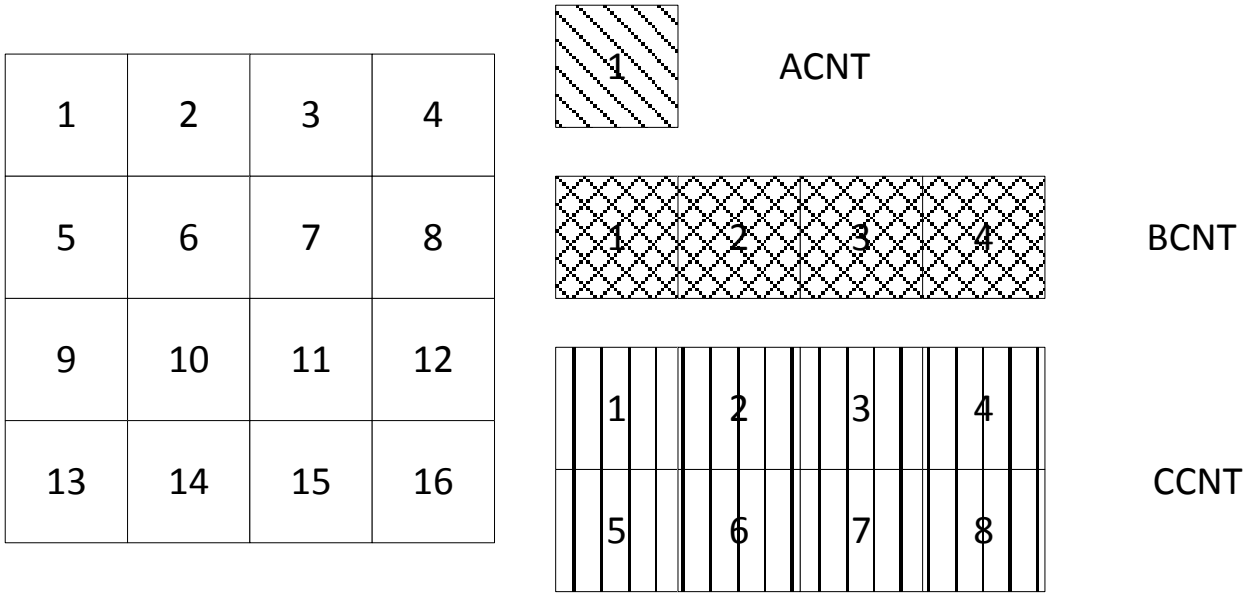

Figure 4.7: DMA memory example

For example, if the peripheral accepts only one byte at a time, then ACNT needs to be 1, and BCNT can be 8. If it's a simple memory transfer then it does not matter. In addition to the ACNT, BCNT, and CCNT, there are also parameters such as source/destination B and C indexes. B and C indexes control how data is copied. For example, if the locations 1, 5, 9, and 13 in Figure 4.7 needed to be copied, than source B index would be 4, which means jump 4 memory locations every copy. Similarly destination B and C indexes can be adjusted as well.

There two types of the transfer synchronizations used by the DMA. An A-synchronized (A-sync) transfer requires event trigger to happen every time ACNT bytes are transferred. An AB-synchronized (AB-sync) transfer only needs one event trigger per whole ACNT*BCNT bytes. A-sync transfer would be suitable if the peripherals can only accept one element at a time, and sends an event after that. An AB-sync is more suitable for memory transfer, where once the transfer started, it will be finished completely. It is also possible to make ACNT equal to ACNT*BCNT and use A-sync as well.

Finally there is another important parameter for the project in the DMA configuration: linking address. When DMA finishes transfer, it is desirable to start a new transfer somehow while the next buffer is being filled, thus achieving a double buffering. For this exact reason, a parameter set has an option of including a link address. Link address allows DMA channel to get reconfigured with a new set of parameters after transfer is finished. In order to achieve double buffering two full set of parameters need to be configured. Parameter set one needs to link to parameter set two, and vice versa.

For this project two set of parameters were created. Destination address for both of them was the EMIF port, source address was two separate buffers (ping and pong buffers). ACNT = 4, as each element is a 32-bit word, BCNT = 1200 the size of the buffer, and CCNT = 1. Elements are located sequentially, which means source and destination B and C index = 1. Transfer synchronization is set to AB, such that only one event is used to trigger full buffer transfer. Timer 0 is used for the event trigger. In order to make transmission of the data to the DAC look continuous event trigger needs to be synchronous with the buffer transmission. Buffer length is 1200 words, and the EMIF clock rate is 12 MHz, which means it takes 0.0001 seconds for buffer transfer to complete. This means that an event trigger needs to happen at the rate of 10 kHz. Timer 0 is setup in *Clock Manager* under *Scheduling* section of the BIOS GUI. Timer 0 is also used as a tick reference for the BIOS scheduler. Timers on the DSP use as a reference a 200 MHz clock [23]. In order to achieve correct interrupt rate, Timer 0 is setup in 32-bit unchained mode, with period register equal to 19999. This is because timer interrupt rate according to the manual [17] is

$$f_{INT} = \frac{Timer\ input\ clock\ rate}{Programmed\ timer\ period + 1} = \frac{200\ MHz}{19999 + 1} = 10\ kHz$$

An extra cycle is needed to reload timer register with the period value. Before EDMA starts using Timer 0 as the event trigger first transfer is initiated manually during EDMA initialization.

During design process it was discovered that when EMIF works in continuous mode like it was presented with the pseudo-code at the beginning of the chapter, generated clock has gaps in it. This gaps cause the transmitted data to be incorrect. Problem was addressed to the Texas Instruments and they found a solution. In order for the gaps to disappear value 0x80000753 needs to be written to the reserved register at address 0x7000000C. This register is referred in the code as the EMIF_PAUSE_REG. Even though EMIF access in the final project is not

continuous, it was decided to keep the solution present in the final code to avoid possible problems.

4.2.2 Modulating signal

Symbol mapping is done inside of the *modulate* function called by the transmitter task. Inside of the *modulate* function bits are mapped to symbols using a "for" loop. Binary data for mapping is represented using D x L array where D is the number of bits per symbol and L is the length of the array. For the bit error rate testing in this project L = 63 and D = {1, 2, 3, 4}. If the data needs to be transmitted, D can be one of four values depending on the modulation, and L varies with the size of the data to be transmitted. At the symbol rate of 1 MSym/s and a DAC clock of 12 MHz, each column of the binary D x L array is mapped to 12 symbols. For linear modulations such as M-PSK, M-ASK, M-QAM, and M-PAM those 12 symbols are all the same, creating a square pulse. For the M-FSK and MSK modulations pulse shape is not a square wave and 12 symbols represent a half sinusoid for MSK and different period sinusoids for the M-FSK. For modulations where quadrature channel needs to be delayed by half-symbol with the respect to in-phase channel, such as MSK and O-QPSK, additional variable is introduced. The variable makes sure I and Q channels are modulated half-symbol apart. This is achieved by mapping binary data 6 symbols at a time, rather than 12 symbols.

In order to optimize symbol mapping code, a technique called loop unwrapping is used in addition to the optimization done by compiler. The length of one DMA buffer is 1200 symbols and there could be 2 methods to map symbols. Method 1 maps symbols one at a time:

```
For(i = 0; i < 1200; i++) {
  dma_buffer[i] = IQsymbol_value;
}
```

Method 2 maps symbols twelve at a time:

```
For(i = 0; i < 1200/12; i++) {
  dma_buffer[i]    = IQsymbol_value;
  dma_buffer[i+1]  = IQsymbol_value;
  …
  dma_buffer[i+11] = IQsymbol_value;
}
```

Using method 2 is much more efficient, because information assigned to the buffer is redundant, and less iterations decrease load on the processor. For comparison Table 4.2 provides measured processor load without compiler optimization before and after described loop unwrapping. It can be clearly seen that loop unwrapping helps a lot. After compiler optimization processor load drops even more.

Modulation	Method 1: No loop unwrapping	Method 2: Loop unwrapping used
BPSK, BASK	61.5 %	20.5 %
QPSK, 4-ASK	70 %	25 %
16-QAM, 16-APSK	85 %	45 %
FSK, MSK	95 %	50 %

Table 4.2: Comparison of processor load before and after loop unwrapping

When symbols are mapped user needs to keep in mind that DAC accepts symbols in different non-intuitive format. The mapping is described by equation (2.1). Looking at equation (2.1) it can be seen that in order to get a maximum positive value input needs to be 0, and to get maximum negative value input needs to be 1023. This means that input is inverted and has a DC offset. Good way to convert decimal values for mapping to the DAC values is to follow the algorithm (example using 8-PAM mapping):

```
Symbols = [-7 -5 -3 -1 1 3 5 7]/7 = [-1 -0.714 -0.429 -0.143
0.143 0.429 0.714 1]

Round([-1 -0.714 -0.429 -0.143 0.143 0.429 0.714 1]*512) +
512 = [0 146 293 439 585 731 878 1024]

Reverse order and make sure values are between 0 and 1023:
= [1023 878 731 585 439 293 146 0]
```

Using this algorithm decimal values for other modulations can be easily converted as well. Maximum energy of any symbol needs to be normalized to unity in order for it to be correctly received later.

4.3 Receiver Software

Receiver part of the DSP software is responsible for processing data after pre-processing performed by the FPGA hardware. Receiver takes decimated and filtered values from the FPGA and removes frequency and phase offset using decision directed digital phase locked loop (DD-

PLL). As the PLL requires decision making, symbols are also demodulated and converted to bits at the same time. Symbol decisions are made using least distance approach. After symbols are demodulated they are compared to the expected bits and error is calculated and sent to the PC. If the bit error analysis is not performed, then information is recovered and sent back to the PC. Receiver software is implemented using software interrupt thread. Receiver software could be implemented using a task similar to the transmitter software. The reasoning for using software interrupt was that software interrupt has a priority over tasks.

Every time FPGA finishes filling the memory buffer, interrupt is triggered, which causes DSP to create and run software interrupt thread.

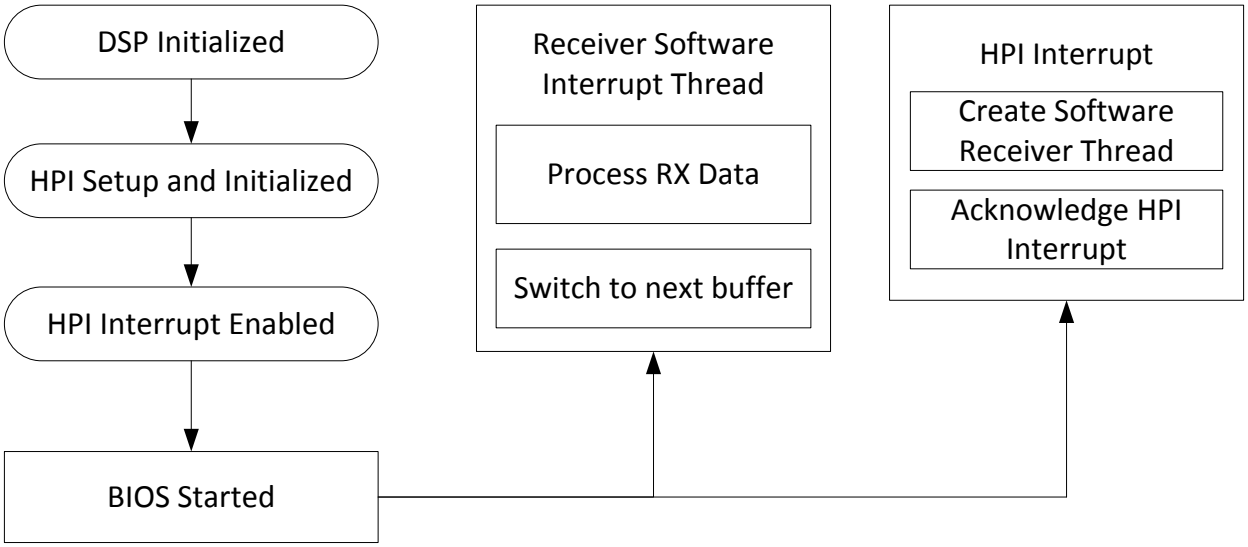

Figure 4.8: Software Flow Diagram for the Receiver

Before HPI interrupt handler exits, HPI interrupt is acknowledged to restart FSM in the FPGA. If at any moment user wishes to disable receiving software, a flag is set, and next time HPI interrupt handler is called, interrupt is sent to the FPGA, which stops the FSM. At the end of the software interrupt thread, buffers are switched for next iteration. Overall software flow for the receiver software is shown in Figure 4.8.

The incoming data to the DSP is in the Q0.9 notation, however, inside of the receiving software all of the calculations are done in the Q2.13 notation. Incoming samples are first shifted to the left by 6 bits to convert Q0.9 to Q0.15 notation, and the result is divided by 4 using 2 bit shift to the right to get a Q2.13 notation. The reason it needs to be done in 2 steps is for the sign extension reasons. If the incoming samples are just shifted to the left 4 bits, then a negative

integer will not be represented correctly. Q2.13 notation was chosen to eliminate risk of the overflow. Q0.15 notation can represent fixed point numbers between $[1-2^{-15},-1]$. If two of these numbers are added together, they might overflow, because DSP represents complex numbers using 32-bit register with 16-bit dedicated to real part, and 16-bit dedicated to imaginary part [11]. To make sure result of any operation never overflows it was decided to add another bit to the integer portion increasing workable range to $[4-2^{-13},-4]$.

4.3.1 Decision Directed Phase Locked Loop

One of the primary functions in the receiver is the frequency and phase error recovery for the coherent modulation. It is possible to demodulate M-ASK, M-FSK, and M-DPSK modulations using non-coherent methods. Performance drops this way by 3 dB for binary modulations in some cases, and even more for larger constellations. Therefore it is more efficient to demodulate data using coherent method. Coherent method requires locking onto the exact phase and frequency of the carrier, which is achieved using a Phase Locked Loop (PLL).

In general PLL works by passing phase error through the loop filter which controls either VCO (Voltage Controlled Oscillator) in the analog PLL, or NCO (Numerically Controlled Oscillator) in the digital PLL. For this project two different types of the PLLs were used.

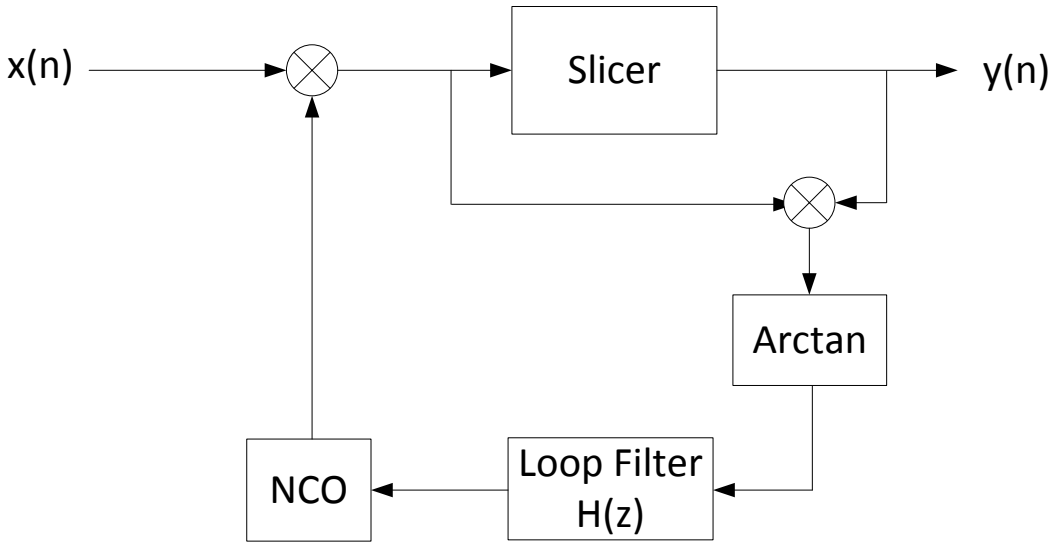

Figure 4.9: Decision Directed PLL Block Diagram

Decision directed PLL is presented in Figure 4.9 and can be referenced from [25]. In the decision directed PLL phase error is calculated based on the estimated received symbol, and the actual

received symbol. For the 8-PSK modulation N-power PLL was used, because it showed better performance. The structure of this PLL is shown in Figure 4.10. In the N-power method PLL, phase error is calculated by taking incoming signal to the 8^{th} power. When a complex input signal is taken to the 8^{th} power, modulation is removed and a pure carrier can be recovered. The recovered carrier has the N-times larger frequency, and needs to be divided by N. When the actual carrier is recovered, it is passed through the loop filter and tracked by PLL as well. The drawback of the N-power method is that under presence of noise performance is worse. The advantage of the N-power method is that, because is locks onto the carrier there is less chance of the phase ambiguity. Note that for

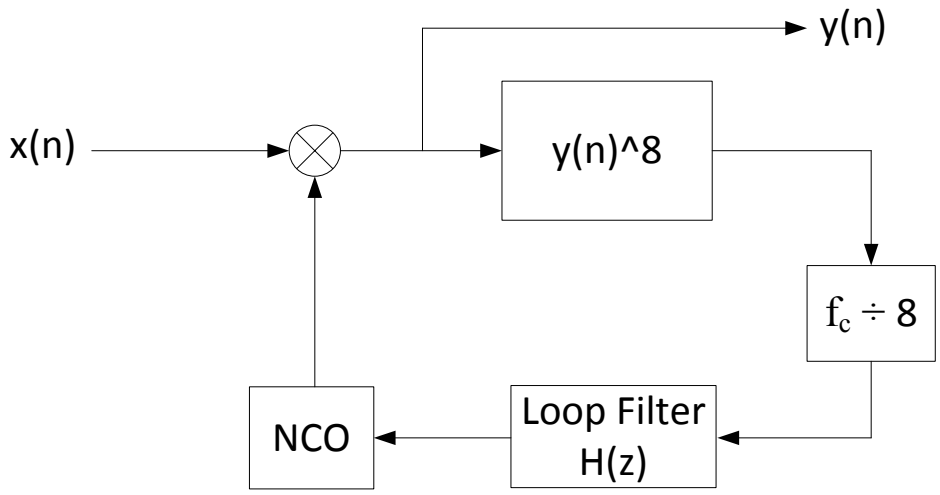

Figure 4.10: PLL for the 8-PSK modulation

the M-PSK modulations the incoming signal looks like:

$$y(n) = e^{j\left(2\pi f_c t + \frac{2\pi N}{M}\right)} \quad N = (0, \dots, M-1) \quad M = \{2, 4, 8, \dots\}$$

As there is no magnitude, taking M-PSK signal to the 8^{th} power is just multiplying input phase by 8. After that phase is reduced to the range $(-\pi, \pi)$ and divided by 8. Division and multiplication by 8 are just bit shifts. With this modification N-power method PLL becomes what is shown in Figure 4.11. When PLL is synchronized reduction step is bypassed and PLL is just as efficient as the DD-PLL in Figure 4.9.

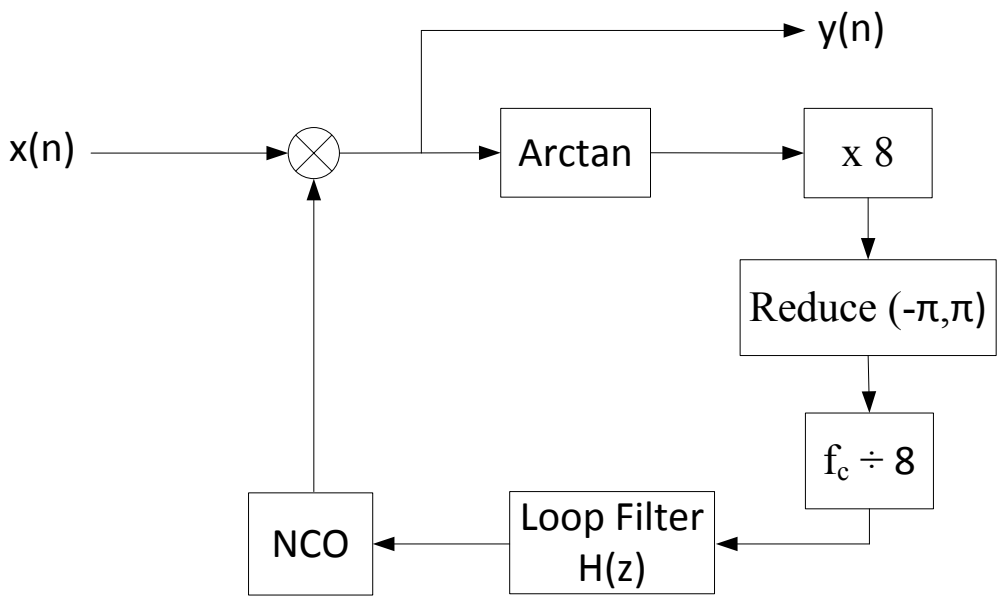

Figure 4.11: Simplified PLL for the 8-PSK modulation

4.3.2 Loop Filter Design

Loop filter needs to filter the error signal with the additive white noise on it. If the loop filter bandwidth is large, synchronization happens quickly, but PLL is very sensitive to noise. With the small filter bandwidth, synchronization is slower, but PLL is more prone to noise. Loop filter bandwidth also needs to be large enough such that possible frequency offsets can be compensated.

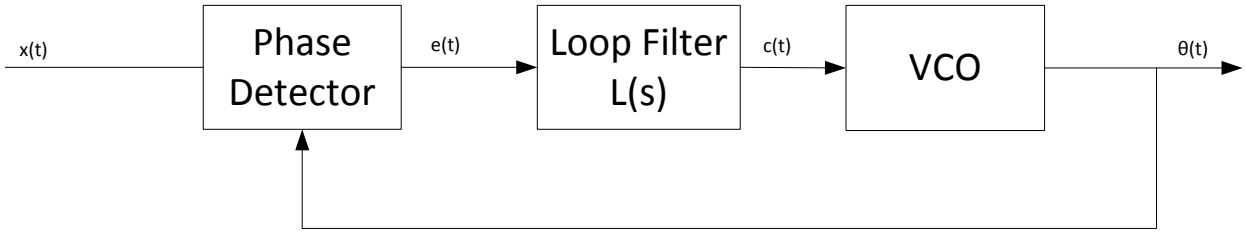

Figure 4.12: General structure of the continuous time PLL

General structure of the continuous time analog PLL is shown in Figure 4.12. Ideally phase detector should produce error signal:

$$e(t) = \theta(t) - \hat{\theta}(t)$$

where $\theta(t)$ is the phase of the incoming signal and $\hat{\theta}(t)$ is the estimated phase, and VCO is just an integrator with some gain that can be absorbed by the loop filter. Taking everything to the Laplace domain to determine transfer function, error signal becomes $E(s)$, loop filter is $L(s)$, and VCO has a transfer function $1/s$. After manipulations transfer function of the whole system becomes:

$$H(s) = \frac{L(s)}{s+L(s)} \tag{4.1}$$

The error transfer function is then:

$$E(s) = \Theta(s) - \widehat{\Theta}(s) = \Theta(s) - E(s) * L(s) * \frac{1}{s} \rightarrow E(s) = \frac{s*\Theta(s)}{s+L(s)}$$

To find the steady state error for the PLL, Laplace final value theorem can be applied:

$$\lim_{t\to\infty} e(t) = \lim_{s\to 0}\ s * E(s) = \frac{s^2\Theta(s)}{s+L(s)}$$

If the input signal has only a phase offset $\theta(t) = Cu(t)$, Laplace transform will be $\Theta(s) = C/s$, and the steady state error of the PLL will be:

$$\lim_{s\to 0} \frac{s*C}{s+L(s)} = 0$$

Therefore for phase only error, $L(s)$ can be just a proportional filter and a zero steady state error will be achieved. A more likely case is that input phase error will have a frequency offset as well $\theta(t) = 2\pi f_\Delta tu(t)$, with the Laplace transform $\Theta(s) = 2\pi f_\Delta/s^2$. Given this conditions, a steady state error then becomes:

$$\lim_{s\to 0} \frac{2\pi f_\Delta}{s+L(s)} = \frac{2\pi f_\Delta}{L(s)}$$

This means that with proportional only filter, steady state error will increase as the frequency increases. If the loop filter has a form of $L(s) = K * \frac{s+\tau_2}{s+\tau_1}$ steady state error becomes:

$$\lim_{s\to 0} E(s) = \frac{2\pi f_\Delta * \tau_1}{K*\tau_2}$$

The only way to achieve a zero steady state error with the frequency offset input is to have $\tau_1 = 0$, which makes $L(s) = K * \frac{s+\tau_2}{s}$. This type of filter is called proportional-integral filter. A

higher order filters can be used, but proportional-integral filter is a very common filter for the second-order PLL. More information about continuous time PLL can be found in [29].

Design of the proportional-integral filter was done experimentally. The goal was to make loop filter be able to synchronize to all possible frequency offsets, with the minimum amount of loop bandwidth to minimize effects of the noise. Receiver and transmitter internal oscillator has an accuracy of ±75 ppm. With this accuracy expected frequency offset can be as large as 112 kHz. From observations it was determined that a frequency offset of 75 kHz is typical. For tightly packed modulations such as 16-QAM, a frequency offset of 75 kHz is hard to recover, and IF filter needs to be adjusted as well. In order to create a filter, first a VCO transfer function of $1/s$ has been created. After that a MATLAB *sisotool* was used with automatic PID tuning. PI was selected as a type of the filter with test bandwidth and a phase margin of 45°. The resulting complete transfer function $C(s)/s$ was converted to the discrete time using bilinear transform with the following mapping:

$$s = \left(\frac{2}{T}\right) * \frac{z-1}{z+1}$$

where T is the sampling period [30]. T = 1 μsec for a data rate of 1 MSym/sec. Bilinear transform converts a stable continuous time system to a stable discrete time system. There are other types of transformations from continuous time to the discrete time, but they might not be as reliable. When the complete discrete time system was calculated, it was implemented in the DSP with the help of IQmath library functions. The resulting implementation was tested against BPSK signal with different frequency offsets. Experimentally it was determined that the minimum filter bandwidth to achieve 75 kHz frequency offset synchronization is 7500 Hz. Final resulting discrete time filter including NCO transfer function (which is just an integrator) is:

$$H(z) = \frac{0.01705 + 0.000785 * z^{-1} - 0.01627 * z^{-2}}{1 - 2 * z^{-1} + z^{-2}}$$

Inside of the DSP the following filter is implemented as a Direct Form I filter, which means all delays in the transfer functions are a separate memory location, and all coefficients are directly copied from transfer function.

It is possible to design loop filter directly in the discrete time using same *sisotool* in MATLAB replacing a VCO integrator $\frac{1}{s}$ with the discrete form $\frac{1}{1-z^{-1}}$. This method was not used however, because majority of the literature about PLLs use continuous time for coefficients

calculation. As the original filter design was based on that literature, bilinear transform was a preferred method.

4.3.3 Slicer

One of the important parts of the PLL in Figure 4.9 as well as the whole receiver is the module called slicer. Slicer uses the least distance calculations to determine the most likely symbol that was transmitted. Expression (4.2) below shows how it is done mathematically.

$$\hat{x}(n) = \min\left[\sqrt{\{y(n)_I - \overrightarrow{\boldsymbol{x_I}}\}^2 + \{y(n)_Q - \overrightarrow{\boldsymbol{x_Q}}\}^2}\right] \tag{4.2}$$

Incoming symbol $y(n)$ is compared against all the possible symbols in the constellation $\vec{\boldsymbol{x}}$ and the most probable symbol $\hat{x}(n)$ from $\vec{\boldsymbol{x}}$ is used. A square root operation in the expression (4.2) is very resource intensive compared to the squaring, and can be left out. As square root operation is needed to get the exact magnitude of the difference, it is sufficient to just use minimum of the magnitude squared. Complex subtractions and multiplications were done using dedicated assembler functions in the DSP, which can be found in [11].

For constellations larger than 8 symbols slicer function uses expression (4.2) without the square root inside of the loop. For smaller modulations up to the order 4, using a loop and expression (4.2) becomes inefficient. For those modulations (BPSK, BASK, 4-ASK, QPSK, O-QPSK) a simple if-else statements are used where incoming symbol $y(n)$ is compared against particular region. For ASK modulations, a non-coherent detector was used for the PLL symbols, but a coherent detector was used for the actual symbol decoding. Coherent ASK detection is better in terms of error performance than a non-coherent detection, but non-coherent detection uses both I and Q channels for symbol estimation. Because of that, PLL synchronizes quicker and more reliably with the non-coherent detector, but actual symbols are decoded after synchronization coherently.

In order to get bits out of the decoded symbols efficiently, along with the possible array of symbols $\vec{\boldsymbol{x}}$, additional array of integers $\vec{\boldsymbol{b}}$ was created. A partial content of the arrays for the QPSK modulation is shown in the Table 4.3 below. This way when slicer determines correct symbol, an integer is taken from array $\vec{\boldsymbol{b}}$ and sent back as well.

Index	Array $\vec{x}$	Array $\vec{b}$
2	0x16A016A0	0
3	0x16A0E95F	2
4	0xE95F16A0	1
5	0xE95FE95F	3

Table 4.3: Partial content of the arrays in the slicer function

In order to recover binary data from the returned integer, a mask with the right shift is needed. For example, following pseudo-code shows how returned integer from 16-QAM modulation is converted to bits:

```
rxInt = 13;
bit1 = rxInt & 0x1;
bit2 = (rxInt & 0x2) >> 1;
bit3 = (rxInt & 0x4) >> 2;
bit4 = (rxInt & 0x8) >> 3;
```

4.4 Error Testing and Data Communication

The main goal of this project form the beginning was to be able to determine bit error rate performance of different modulations using different antennas. As the system was developed further it was beneficial to also add a data transmission capability if the system would be used for other purposes besides testing of the antennas. For this reason two different protocols were developed. Both of the protocols are described in detail in the sections that follow.

4.4.1 Error Testing Software

There are many ways an error testing can be performed. In essence bit error testing is performed by comparing received bits against a known transmitted sequence. For this project a bit sequence needed to satisfy couple conditions: sequence needed to be long enough such that resulting spectrum does not have visible spectrum lines; an equal amount of zeros and ones needed to be present to avoid DC offset and to make a valid spectrum; all constellation points should be transmitted for valid error result, and sequence should be fairly easy to synchronize to.

One of the binary sequences that satisfies most of the conditions is the maximal length sequence, abbreviated m-sequence. All m-sequence belong to a class of binary sequences called pseudo random binary sequence or PRBS. They are called PRBS because they have properties

similar to an infinitely long truly random binary sequence, but there is a known way to generate them, and they are finite in length. M-sequences are generated using linear feedback shift register (LFSR). An example of such register is shown in Figure 4.13 and in Figure 4.14. An LFSR in Figure 4.13 is known as a Fibonacci LFSR. Using Figure 4.13, Fibonacci LFSR bits are output on the right, and the shift operation is to the right as well. The numbering of the registers is from left to right. In Galois LFSR from Figure 4.14, bits are also output on the right, and shift operation is to the right, but numbering of the registers is reversed, and is from right to left [26].

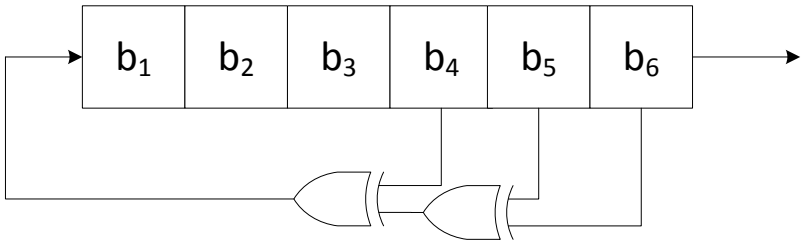

Figure 4.13: 6-bit Fibonacci LFSR

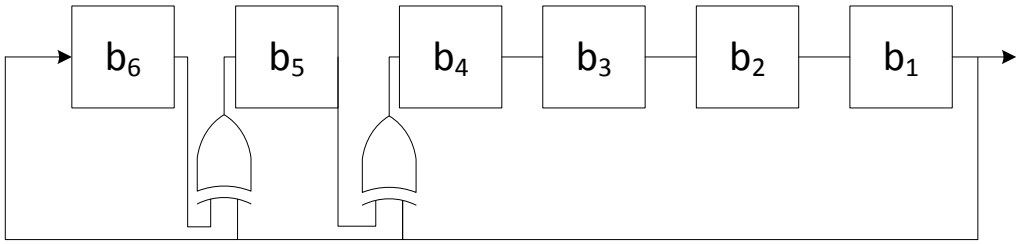

Figure 4.14: 6-bit Galois LFSR

Galois LFSR is advantageous over Fibonacci LFSR in terms of implementation either in hardware, such as FPGA, or in software. Galois LFSR performs XOR operations in parallel, which is advantageous in terms of data path for FPGA, and is easier to implement in C programming language. Location of the taps that create m-sequence is called generator polynomial. There are limited number of the generator polynomials for each LFSR length that create m-sequence. These polynomials can be found in [26].

The length of the m-sequence is $N = 2^B - 1$, where B is the number of register bits. The initial condition of the LFSR can be anything but all zeros, because that would not create any data. M-sequences have couple properties that are very useful. The autocorrelation of the m-sequence is shown in Figure 4.15. The value of the autocorrelation is 1 when the sequence is aligned properly, and $-\frac{1}{N}$ when it is not aligned. Therefore the longer the m-sequence the more

ideal autocorrelation function is. M-sequence contains exactly $\frac{N}{2}$ ones, and $\frac{N}{2} - 1$ zeros. When the signal is modulating repeating m-sequence, the output spectrum will follow the pulse shape outline, with spectral lines located at the interval $\frac{R_S}{N}$. An example of this can be visible in Figure 4.16. When the sequence is long enough, theses spectral lines become invisible at the spectrum analyzer output.

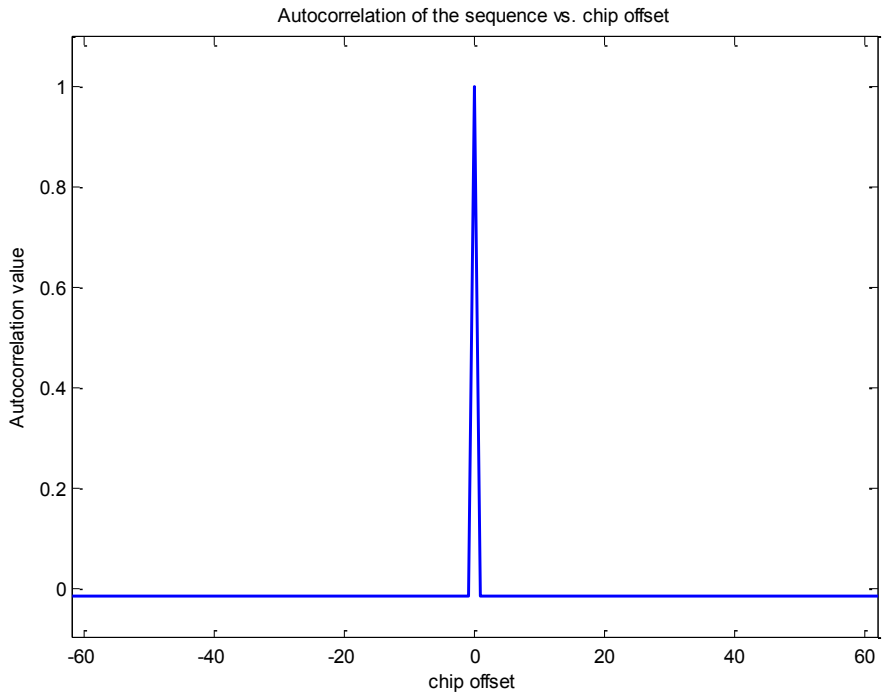

Figure 4.15: Autocorrelation of the m-sequence

In order to test bit error rate performance transmitter software modulates a repeating m-sequence. For larger constellations (M-PSK, M-QAM, M-ASK, etc.) each constellation symbols is modulated using multiple m-sequences of the same length, made using same set of taps, but from different initial conditions. Different initial conditions create a phase shift for each m-sequence which introduces more randomness allowing mapping of all symbols in constellation. An example of this is demonstrated using Table 4.4. For the project the length of the m-sequence was chosen to be 63 bits long. There are 3 generator polynomials that create m-sequence 63 bits long, and polynomial $X^6 + X^5 + X^4 + X + 1$ was chosen. This means that taps 6, 5, 4, and 1 are used in the feedback. An m-sequence of 63 bits long is long enough such that spectral lines are very close together, and short enough that it is quickly synchronized.

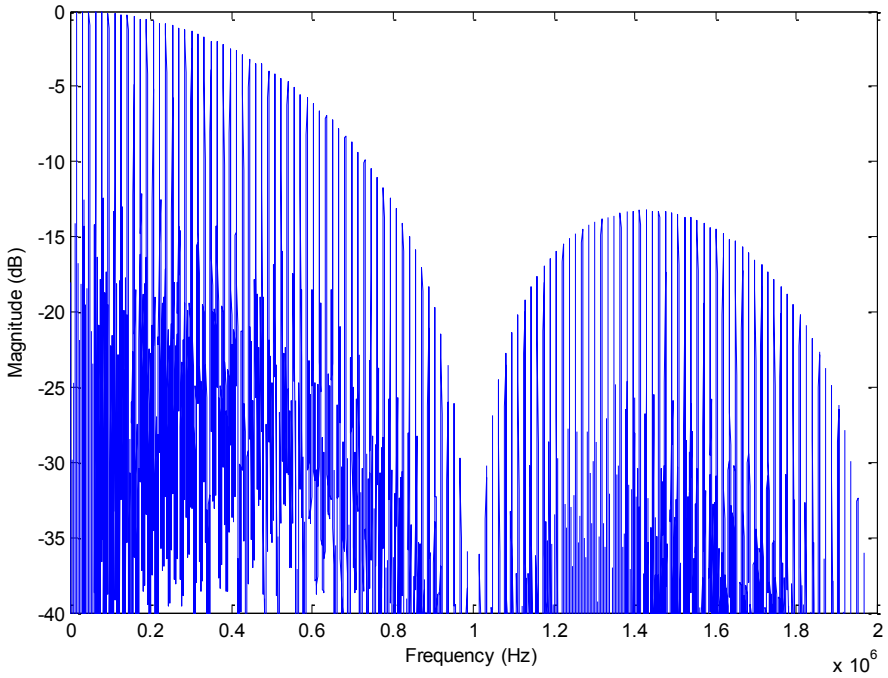

Figure 4.16: FFT Spectrum of the m-sequence modulated signal

M-sequence phase 1	0	1	0	1
M-sequence phase 2	1	0	1	0
M-sequence phase 3	0	0	1	0
M-sequence phase 4	1	0	0	1
Resulting Integer symbol	5	8	6	9

Table 4.4: Example of mapping m-sequence for larger than binary constellations

In the receiving software received symbols are broken down into bits as described in chapter 4.3.3. Received bits are independent of each other, and up to 4 separate m-sequence generating LFSRs are running at the same time synchronizing to the received bits. As the receiver does not know at what position of the m-sequence it started receiving, all possible phases are correlated until lock is achieved. The following pseudo-code shows how the receiving m-sequence is correlated:

```
Correlate receiving bits against replica m-sequence
If Correlation == 0 or 63 ± Threshold
  Compare every next received bit against current m-sequence
  value
Else
  Reset correlation value
  Increment initial condition
  Repeat
```

Note couple of important things. First, autocorrelation value for m-sequence as shown in Figure 4.15 is valid when m-sequence is represented using mapping $+1 \rightarrow 0, -1 \rightarrow 1$, and multiplication is used. When the m-sequence is binary, multiplication becomes XOR operation in correlation, and autocorrelation function is 0 when sequence is perfectly aligned, and $\frac{N}{2} - 1$ when sequence is not aligned. Hence the value being looked for is not +1, but rather 0. Second, when inverted m-sequence is correlated against non-inverted sequence the expected results flip. This happens in symmetrical constellations (M-PSK, and M-QAM) where phase ambiguity exists, and constellation rotates as it being received. Finally, when errors are present the cross-correlation function of the m-sequence with errors against clean m-sequence look something like Figure 4.17. The peak of the cross-correlation is no longer +1 (or 0 for binary), and therefore some threshold is needed to acquire lock.

As soon as the lock is acquired every bit coming out from local m-sequence is compared against received bit, and any errors can be detected this way. As a safety feature against extra errors, m-sequence lock is constantly tracked as the bits are compared, and if correlation value is not valid any calculated errors are discarded. This is done such that if constellation rotates due to phase ambiguity in the middle of bit comparison, extra errors are not introduced by it.

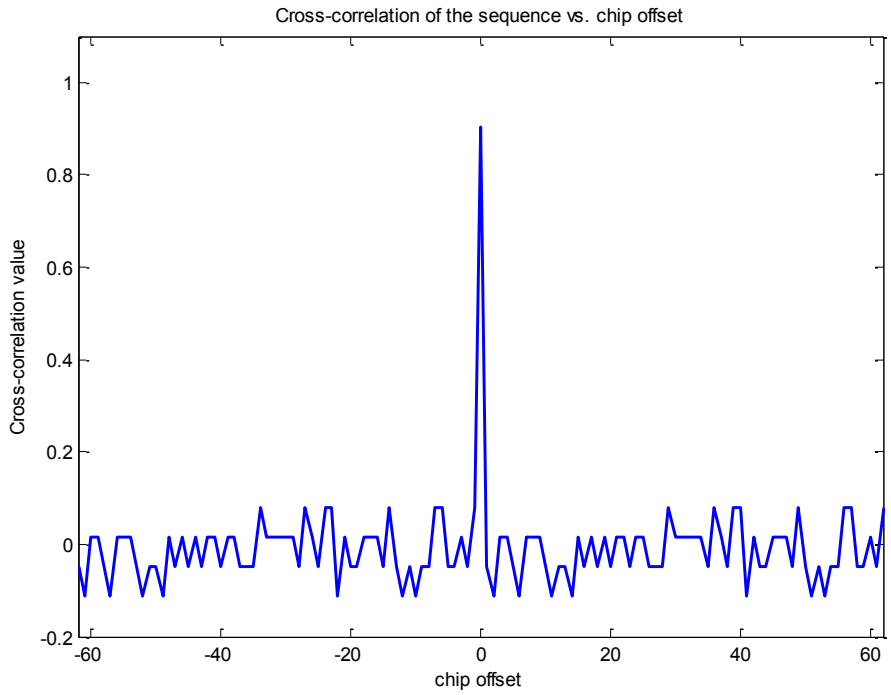

Figure 4.17: Cross-correlation of erroneous and correct m-sequence

The pseudo-code below explains how m-sequence lock and bit comparison is done at the same time to reduce extra errors:

```
sum = sum + rxBit ^ (mseq & 0x1) // Correlation
If( rxBit != mseq & 0x1 )  //Non inverted m-sequence case
  Errors++
If( !rxBit != mseq & 0x1 ) // Inverted m-sequence case
  Errors_inv++
If(number of rxBits == m-sequence length) // Correlation done
  If( sum <= Threshold)               // Non-inverted case
       totalErrors = totalErrors + Errors //Correlation valid
       Errors = 0
  Else If( sum >= N - Threshold)      // Inverted case
       totalErrors = totalErrors + Errors_inv // Valid
       Errors = 0
  Else
       Errors = 0 // Correlation not valid, discard errors
```

4.4.2 Noise generation

For the cases when the performance of the receiver needs to be tested separately from the performance of the communication link, or when a new communication algorithm needs to be tested, there is an ability to add normally distributed noise to the received signal. There are two methods that are used to generate normally distributed noise inside of the DSP. For a large constellation such as 16-QAM, DSP is under heavy load due to larger constellation that needs to be mapped, and four separate m-sequences that need to be correlated. For this case normally distributed numbers were created using MATLAB with zero mean and unity variance. A total of 16384 numbers were created for I and Q branches separately, and converted from decimal representation to the DSP's Q2.13 notation. These 16384 numbers were stored in 2 different signed integer arrays. When noise is added to the received signal, a value is taken from I and Q arrays, and multiplied by the desired standard deviation to achieve required E_b/N_0. This solution does not consume a lot of processor power, but consumes a lot of memory. In order to achieve more randomness larger set of numbers can be created.

When there exist extra processing power, and there is a need for a more random normally distributed numbers, a Box-Muller method was chosen to convert uniformly distributed numbers to a normally distributed numbers. A Box-Muller method converts uniformly distributed random variable to normally distributed random variable the following way:

$$N_1 = \sqrt{-2 * \ln(U_1)} * \cos(2 * \pi * U_2)$$

$$N_2 = \sqrt{-2 * \ln(U_1)} * \sin(2 * \pi * U_2)$$

where N_1, N_2 are independent normally distributed random variables, and U_1, U_2 are independent uniformly distributed random variables [31]. In order to generate independent uniform variables 6 32-bit m-sequences were used. Using the DSP's Q2.13 notation, creating uniform variable requires 13 bits. 4 most significant bits are taken from two separate m-sequences and 5 most significant bits from the third m-sequence. Another three m-sequences are used for the second uniform variable. As the length of the LFSR is 32-bits, the total period is very long, creating a large set of randomly distributed numbers. Square root and natural log operations are very processor intensive, and natural log operation does not work correctly for very small numbers. Because of that a look-up table was created containing values of $\sqrt{-2 * \ln(U_1)}$ for all possible values of U_1. As the value of $U_1 = 0$ is not a valid value, table entries are for indices $U_1 + 1$. Similarly to the first described method, in order to get desired E_b/N_0 calculated normal variables are multiplied by desired standard deviation. Tails of the normal distribution depend on how small value U_1 can be, and therefore this method does not work too well for high values of E_b/N_0.

4.4.3 Data Transmission

In order to transmit and receive data, couple of the steps needs to be performed. First step is converting data to bits, as the transmitter expects an array of bits for the transmission. Second, padding needs to be added to the main data in order to be able to find the beginning and the end of the data stream.

Converting data to bits is performed by the main computer communicating with the DSP board. This is done to reduce amount of work that the DSP needs to perform to send the data. Also this allows user to write their own application that would provide transmitter with the bits necessary as they wish. Currently developed protocol allows user to submit up to 255 symbols for modulation, which allows data size to be up to 1 kbit or 128 bytes. For other than binary modulations, data is mapped into parallel arrays as shown in Table 4.5

Original data: 101101110101001					
Data Array 1	1	1	1	1	0
Data Array 2	0	0	1	0	0
Data Array 3	1	1	0	1	1

Table 4.5: Example of mapping data for 8-level modulation

If original data was represented in bytes, there is no zero padding needed for the 2 level, 4 level, and 16 level modulation. However for the 8 level modulation number of bits might or might not be evenly divisible by 3, and some zero padding might be necessary.

In order to be able to find beginning of data synchronization is necessary. The start of the data needs to be indicated by distinct sequence, such that it would be impossible to mistaken data for it when it is corrupted by noise and such that a data pattern present equal to the synchronization sequence is also unlikely. Similar to the bit error rate testing sequence one of the best choices for the synchronization sequence is the m-sequence, because of its properties. Compared to the bit error testing, however, synchronizing data has different challenges. The main challenge is how to find synchronization sequence in the middle of the data stream. For this project, the synchronization sequence is a 15-bit m-sequence. There is only one generator polynomial, which is $X^4 + X^3 + 1$. The sequence is short enough that synchronization happens quickly, and it is long enough that cross-correlation of it with data will give less false positives. It is also long enough that probability of finding same bit pattern in data is low. A longer synchronization sequence should theoretically improve odds even further, but synchronization will take longer as well. Because 15-bit m-sequence is not followed by another m-sequence, like in bit error rate testing case, a different initial condition cannot be used for the synchronization. The best approach to synchronization is to keep correlating incoming data against the local m-sequence at the receiver until correlation peak appears. The method is based on the assumption that original data was represented using bytes, and therefore resulting number of symbols will be an even number. Even number of symbols will most likely not be multiple of odd m-sequence length of 15, and because of that eventually sequence will align itself correctly. There are still some cases when the received data will never synchronize if it is just left alone. For those cases an input FIFO is used. FIFO length needs to be no longer than $\frac{N}{2}$. If after predetermined amount of cross-correlations there is no correlation peak, input is delayed using FIFO until synchronization is achieved. Figure 4.18 shows cross-correlation of streaming data containing an

m-sequence against a local replica m-sequence with the FIFO present. It can be seen that for the first 90 integrations, the peak could not detected. As soon as the correct FIFO delay is found exact position of the m-sequence is known, and data is tracked. This is evident with 2 correlation peaks at around 90th and 110th integration.

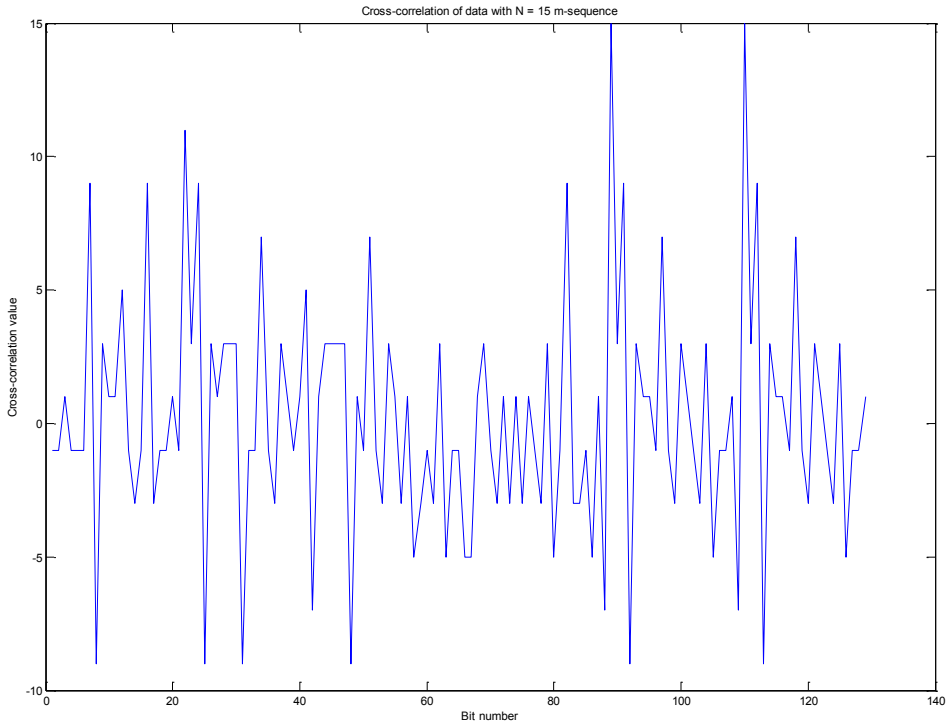

Figure 4.18: Cross-correlation of data with m-sequence

The pseudo code describing the synchronization process is shown below:

```
// Input delay FIFO
FIFO[3] = FIFO[2];
FIFO[2] = FIFO[1];
FIFO[1] = FIFO[0];
FIFO[0] = rxBit;

// Correlate incoming data against local m-sequence
sum = sum + FIFO[fifo_idx] ^ (mseq & 0x1)

// If correlation succeeds next incoming bit is the data
If(sum == 0 or 15 ± Threshold)
  // Start receiving data

// Keep count of number of integrations
Integrations = integrations + 1
If(number of integrations == LIMIT)
```

```
Integrations = 0;
fifo_idx = fifo_idx + 1;        // Change delay
```

Similarly with the bit error rate testing m-sequence, when errors are present cross-correlation will not always be 0 or 15, but because m-sequence is shorter, threshold should be tighter as well. As soon as the data is received, transmission of data is turned off, and data is sent back to the PC. Dynamically allocating an array inside of the receiving portion of the software is resource consuming, and stalls the process, therefore receiving array is pre-allocated to the maximum bit size allowed so far, which is 1024 bits. If the sequence is shorter than that, a byte value 255 is transmitted at the end of the bit sequence. The bit sequence is transmitted back to the PC, where it is converted to the user defined data.

The process of mapping bits for multi-level modulations was described and an example shown in Table 4.5. With the addition of the m-sequences the mapping is done as follows:

Original data: 101101110101001						
Data Array 1	**15-bit m-sequence**	1	1	1	1	0
Data Array 2	**15-bit m-sequence**	0	0	1	0	0
Data Array 3	**15-bit m-sequence**	1	1	0	1	1

Table 4.6: 8-level modulation mapping with synchronization example

It can be seen that for larger constellations it is better to send more data bits, otherwise majority of data will consist of just m-sequence bits. Also, each 15-bit m-sequence at the beginning of the array in Table 4.6 is the same. This is done such that only one local m-sequence generator is needed for synchronization. Finally, all L m-sequences, where L is the number of bits per symbol, need to be either non-inverted, or inverted for the data to be correctly recovered. Therefore the only time data is valid is when all L cross-correlations have the same value.

4.5 Network Communication

As the DSP is performing modulation and demodulation of data, as well as error calculation, it is essential to show this data to the user. Of course it is possible to just run the DSP platform using Code Composer Studio and monitor data using debugger. This method would require user to be familiar with the Code Composer Studio software, and have it installed on their computer. This

is inconvenient, and therefore it is better to communicate the information with the PC using more standard interface.

DSP is capable of communicating with the PC using a number of the interfaces, including UART, parallel port, and Ethernet. UART and parallel port are older interfaces that no longer come standard on most computers, and would require a USB adapter. Therefore the best interface to use is the Ethernet. As previously described in chapter 4.1.2 TMS320C6455 DSP that was used in the project has a network package already supplied, that allows easy use of the interface. Package provides ample amount of the example programs to jump start the programming.

Network communication between the PC and the DSP was done using UDP protocol. UDP stands for User Datagram Protocol and is a simpler version of the more commonly seen TCP protocol. UDP protocol is used when data transfer reliability is not a major concern, as UDP protocol is "connectionless". This means that when data is send to a particular destination, UDP protocol does not know if the destination is valid. The data is just send over. Therefore it is up to the user to implement method to make sure connection is reliable. Because of the simplicity of the UDP protocol it is much faster than TCP protocol, and more commonly used for streaming data, and embedded systems, where speed is more crucial. A good example of everyday UDP protocol use is the facsimile.

4.5.1 Network Protocol

In order to communicate between the PC and DSP a set of commands needed to be developed. The developed set of commands is shown in the Table 4.7. Overall commands starting with byte 0 indicate a control command to the DSP, such as set modulation, or turn on/off BER testing. Commands starting with the byte 1 indicate data transfer commands. Command {2,0} is used as an echo command to make sure DSP is connected, as previously mentioned UDP protocol is "connectionless" and does not indicate if the other device is receiving or sending data. All of the commands send to the DSP are responded back to make sure command was acknowledged and to show updated status to the user. For bit error rate testing, errors are calculated until limit is reached. Currently the limit is set at 6300 bits. Command {0, 200, 0} is send over every 250 ms by the PC to the DSP.

PC Command	DSP Response	Description
{2, 0}	{2, 0}	Echo command to see if DSP is connected
{0, 0-15}	{'M', 0-15}	Set the desired modulation. DSP responds with acknowledgement that modulation was set
{0, 255}	{'E', 0}	Turn ON BER testing / Turn OFF data TX/RX
{0, 254}	{'E', 1}	Turn OFF BER testing / Turn ON data TX/RX
{0, 100}	{'F', 0/1}	Turn ON/OFF FPGA communication. DSP responds with the state of the FPGA communication (0 – off, 1 – on)
{0, 200, 0}	{'B', 2 byte value}	Send the number of total errors back to the PC when BER is ON, or send 0xFFFF if BER is not ready
{0, 250, 2 bytes}	{'N', 0,0,0}	Set the noise standard deviation if artificial digital noise is added by DSP
{1, DATA_LENGTH, data}	{1, 'D'}	Data to be modulated and send. DATA_LENGTH is the number of symbols, based on modulation currently selected and number of bit in a data
{0, 200, 1}	{'D', bits, 255}	Data received by the receiver. DSP sends an array of received bits, followed by end character 255

Table 4.7: A summary of the network commands

If the BER calculations are ready, the result is send back; otherwise a dummy load of 0xFFFF is send back. Current protocol supports number of errors to be transmitted back up to about 65500. Every time error is reported back, PC also switches which BER sequence in multi-level modulations is used. This is done to make sure average error is correct, and particular set of symbols does not overpower the other. For data transmission case, PC converts any data user

wants to transmit, usually in bytes, to bits, adds the number of bits for synchronization using framing shown in Table 4.6, and calculates based on modulation number of symbols to be transmitted. Calculated number is the DATA_LENGTH field of the command. Note, synchronization bits are added by the DSP.

4.5.2 Graphical User Interface for the PC

For ease of user interaction with the DSP, a graphical user interface (GUI) was created using Visual Basic .NET language. The GUI is shown in the Figure 4.20 and Figure 4.20. Visual Basic was chosen, because it is very easy to write graphical programs with it. An example program for UDP communication in Visual Basic .NET can be found in [27]. Figure 4.19 shows GUI at the start-up state. All fields are grayed out, because DSP and PC are not connected. As soon as valid IP address is input into *IP Address* field, and *Connect* button is pressed other input fields are enabled. *DSP Response* window shows the acknowledgements by DSP parsed to user friendly text. *Received Data* is used to show any data that was received by the DSP, converted from bits to ASCII code. Figure 4.20 shows GUI with activity in it during text reception. When GUI's *BER Test* flag is checked *Text to send* and *File Path* windows are grayed out, while *EbNo* and *BER* are enabled. *BER* window shows running bit error rate results. The error result is shown is the form:

Errors/Total Bits per Iteration: Decimal Error Rate

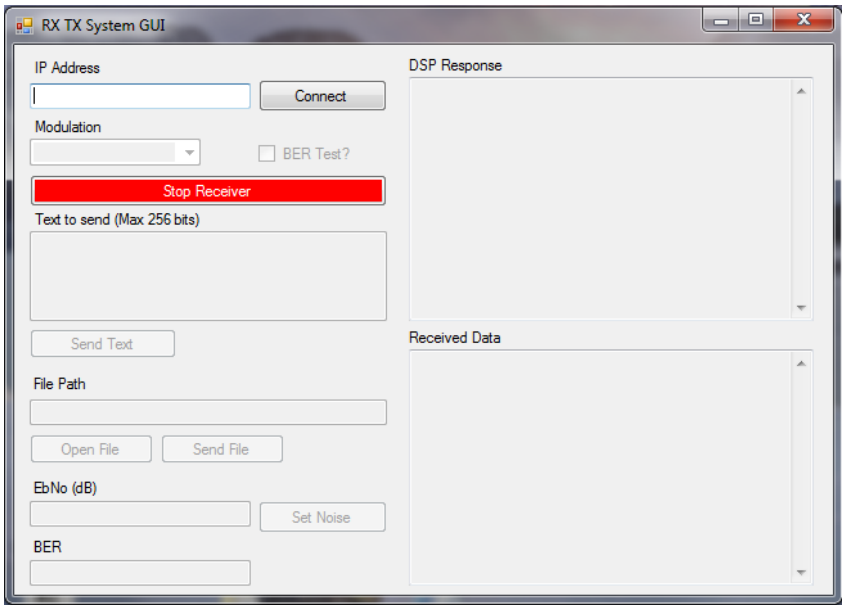

Figure 4.19: GUI to communicate with the DSP

When *Errors* is equal to *NaN* it means that error calculations are not finished due to some reason. Most often it is due to lack of m-sequence synchronization, or lack of PLL synchronization. *Total Bits per Iteration* as previously mentioned is set to 6300 bits, but can be changed any time in both GUI code and DSP code. *Total Bits per Iteration* should be small enough such that during cases of low SNR it does not take a long time for the system to show BER results.

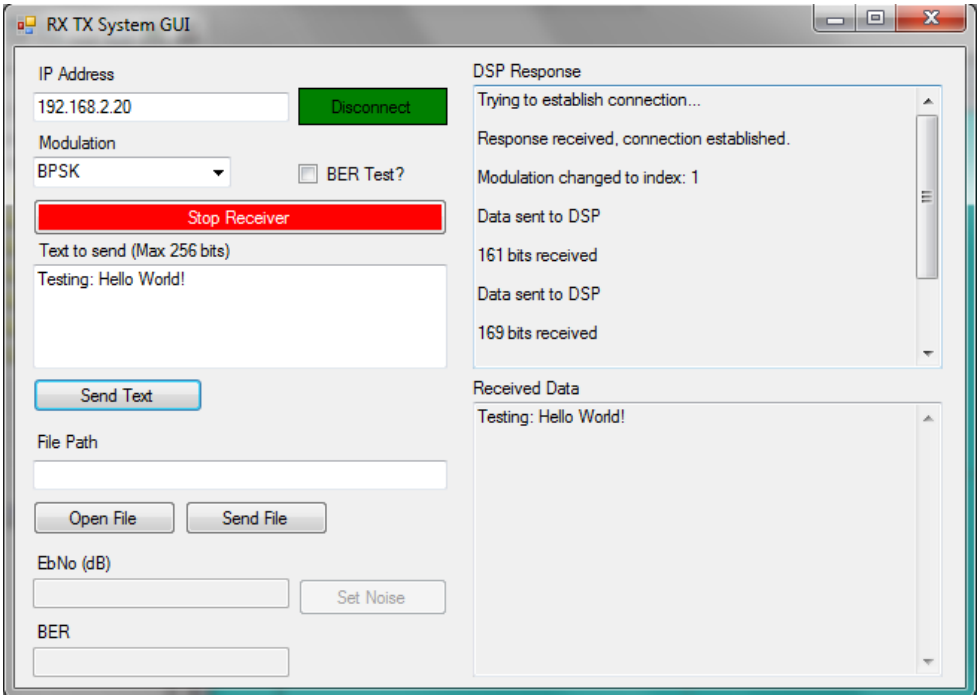

Figure 4.20: GUI with some activity

4.6 Programming DSP for Standalone Operation

Even though Ethernet allows user to exchange information between PC and the DSP easily, unless DSP is working in standalone operation, Code Composer Studio is still needed to load the code on to the DSP and run it. This creates same problems for the user that were already mentioned in chapter 4.5.

TMS320C6455 DSK has a set of switches that allow user to select type of boot to use. The switches and the boot option they enable are shown in Table 4.8. The default boot option that comes with the board is indicated with the star. There is an option to boot DSP from the I^2C ROM that's located on the board. I^2C ROM is only 1 Mbit, while full internal RAM space is 2 Mbytes, which means that some programs might not fit. Also, I^2C boot process is pretty slow. An

example code is available that shows how to program I^2C ROM, however it was not working, and that was one of the reasons Flash was selected as program storage instead.

SW3-5 AE19	SW3-4 * AE18	SW3-3 AE17	SW3-2 AE16	Configuration Description
Off	Off	Off	Off	No Boot
Off	Off	Off	On	Host Boot HPI
Off	Off	On	Off	Reserved
Off	Off	On	On	Reserved
Off	On	Off	Off	EMIFA 8 bit ROM Boot *
Off	On	Off	On	I^2C Boot Master
Off	On	On	Off	I^2C slave
Off	On	On	On	Host Boot PCI
On	x	x	x	Serial Rapid I/O Boot see DSP data sheet for more details

Table 4.8: Configuration switched for different DSP boot options [5]

There is also an external 4 Mbyte Flash memory located on board that is mapped to address space 0xB0000000 to 0xB03FFFFF. The Flash is larger than full internal RAM, which means it can be used for program storage and provide additional data storage if needed. The Flash boot option is the default boot option that is shown in Table 4.8.

In order to program DSP for the Flash boot some preparations need to be performed. When the EMIFA 8-bit ROM Boot option is selected EDMA automatically copies first 1 kbyte from Flash memory to the internal RAM [20]. The process was described previously in chapter 4.1.3. Most of the DSP programs are larger than 1 kbyte, and instead of the main program, first 1 kbyte of Flash memory contains a secondary boot-loader that after initial start-up loads the rest of the program. An example of such boot-loader came with the TMS320C6455 DSK. The boot-loader is an assembly file with copy instructions. Similar boot-loader code can also be found in the Texas Instruments reference [20]. In addition to the assembly secondary boot-loader file, a boot section of the memory needs to be created, as described in the chapter 4.1.3, and user linker file created where boot-loader is put into boot section of the memory. After that, all of the program code is converted to an ASCII hex file with the help of the *hex6x* utility. The utility needs a set of commands that are put into a .cmd file. The command file is provided in the Appendix D. The *hex6x* utility maps all of the initialized code to the corresponding Flash location.

After an ASCII hex file is created, it needs to be programmed into the Flash somehow. To do that free software *FlashBurn* from the *Software Design Solutions* is used. *FlashBurn* software comes with pre-written templates, but template for the TMS320C6455 DSK needs to be downloaded separately. Template tells *FlashBurn* how to access Flash memory on the board. The process of programming Flash memory using *FlashBurn* is shown in Figure 4.21.

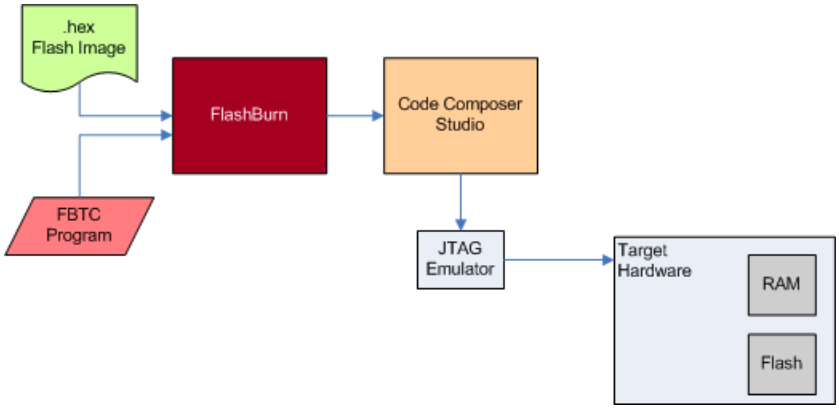

Figure 4.21: Diagram of how FlashBurn works [33] [Fair Use]

FlashBurn requires the user to have Code Composer Studio installed. Right now *FlashBurn* works with Code Composer Studio up to version 3.3. When user runs *FlashBurn* application it puts temporary code in the DSP that erases and programs the Flash memory.

After programming is finished the user will see 2 out of 4 LEDS on the DSK blinking. Blinking LEDs indicate that network driver is functioning, and that overall DSP is working as well. If the LEDs are not blinking, there is a white button next to the Ethernet connector that needs to be pressed. The white button is the reset. Sometimes it might take multiple resets until DSP enters working state.

5. System Performance

The software defined transceiver performance shown in the next couple sections was measured using two methods. For the BER curves, a clean signal with uniformly spaced samples was placed inside of the block memory in the FPGA as described in chapter 3.2.1, and played back continuously. Signal contained a 30° phase angle shift, and a 15.873 kHz or 31,746 kHz frequency shift from the IF frequency of 10 MHz. After passing through the processing blocks inside of the FPGA, a digitally generated white noise was added to the signal inside of the DSP, before the PLL to simulate what effect a noisy signal would have on PLL performance. A

second test performed was a wireless link test. For this test two monopole antennas were made tuned to 1 GHz center frequency. Antennas were separated by a distance of 3-4 meters. Two conditions were tested: one where antennas are in line-of-sight with each other, and two, where antennas are not in line-of-sight. These tests were performed to see the effects of possible reflections on the transmitted signal, and to calculate the base performance of the transceiver with the implemented timing error correction.

5.1 Start-Up Sequence

Before error testing or data transmission can be performed using transceiver, it needs to be configured through the ComBlock Control Center which can be downloaded together with the driver for the FPGA from ComBlock website. After applying power to the DSP board, the user should see 2 blinking LEDs on the DSP board. As previously mentioned, if the 2 LEDs are not blinking then user needs to press white reset button, until the LEDs start blinking. Even though modulation can be changed from the network GUI, AGC needs to be adjusted from the ComBlock Control Center. Configuration window is shown in Figure 5.1. Reg 0 is responsible for setting average energy for the modulation. User can reference Table 3.1 for the value and corresponding modulation.

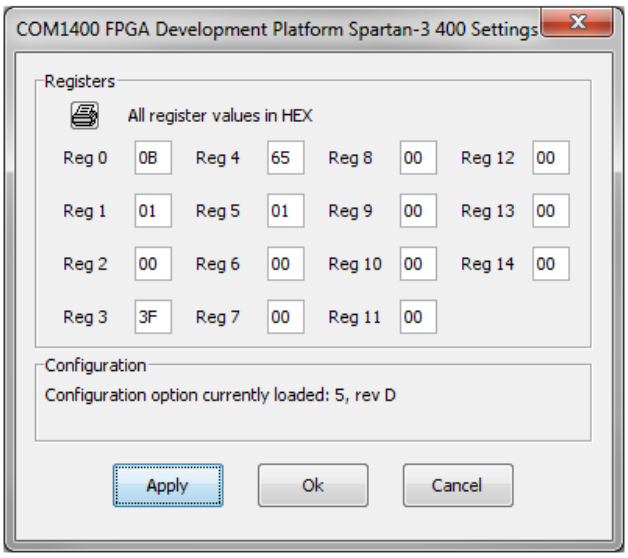

Figure 5.1: ComBlock configuration window

Value needs to be entered in hexadecimal formal. Reg 1 selects the source of the signal (0 – FPGA memory, 1 – directly from ADC). Reg 3 and Reg 4 set the 16-bit value for the IF frequency. To find the value required for the desired frequency, the following formula is used:

$$D = \frac{f_{IF} * 2^{16}}{40 * 10^{6}}$$

where f_{if} is the desired IF frequency, and D is the decimal value that corresponds to it. As with the Reg 0 , decimal value needs to be converted to the hexadecimal value. IF NCO has an approximate step size of 610 Hz. Finally, Reg 5 sets the maximum limit timing error can reach before delay is adjusted. The maximum error limit value and corresponding decimal value can be referenced from Table 3.2. In addition to the configuration registers, there are also 8 status registers. User can use status register 0 to see if the FPGA and DSP are communicating successfully. Figure 5.2 shows two status registers values. Window on the left is a snapshot of

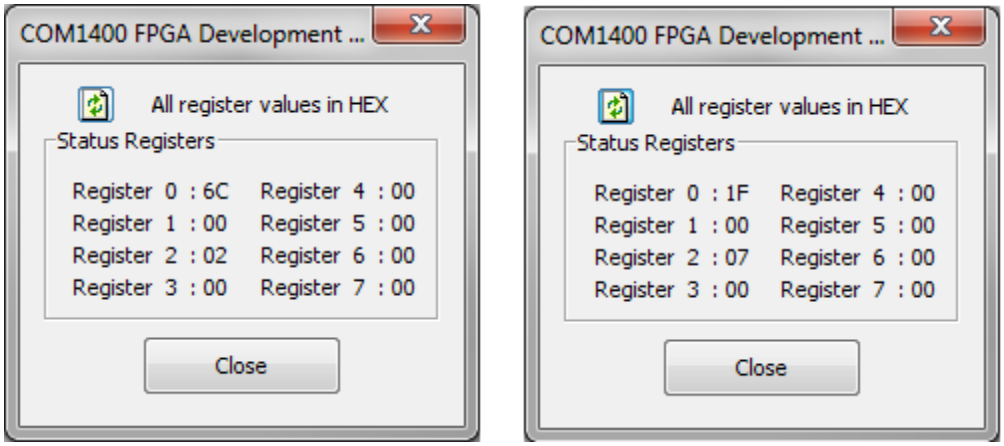

Figure 5.2: Monitoring registers values for active and inactive DSP link

the status registers values when DSP link is working properly. If the DSP link is off, or has not been started, it will look like window on the right. If the user turned off FPGA receiving from GUI, they can confirm if they see value *1F* in the status register 0.

For user reference one of the FPGA's implemented blocks is the ComScope. ComScope is the ComBlock's version of the signal analyzer. An example of the ComScope screen can be seen in Figure 5.3

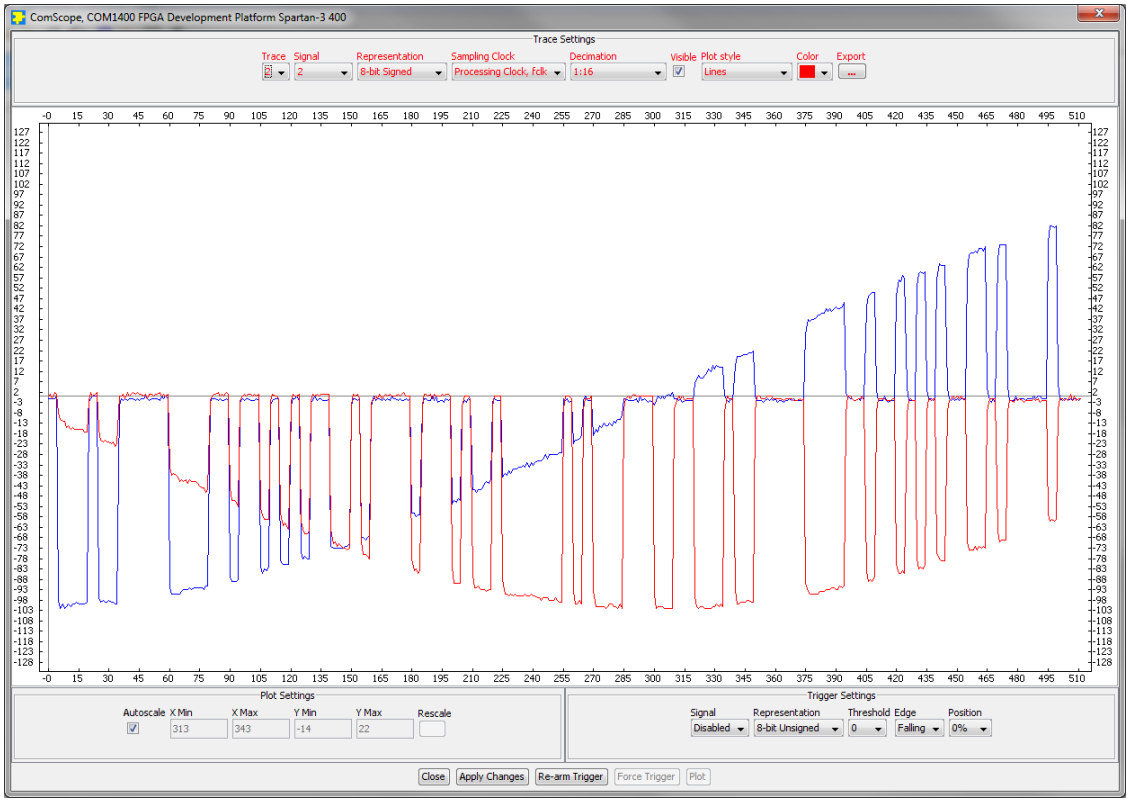

Figure 5.3: ComScope Output Screen

Currently there are couple of the outputs wired into the ComScope that the user can see using a ComScope icon in the ComBlock Control Center. Trace 1 and Trace 2 correspond to I and Q channels respectively. Signal 1 for both traces is the output from the AGC before decimation. Signal 2 is the output after CIC filter. This output is usually better to observe, as it does not contain any noise from mixing. Signal 3 is the output of the matched filter. User can reprogram these outputs at any time. For more information on the ComScope user should refer to ComScope manual [32].

These instructions assume that DSP and FPGA were both pre-programmed with the correct software. If for any reason this is not true, user needs to use *FlashBurn* utility to program DSP, and use ComBlock Control Center to program FPGA software into flash memory. ComBlock Control Center manual and ComBlock website describe how this can be done.

As all four ComBlock blocks are connected together, they can be configured from the same interface. The only other two configurable devices are COM-3002B, the quadrature down-converter, and COM-4002D, quadrature modulator. ComBlock Control Center can be used to change their center frequency, turn on frequency hopping, and monitor output power from

modulator. User should also remember that the maximum input power that will not cause damage to the COM-3002B is -5 dBm.

5.2 Modulator Output

The output of the modulator was taken at two different points. The baseband output from the DAC was taken using digital oscilloscope. Figure 5.4, Figure 5.6, Figure 5.8, Figure 5.10, Figure 5.12, and Figure 5.14 show the output of the DAC. The top plot is the in-phase channel, while the bottom plot is the quadrature channel. Figure 5.8 also shows additional bars on the plot that confirm that MSK and O-QPSK modulations have in-phase and quadrature channels half-symbol apart. Second output was measured using Spectrum Analyzer after the signal was modulated to the carrier frequency. Figure 5.5, Figure 5.7, Figure 5.9, Figure 5.11, Figure 5.13, and Figure 5.15 show output spectrum of the modulated signal using 5 MHz and 10 MHz spans, with 500 kHz, and 1 MHz per division respectively. As expected linear modulations, such as M-ASK, M-PSK, M-QAM, and M-APSK have all first null bandwidth of 2 MHz. M-ASK modulations also have a DC spectral line, that can be clearly seen in Figure 5.5 and Figure 5.11. M-FSK modulation was implemented with carrier spacing equal to the symbol rate. This means that expected first null bandwidth for BFSK is 3 MHz, and 5 MHz for the 4-FSK. This can be confirmed with Figure 5.7 and Figure 5.10. Although individual peaks are not as visible in Figure 5.10 10 MHz span, they are more discernable in 5 MHz span. Using marker values, it can be seen that the carrier spacing is correct. MSK modulation is a O-QPSK modulation with half-sine pulse shape, and expected bandwidth is $3R_s$, which is 3 MHz. This can be confirmed in Figure 5.9. Even though 63-bit sequence does not create spectral lines, for larger order modulations it is very important that m-sequences are positioned in such a way that resulting symbols are very random. Otherwise output spectrum will contain a lot of artifacts.

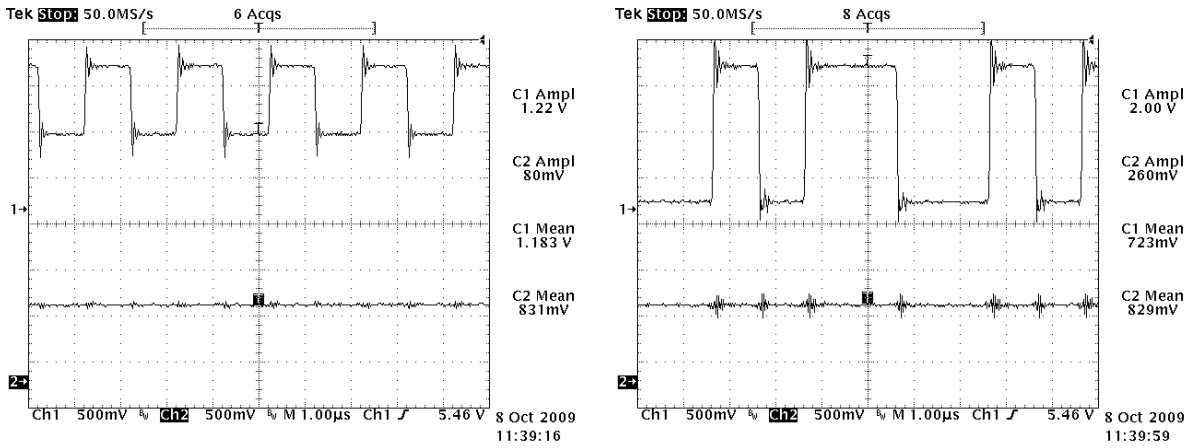

Figure 5.4: BASK and BPSK DAC outputs

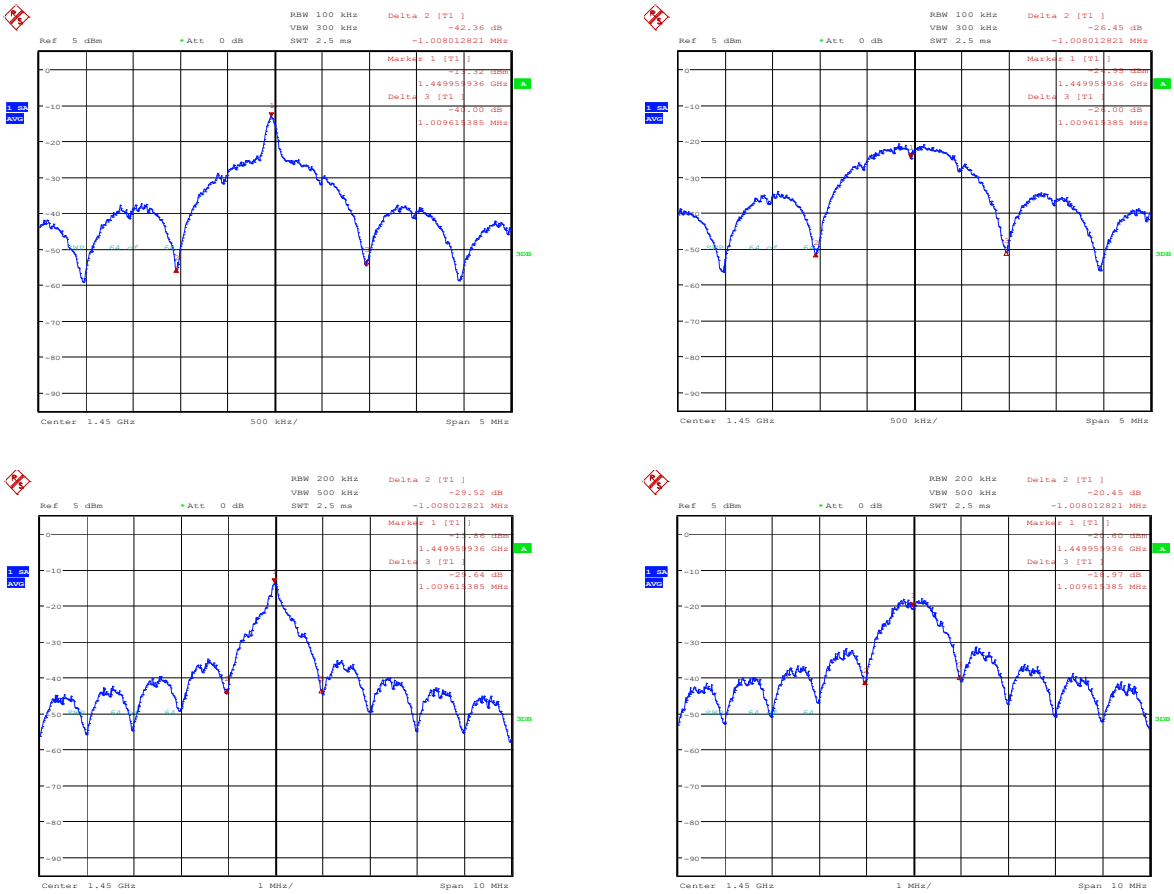

Figure 5.5: 5 MHz and 10 MHz span of BASK and BPSK modulator output

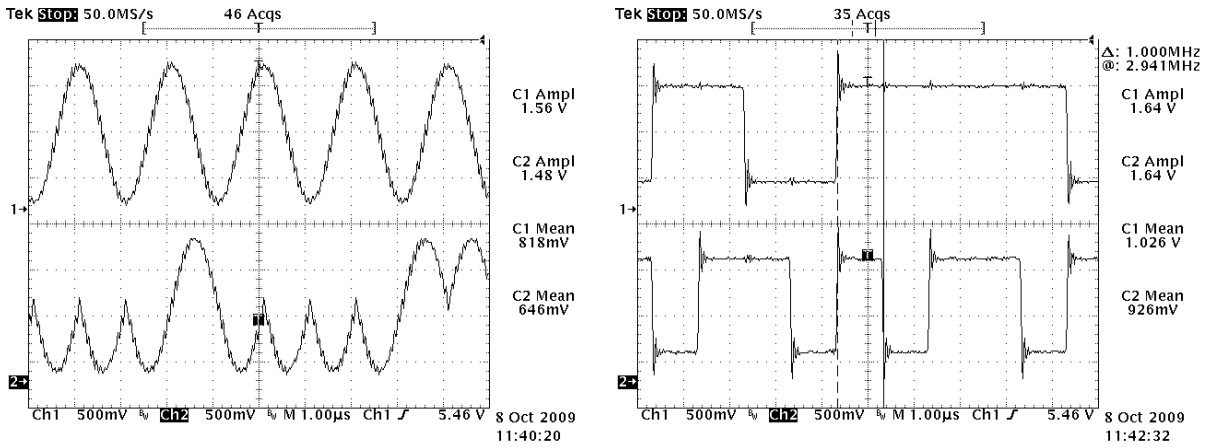

Figure 5.6: BFSK and QPSK DAC outputs

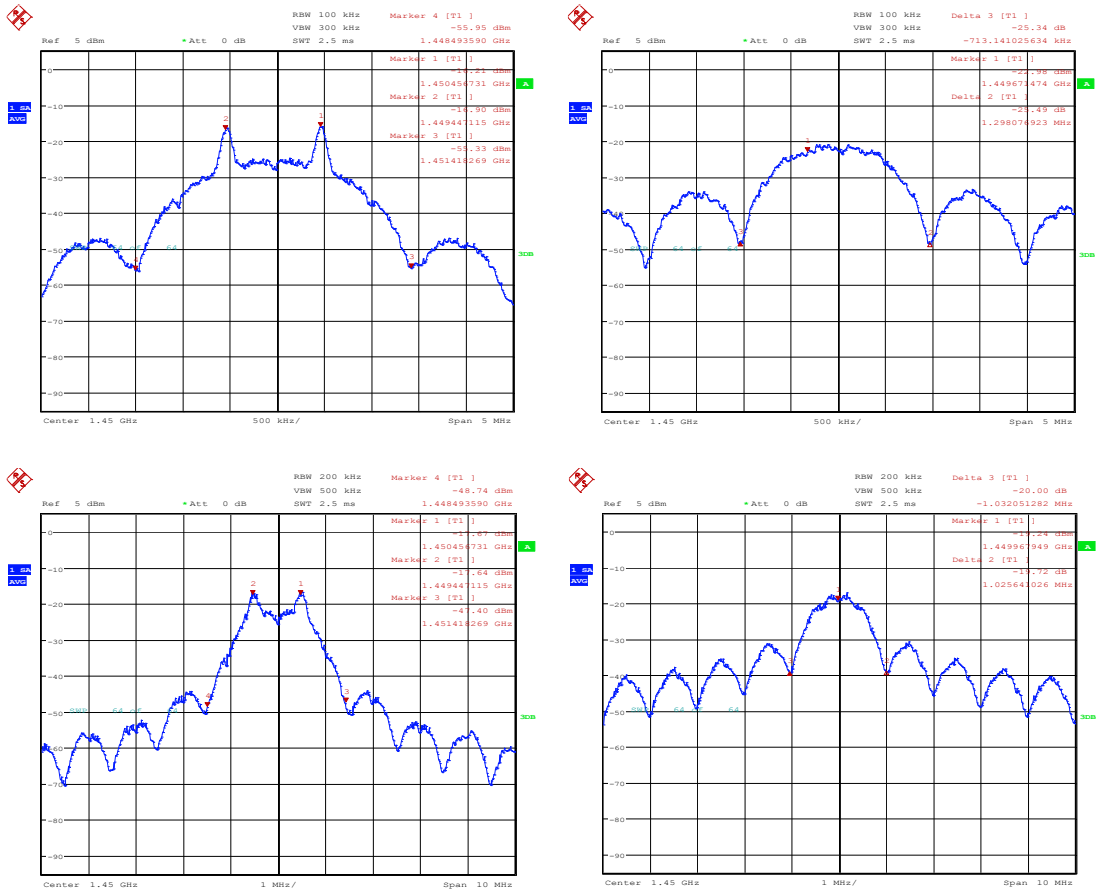

Figure 5.7: 5 MHz and 10 MHz span of BFSK and QPSK modulator output

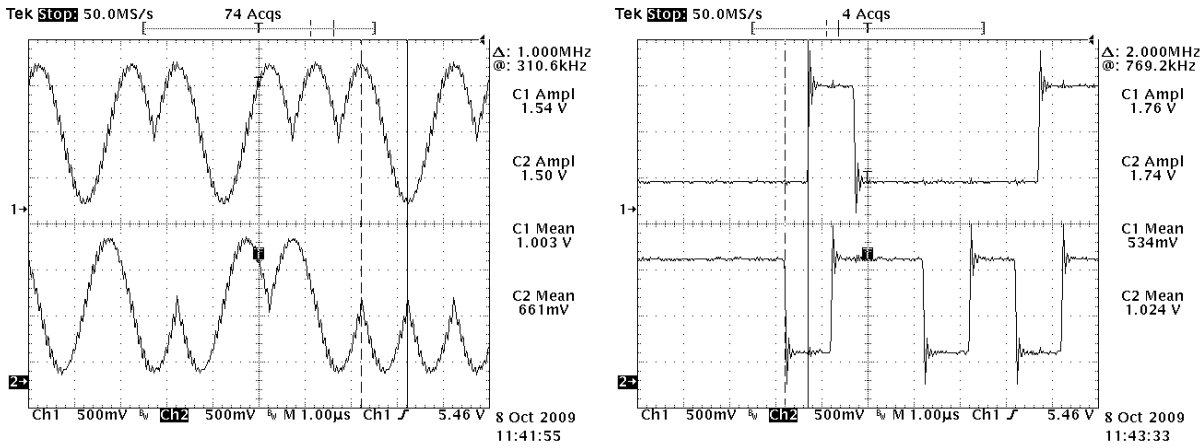

Figure 5.8: MSK and O-QPSK DAC outputs

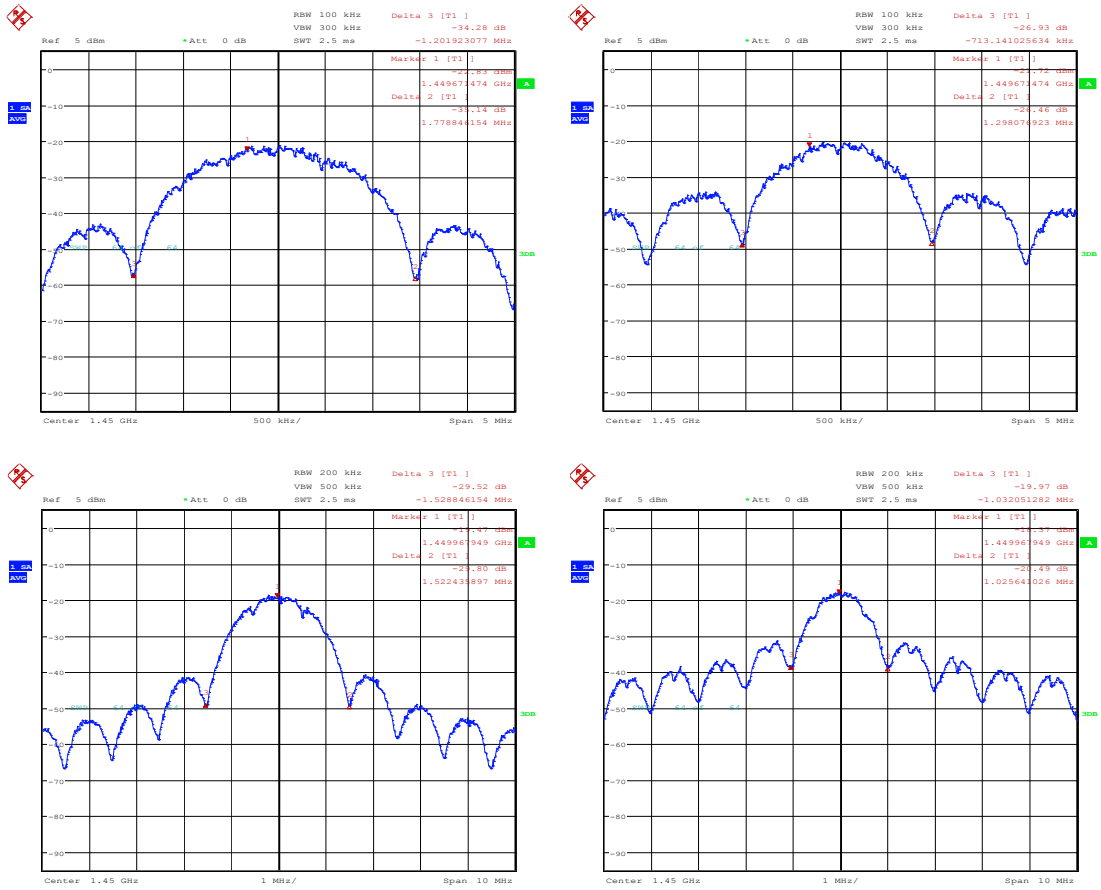

Figure 5.9: 5 MHz and 10 MHz span of MSK and O-QPSK modulator output

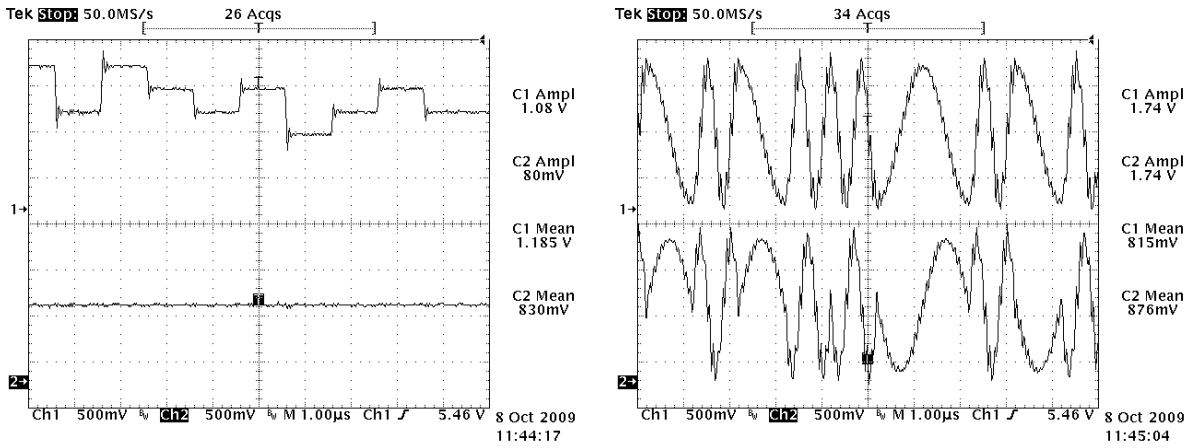

Figure 5.10: 4-ASK and 4-FSK DAC outputs

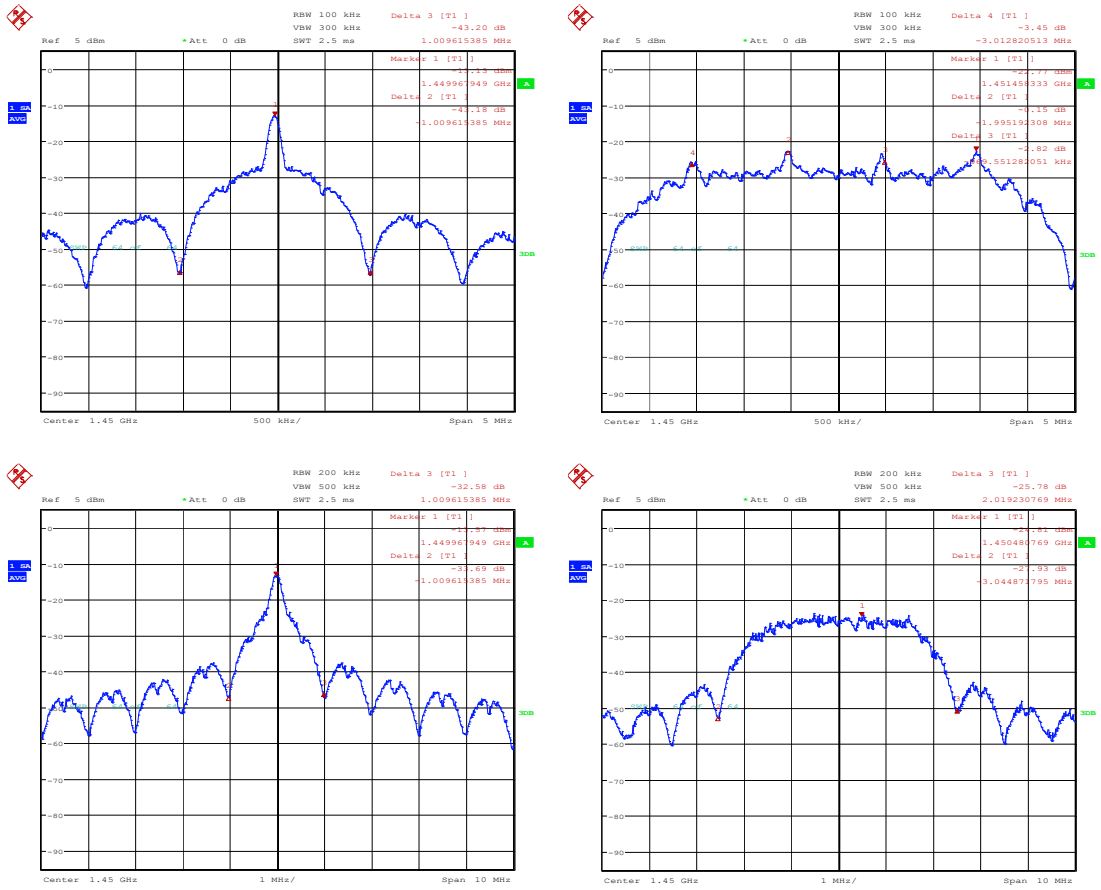

Figure 5.11: 5 MHz and 10 MHz span of 4-ASK and 4-FSK modulator output

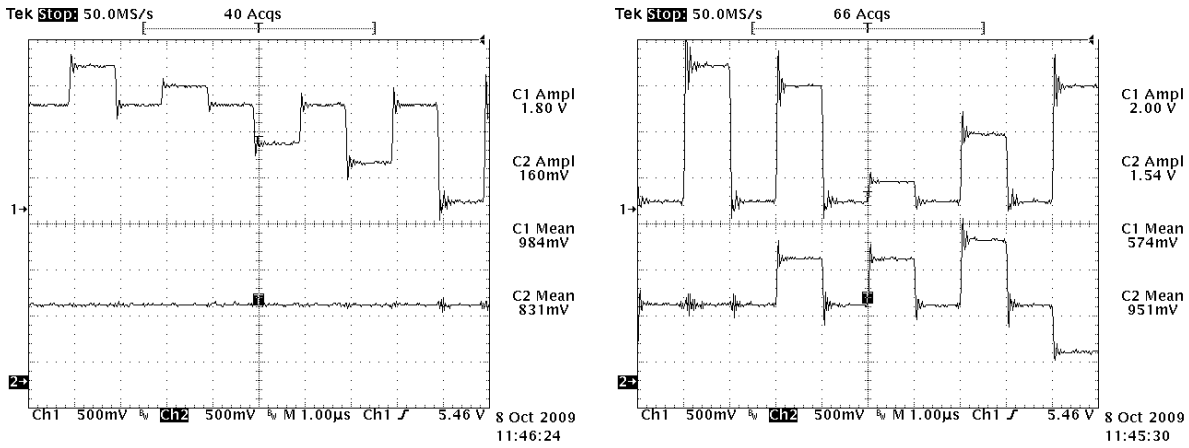

Figure 5.12: 8-PAM and 8-PSK DAC outputs

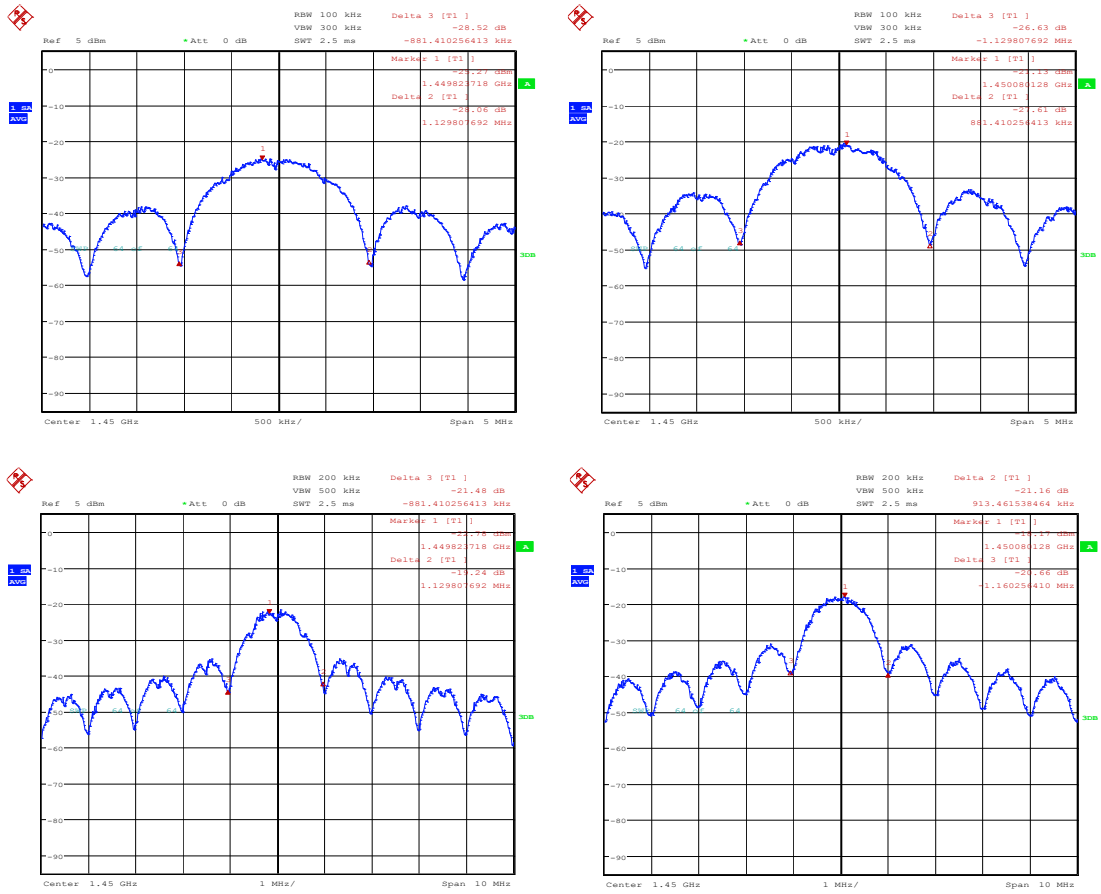

Figure 5.13: 5 MHz and 10 MHz span of 8-PAM and 8-PSK modulator output

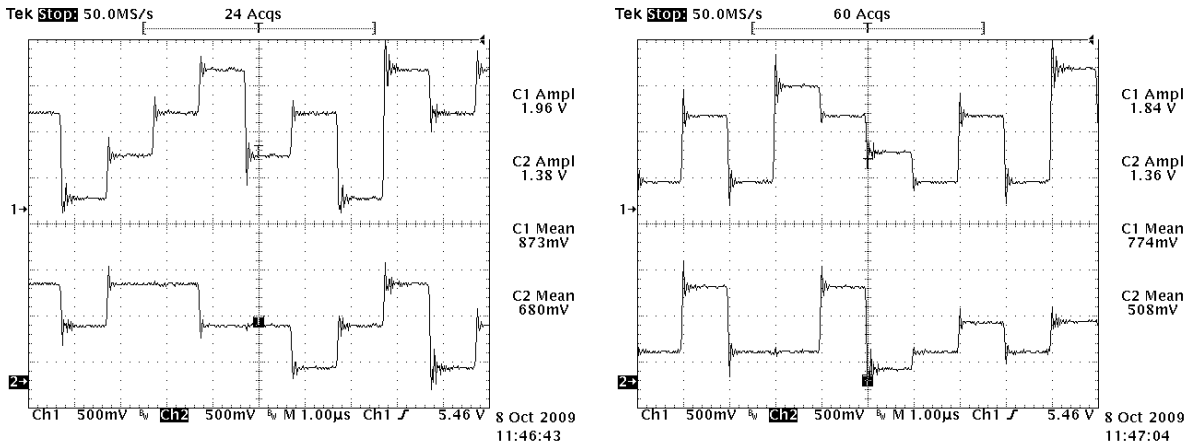

Figure 5.14: 16-QAM and 16-APSK DAC outputs

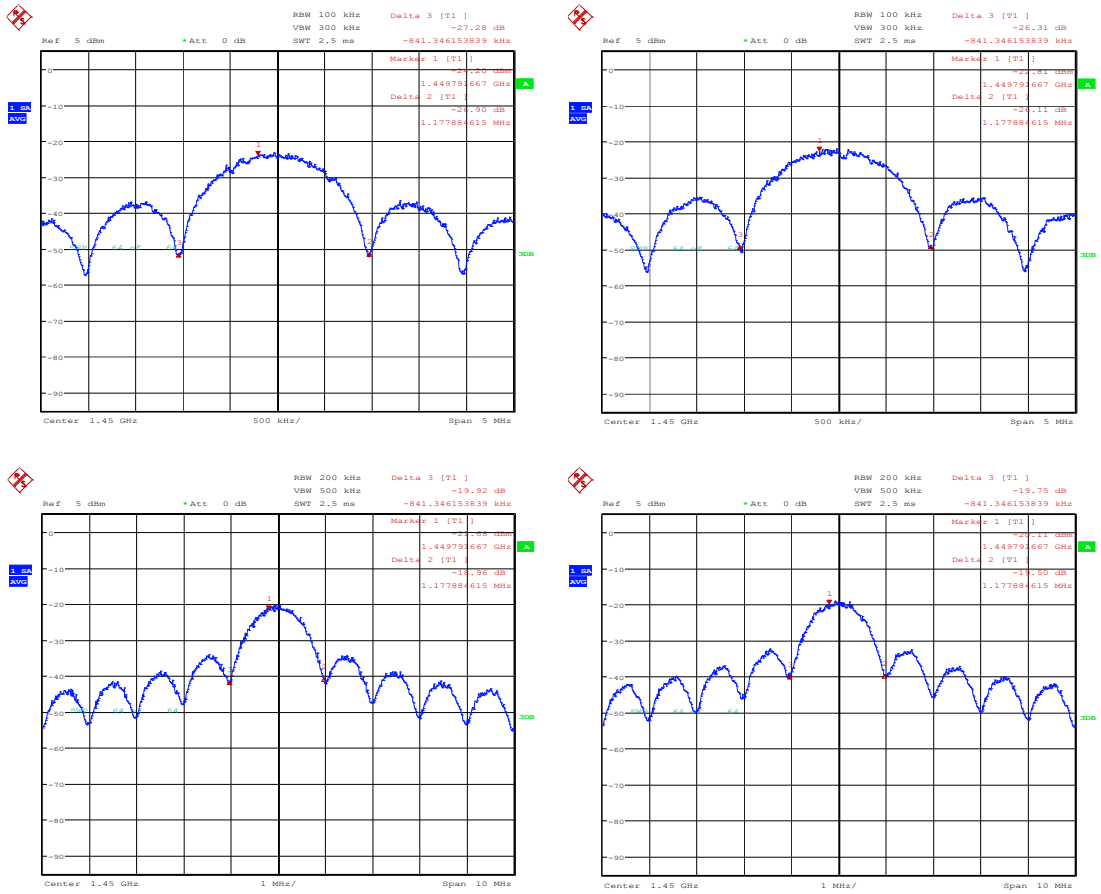

Figure 5.15: 5 MHz and 10 MHz span of 16-QAM and 16-APSK modulator output

5.3 Noise Performance

As previously mentioned, noise performance was tested using the signal that was played back from the FPGA memory. Noise was added at the input of the PLL using a pre-recorded table or the generated noise. In addition to that, network software made sure that for multi-level modulations all m-sequences were tested with equal probability for more reliable bit error rate. Figure 5.16 shows the bit error rate curves for the E_b/N_0 values between 3 and 10 dB with increment of 1 dB, for the theoretical and measured values of BASK, BPSK, and QPSK modulations. As it can be seen, the performance is very good, and matches very closely theory. At the high values of E_b/N_0 it was very hard to create noise that would cause the error, and therefore BPSK modulation had 0 errors at 10 dB E_b/N_0. Figure 5.17 shows bit error rate curves for the same E_b/N_0 values as Figure 5.19, but for the 4-ASK and 8-PAM modulations. It can be seen that performance matches theory very good as well. Error is smaller for the 4-ASK and 8-PAM modulation at lower value of the E_b/N_0. There is no obvious reason why this is happening, but it can be because of the quality of the generated noise.

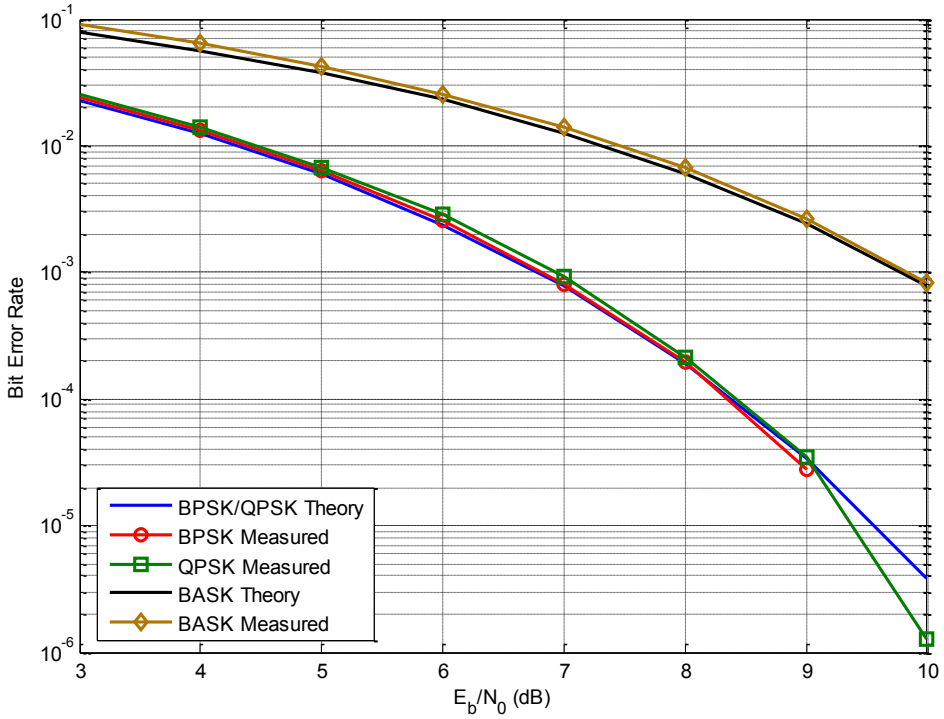

Figure 5.16: BASK, BPSK, and QPSK theoretical and measured performance

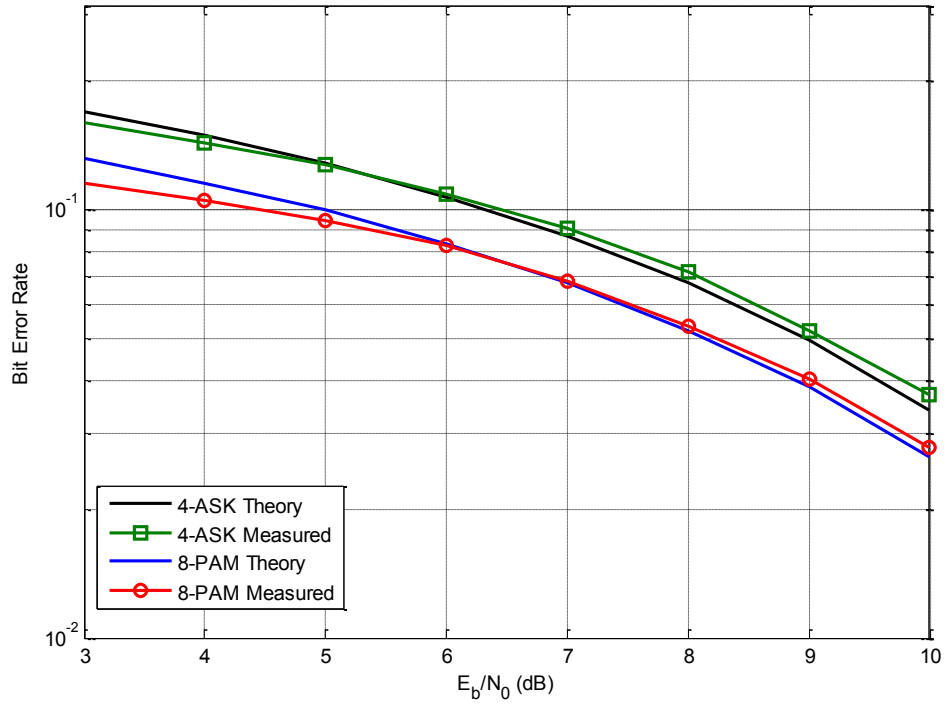

Figure 5.17: 4-ASK and 8-PAM theoretical and measured performance

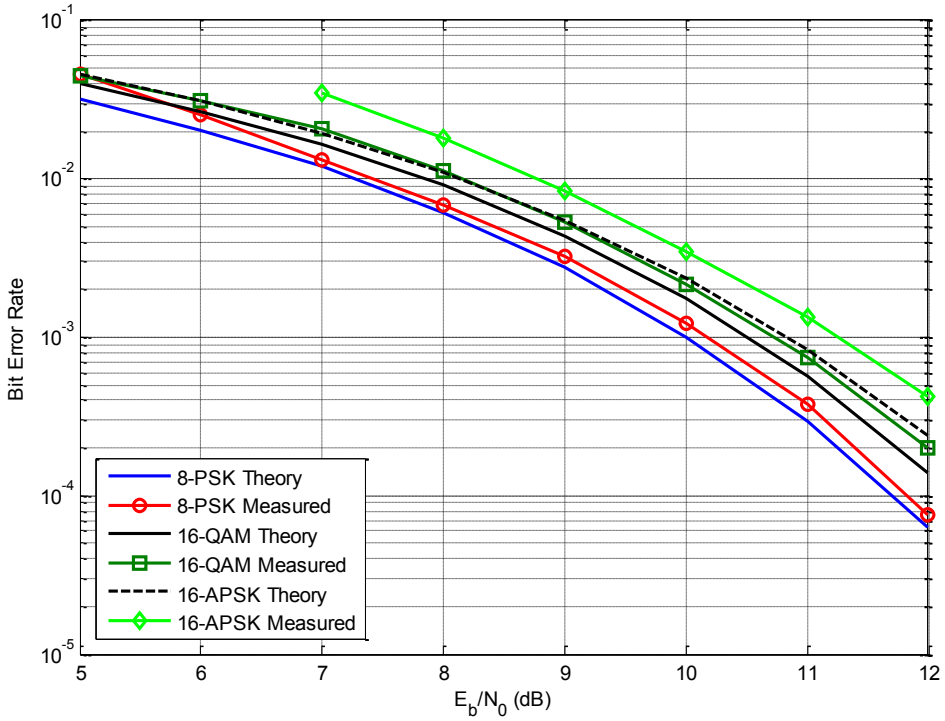

Figure 5.18: 8-PSK, 16-QAM, and 16-APSK theoretical and measured performance

For 8-PAM modulation it can also be due to the fact that constellation rotates, and error calculation can be in the middle of it. Figure 5.18 shows bit error rate curves for the E_b/N_0 values between 5 and 12 dB in 1 dB increments. This is because higher order modulations do not perform well at lower E_b/N_0 values and as decision directed PLL is used, it starts to lose track. For 8-PSK modulation M-power method is used, which has higher self-noise, and does not work as well at lower E_b/N_0 as well. 8-PSK modulation performs very well compared to theory, and has degradation no more than 0.4 dB. 16-QAM modulation also performs pretty well, and has no more than 0.5 dB difference from the theory. 16-APSK modulation was the worst performing modulation. It lost synchronization at E_b/N_0 below 7 dB, and overall performs around 1 dB worse than the theory. This can be attributed to the fact that 16-APSK modulation has a very small phase ambiguity, and constellation is more likely to turn as the signal is received.

Overall all modulations performed very well against the theory. There is a big problem of phase ambiguity in symmetrical constellations like PSK, and QAM. Constellation is rotating while the signal is being received, which causes 2 problems: loss of m-sequence tracking, and additional errors during tracking. Both problems can be solved by introducing error correction, or protocol that would detect phase ambiguity, or simply reducing the threshold at which m-sequence lock is acquired. It is also possible to modify the PLL for better performance.

5.4 Received Constellations

Figure 5.19 through Figure 5.22 show the received constellations for all implemented modulations in the receiver. The images were taken when the receiver is completely synchronized and is connected to the transmitter via the cable. When the receiver loses timing lock, symbols in the images start to move towards the origin which causes an extra error. Overall it can be seen that during correct synchronization constellations are correct and received symbols are very tight, indicating that there is no large amplitude variation between samples.

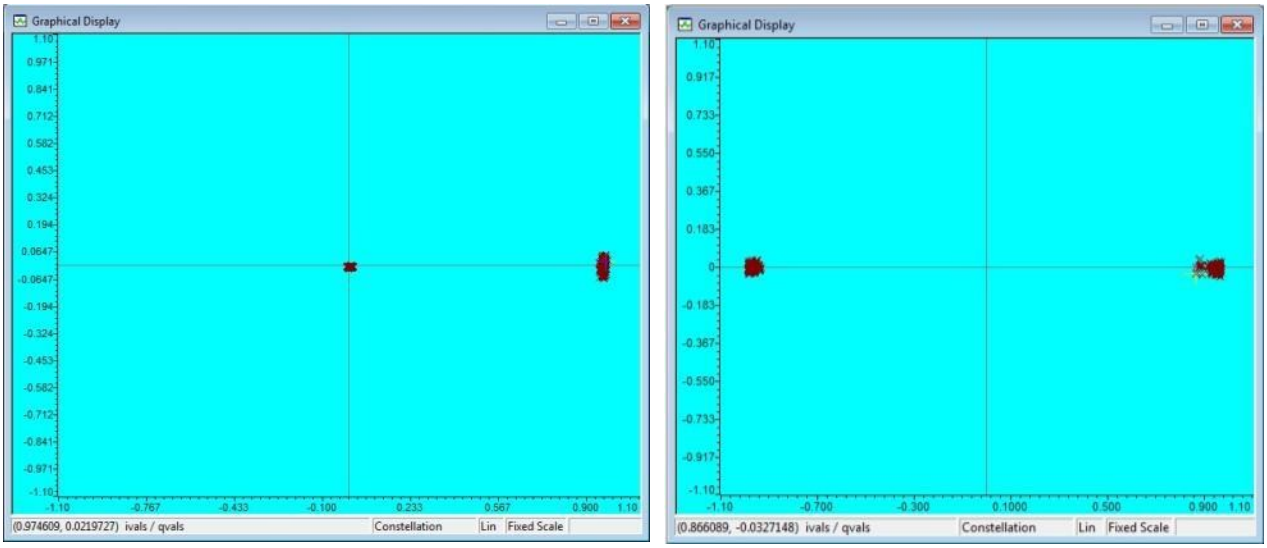

Figure 5.19: BASK and BPSK constellation at the receiver

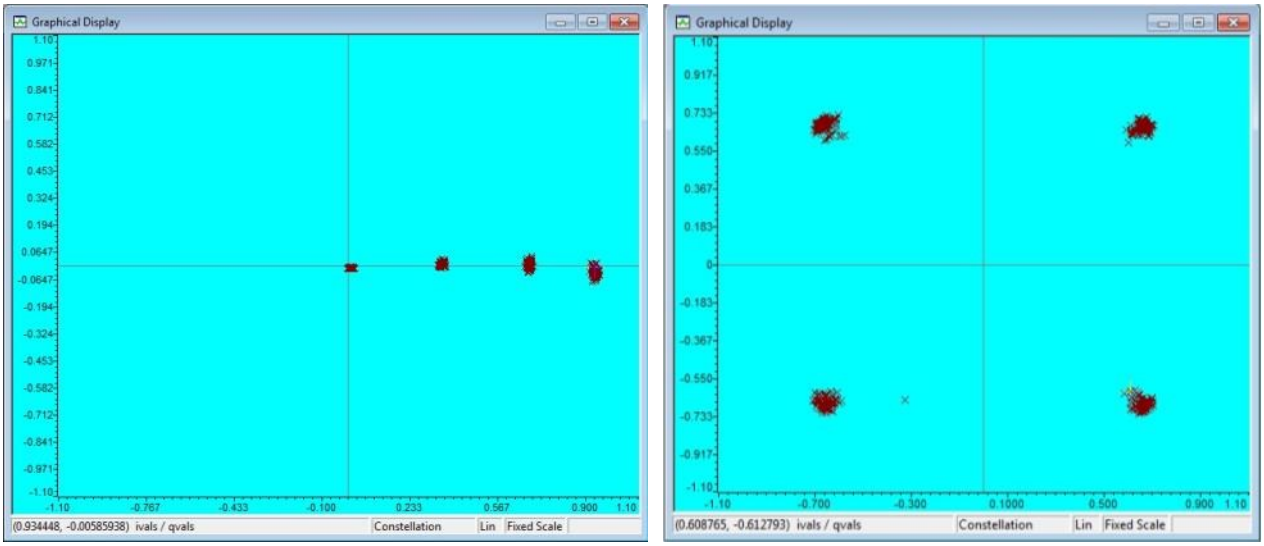

Figure 5.20: 4-ASK and QPSK constellations at the receiver

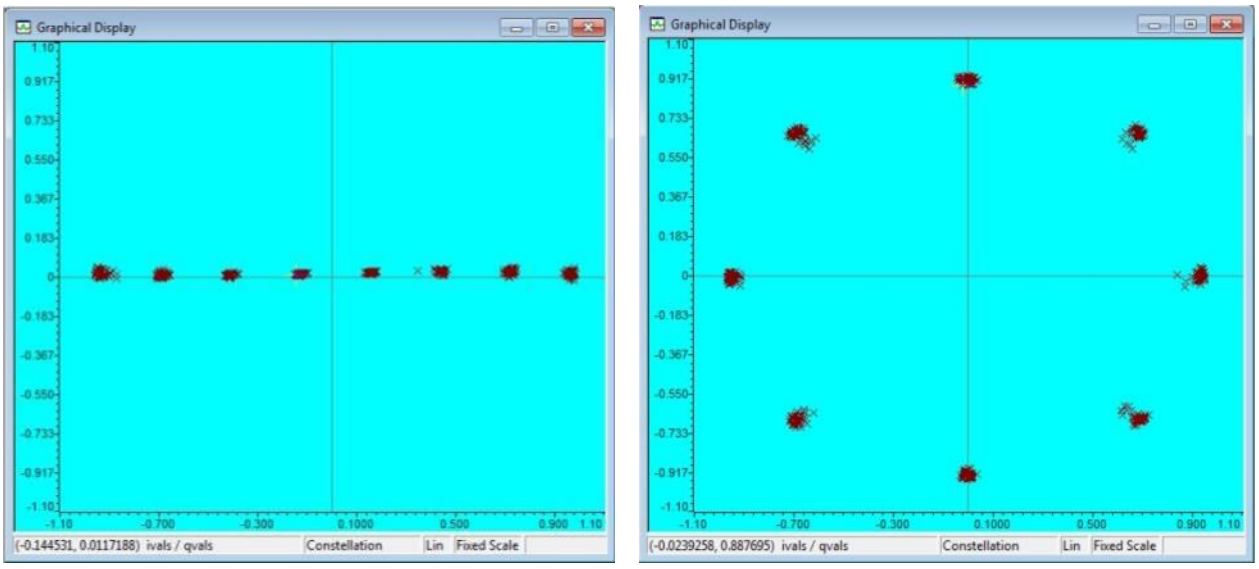

Figure 5.21: 8-PAM and 8-PSK constellations at the receiver

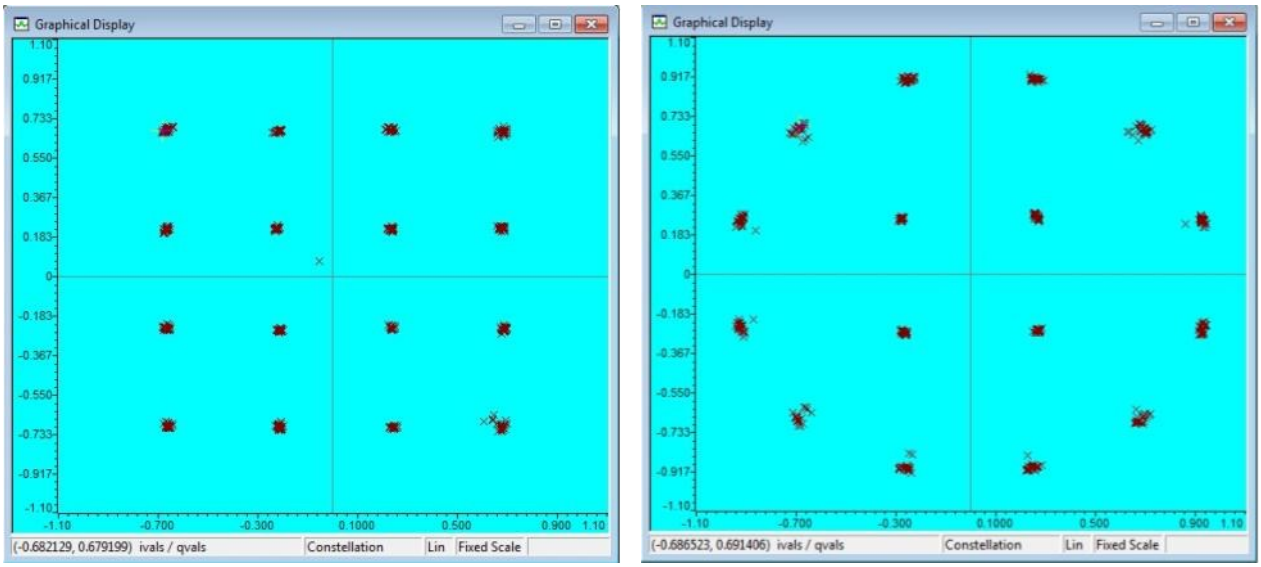

Figure 5.22: 16-QAM and 16-APSK constellations at the receiver

5.5 Wireless Link Performance

The second type of test that was performed with the transceiver was a wireless link test. As the main purpose of the software defined transceiver was from the beginning to perform wireless link bit error rate performance tests, it was essential to see what the base error would be with high SNR level. It was already determined from direct connection by cable of transmitter to the receiver that there is an extra bit error rate present due to timing error issue. However if the error

has a steady behavior in wireless link, it can be subtracted from overall error during performance tests, thus allowing calculation of actual performance. Two tests were performed for each modulation to determine if there is some consistency, and two different conditions were tested.

First condition tested was a direct line-of-sight test, and the setup is shown in Figure 5.23. Two monopoles were positioned about 3-4 meters apart. Monopoles were tuned to a center frequency of 1 GHz. Results of the tests are summarized in Table 5.1. For most modulations in

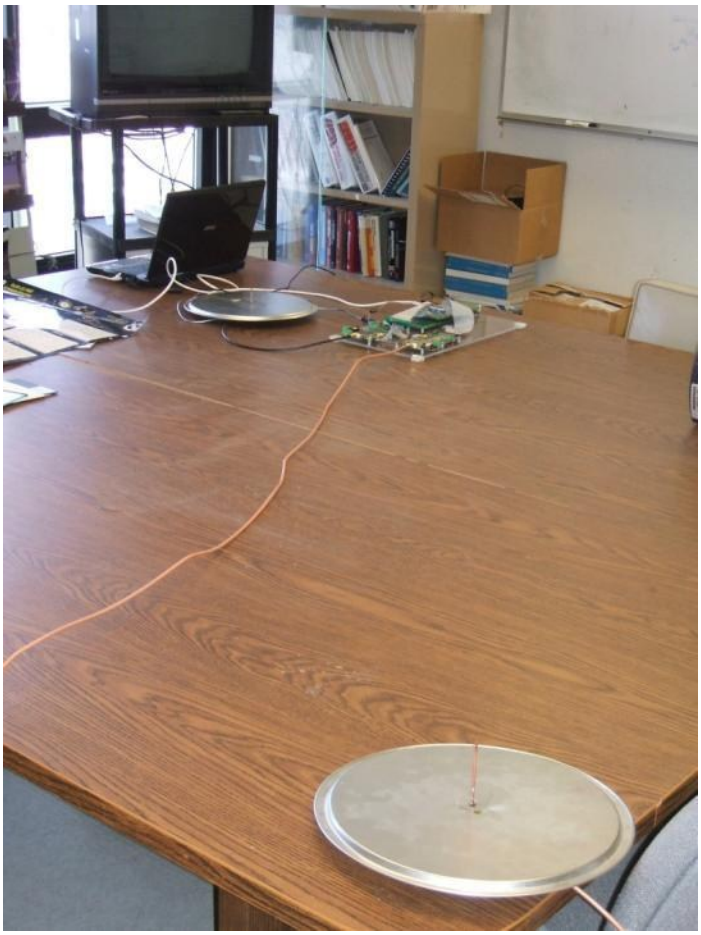

Figure 5.23: Test setup for line-of-sight test

Table 5.1 results are relatively consistent. An average error is around 2-3%. For better results more tests should be collected and averaged out.

Second condition tested was a no direct line-of-sight test, which was done to determine if there any reflections effect on the performance. The setup for the test is shown in Figure 5.24 and Figure 5.25. Receiving antenna was kept in the same location, but the transmitting antenna was positioned behind the book shelves. The results of this test are shown in Table 5.2. Compared to the results in Table 5.1 the steady error rate is the same, and even more consistent.

However, consistency can also be contributed to the fact that different error limits for the time synchronizer were used from the recommended ones in Table 3.2.

Modulation	Error Test 1	Error Test 2
BASK	0.01643164	0.0122165
BPSK	0.0152784	0.01331759
QPSK	0.0072194	0.0188586
4-ASK	0.014609	0.014479
8-PAM	0.032588	0.032715
8-PSK	0.047863	0.0271668
16-QAM	0.0323732	0.024467
16-APSK	0.0248244	0.025308

Table 5.1: Wireless link transceiver performance with direct line-of-sight

Figure 5.24: No line-of-sight test setup, first antenna position

Figure 5.25: No line-of-sight test setup, second antenna position

Modulation	Error Test 1	Error Test 2
BASK	0.0165107	0.0164654
BPSK	0.0138805	0.0130997
QPSK	0.021261	0.0235587
4-ASK	0.023284	0.020410
8-PAM	0.0367070	0.03299138
8-PSK	0.030490	0.0347004
16-QAM	0.0252258	0.02521068
16-APSK	0.027396438	0.02745781

Table 5.2: Wireless link transceiver performance with no line-of-sight

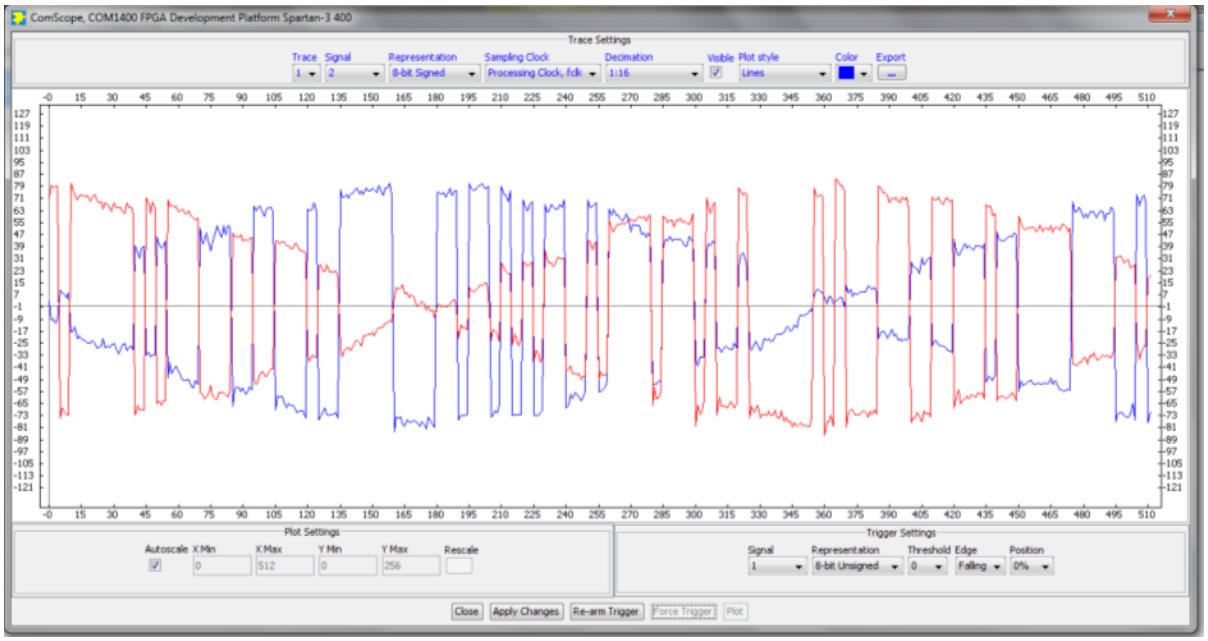

Figure 5.26: Signal after CIC filter in ComScope for line-of-sight test

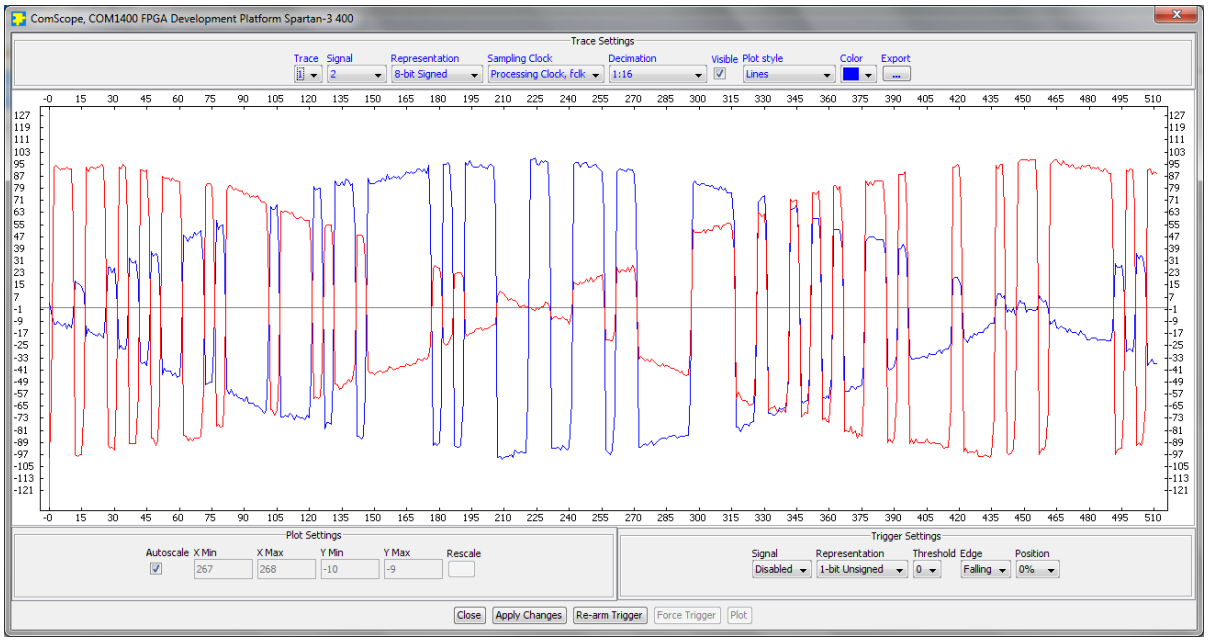

Figure 5.27: Signal after CIC filter in ComScope for no line-of-sight test

For both tests the SNR value was very high, the evidence of which can be seen in Figure 5.26 and Figure 5.27. The output signal after CIC filter is very clear and contains almost no noise, which means the real bit error rate should be almost, if not zero. Because of this fact, the system cannot be used directly to determine bit error rate. It might be possible instead to calculate bit error rate using 2 measurements. First measurement needs to be done with the

reliable antenna with known performance, while second measurement needs to be done using antenna under test. Relative difference will indicate performance change. This method was not tested for this project however. Additionally there is no equalization present in this system, which means that multipath will cause significant performance drop. To avoid multipath effects, data rate can be reduced, or the system can be tested inside of the anechoic chamber.

6. Conclusions and Future Work

6.1 Conclusion

The system presented in this thesis work has been constructed and tested under different conditions. Software was written for the DSP processor and FPGA. DSP transmitter is working correctly based on the output of the baseband signal at the oscilloscope and spectrum output at the spectrum analyzer. Assuming there is no timing error present, graphs show that DSP receiver performs close to theoretical value for the bit error rate. As the constellation size starts to increase performance drops, which could be caused by phase ambiguity that is constantly changing. With the complete wireless link it was determined that even with a very high SNR system still has 1%-4% bit error rate. Because of this, direct performance measurements of the antenna cannot be done on this system. It is possible to do relative comparison instead between a calibrated antenna and an antenna under test as described in the section 5.5.

6.2 Contributions

Major contribution of this thesis is the creation of a new software defined radio system with a lot of flexibility and extra capabilities for a much lower price with comparable systems. The presented system cost around $1400, which is comparable to USRP2. Compared to USRP2, presented system has a direct access to FPGA, and is easier to reconfigure. Additionally presented system has an audio ADC and DAC, which are also available in Lyrtech and WARP platform, but Lyrtech and WARP platforms are more expensive. Presented system is easily reconfigurable, as ComBlock modules can be purchased for different frequency bands, and a bigger FPGA can be purchased. Overall system is very educational, because signal can be tracked at every step of the transceiver. It is a great system to use to teach students theory, and show how problems such as frequency offset can be resolved.

Most of the commercially available systems use FPGA or DSP with general purpose CPU available on it, such that a common operating system, such as Linux, can be installed on them. This project presents an alternative software defined radio platform, where instead of complex OS, a simpler real-time OS used. Therefore another contribution is the development of code using TI BIOS, instead of Linux based code. Finally, most of the application notes create an

interface between DSP and FPGA using External Memory Interface. This project developed a reliable FPGA-DSP interface using HPI interface instead.

6.3 Future Work

There is a lot of the work that still needs to be done for this project, and there are a lot of additional projects that can be designed using existing system. The primary problem that causes system to perform poorly even under high SNR conditions is poor timing error correction. Error performance for the current system can be made better in multiple ways. A forward error correction can be added to eliminate errors caused by timing, but error correction will take additional processing resources, and most importantly it will not give a good way to measure bit error rate performance. Instead it would create reliable data communication. Another way to solve timing error problem is to try and use a more accurate receiver, transmitter, and ADC clocks. Finally, timing error correction can be added that works better than currently implemented one. For a fixed data rate, such as this system, it is possible to implement a clock recovery that does not require feedback.

During testing it was common to see that bipolar signaling can have a phase ambiguity which will cause symbols to be incorrectly demodulated. There was no significant impact of this problem for smaller modulations such as BPSK, and QPSK, but for larger constellations, extra errors would be made. Therefore one of the other modifications necessary in this project is to add a more reliable synchronization to eliminate extra errors.

Current system has a single data rate, and uses square pulses. Square pulses are not the most spectrally efficient pulses, and thus pulse shaping can be added to receiver and transmitter. A data rate can be made variable as well, by changing DAC clock, and using different amount of samples to map each symbol. Currently receiver can only work with linear modulations, because non-linear modulations require additional resources to be used. It would be useful to implement non-linear modulation receiver as well.

A maximum input signal strength that uses full ADC range is -54 dBm, which will still produce a signal-to-noise ratio around 40-50 dB. With such large SNR, it is very hard to test bit error rate performance. Therefore a second, digital AGC needs to be added to DSP or FPGA to make sure signal is between -1 and +1. An AGC control method presented in this project does

not work as expected, possibly due to AGC control voltage changing too fast. Even though AGC voltage changes logarithmically, actual AGC output is more PWM-like. Because of that it is better to build an AGC that produces PWM output instead of multiple voltage levels. As voltage controls amplification in decibels already, it is not logical to control a logarithmic amplifier using logarithmic scale. Another possible change because of that is to change AGC from logarithmic scale to linear scale.

Although the system does not work completely as desired it is still a valuable device for other purposes. DSP board contains additional peripheral inputs and outputs, such as audio ADC and DAC, Ethernet, DIP switches, and parallel and serial ports can be implemented as well. Audio devices can be used to build a complete voice transceiver, or to test performance of different audio codecs. Ethernet is already used for data communication between PC and DSP, but it can also be used to transform this system into a wired-to-wireless link converter. ComBlocks can be purchased for different frequency bands and with different sampling rates. Additional ComBlocks, such as another FPGA board, can be added to the DSP in order to increase input and output capabilities.

Bibliography

[1] M. Manteghi, "A Switch-Band Antenna for Software-Defined Radio Applications," *Antennas and Wireless Propagation Letters, IEEE,* vol. 8, pp. 3-5, 2009.

[2] J. G. Proakis and M. Salehi, *Digital communications*, 5th ed. Boston: McGraw-Hill, 2008.

[3] S. S. Haykin, *Introduction to analog and digital communications*, 2nd ed. Hoboken, NJ: Wiley, 2007.

[4] R. Yates and R. Lyons, "DC blocker algorithms," *IEEE Signal Processing Magazine,* vol. 25, pp. 132-134, Mar 2008.

[5] *TMS320C6455 DSK Technical Reference,* Spectrum Digital Inc., Stafford, TX, 2006

[6] *ComBlock COM1400 Specifications,* Mobile Satellite Services, Gaithersburg, MD, 2005

[7] *ComBlock COM2001 Specifications,* Mobile Satellite Services, Gaithersburg, MD, 2006

[8] *ComBlock COM3002B Specifications,* Mobile Satellite Services, Gaithersburg, MD, 2009

[9] *ComBlock COM4002D Specifications,* Mobile Satellite Services, Gaithersburg, MD, 2005

[10] E. B. Hogenauer, "An Economical Class of Digital-Filters for Decimation and Interpolation," *IEEE Transactions on Acoustics Speech and Signal Processing,* vol. 29, pp. 155-162, 1981.

[11] *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide (SPRU732H),* Texas Instruments, Dallas, TX, 2008

[12] *TMS320C6455 Chip Support Library API Reference Guide,* Texas Instruments, Dallas, TX, 2006

[13] *TMS320C645x DSP External memory Interface (EMIF) User's Guide (SPRU971C),* Texas Instruments, Dallas, TX, 2008

[14] *TMS320C645x DSP Enhanced DMA (EDMA3) Controller User's Guide,* Texas Instruments, Dallas, TX, 2007

[15] *EDMA3 Low Level Driver,* DSP/BIOS Integration Workshop

[16] *TMS320C645x DSP Host Port Interface (HPI) User's Guide (SPRU969B),* Texas Instruments, Dallas, TX, 2007

[17] *TMS320C645x DSP 64-Bit Timer User's Guide (SPRU968),* Texas Instruments, Dallas, TX, 2005

[18] *TMS320C64x+ IQmath Library User's Guide (SPRUGG9),* Texas Instruments, Dallas, TX, 2008

[19] *Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2),* ETSI Standard EN 302 307, 2009

[20] K. Daniel *et. al.,* "Creating a Second-Level Bootloader for FLASH. Bootloading on TMS320C6000 Platform With Code Composer Studio," Texas Instr., Dallas, TX, Application Report (SPRA999A1), May. 2006.

[21] *TMS320C6000 Network Developer's Kit (NDK) Support Package for DSK6455 User's Guide (SPRUES4A),* Texas Instruments, 2008

[22] *COM-4002 L-Band TX Schematics,* Mobile Satellite Services, Gaithersburg, MD, 2007

[23] *TMS320C64x+ DSP Megamodule Reference Guide (SPRU871J),* Texas Instruments, Dallas, TX, 2008

[24] *TMS320C6000 DSP/BIOS User's Guide (SPRU303B),* Texas Instruments, Dallas, TX, 2000

[25] L. Litwin (2003, Apr. 2). *Using PLLs to Obtain Carrier Synchronization: Part 2* [Online]. Available: http://www.eetimes.com/design/communications-design/4138454/Using-PLLs-to-Obtain-Carrier-Synchronization-Part-2

[26] New Wave Instruments (2010, Apr. 5). *Linear Feedback Shift Registers* [Online]. Available: http://www.newwaveinstruments.com/resources/articles/m_sequence_linear_feedbacf_shift_register_lfsr.htm

[27] K. Gunasekara (2004, Nov. 26). *UDP Send and Receive using threads in VB.NET* [Online]. Available: http://www.codeproject.com/KB/IP/UDP_Send_Receive.aspx

[28] *MATLAB Early-Late Gate Timing Recovery Simulink Block,* Natick, MA, 2011

[29] H. Tang, "Notes on DPLL", unpublished

[30] J. G. Proakis and D. G. Manolakis, *Digital signal processing*, 4th ed. Upper Saddle River, N.J.: Pearson Prentice Hall, 2007.

[31] M. Donadio (2011). *How to Generate White Gaussian Noise* [Online]. Available: http://www.dspguru.com/dsp/howtos/how-to-generate-white-gaussian-noise

[32] *ComScope User & Programming Manual,* Mobile Satellite Services, Gaithersburg, MD, 2005

[33] http://www.softwaredesignsolutions.com/flashburn_how.aspx

[34] http://www.spectrumdigital.com/product_info.php?cPath=33&products_id=99&osCsid=1fc55a84c122358fc49a2a123c9d1261

Appendix A: Deriving 4-ASK BER Using Union Bound

The expression for the Union Bound is

$$P_s \leq \frac{1}{M}\sum_{i=1}^{M}\sum_{j=1,j\neq i}^{M} Q\left(\frac{d_{ij}}{\sqrt{2N_0}}\right) \tag{A.1}$$

In expression (A.1) d_{ij} is the distance between symbols in terms of E_b, M the total number of symbols in the constellation, and P_s is the probability of symbol error. Probability of bit error can be approximated as $P_b \approx \frac{1}{\log_2 M} P_s$ if all symbols have equal probability. Expression (A.1) can be simplified as only directly adjacent neighbors for any given symbol will most likely contribute to the error. The simplified expression of Union Bound is

$$P_s \leq \frac{1}{M}\sum_{i=1}^{M}\sum_{j\in A_i}^{M} Q\left(\frac{d_{ij}}{\sqrt{2N_0}}\right) \tag{A.2}$$

where A_i is the set of symbols directly adjacent to symbol *i*. The simplified Union Bound is very accurate, especially at E_b/N_0 valued of 0 dB or higher. To calculate bit error probability for 4-ASK modulation (expression is not available in [2]), first a set of symbols is written in terms of integer values: [0, 1, 2, 3]. Using these values an average energy per symbols is derived:

$$E_s = \frac{0^2 + 1^2 + 2^2 + 3^2}{4} = \frac{14}{4}$$

Which can be converted to the E_b as

$$E_b = \frac{1}{\log_2 M} E_s = \frac{14}{8}$$

Distance *d* between symbols 3 and 2 (and other pairs as well too), for example, will be

$$d = \frac{3}{\sqrt{E_b}} - \frac{2}{\sqrt{E_b}} = \frac{1}{\sqrt{E_b}} = \sqrt{\frac{8}{14}E_b} = \sqrt{\frac{4E_b}{7}}$$

Expression for the symbol probability can be then calculated:

$$P_b \approx \frac{1}{2} * \frac{1}{4}\left(Q\left(\frac{\sqrt{\frac{4}{7}E_b}}{\sqrt{2N_0}}\right) + 2Q\left(\frac{\sqrt{\frac{4}{7}E_b}}{\sqrt{2N_0}}\right) + 2Q\left(\frac{\sqrt{\frac{4}{7}E_b}}{\sqrt{2N_0}}\right) + Q\left(\frac{\sqrt{\frac{4}{7}E_b}}{\sqrt{2N_0}}\right)\right)$$

$$P_b \approx \frac{6}{8}\left(Q\left(\sqrt{\frac{4E_b}{14N_0}}\right)\right) = \frac{3}{4}Q\left(\sqrt{\frac{2E_b}{7N_0}}\right)$$

The same method can be used to derive bit error rate probabilities for the other constellations as well.

Appendix B: MALTAB Code for Generation of Memory Data

```
clear all;
close all;

d = mseq([1 0 0 1 1 1], [1 1 1 1 1 1], 63, 0);

Fs = 40E6;                          % Sample Rate
Rb = 1E6;                           % Bit rate
Ns = Fs/Rb;                         % Ns samples/bit
[xn, pulse, Es] = PulseShape(d, 'SQAR', Ns);
xn = xn(1:length(d)*Ns);

%n = sigma*randn(1, length(xn)) + 1i*sigma*randn(1, length(xn));
n = 0;

% Add noise and normalize it
yn = xn + n;

% Add frequency offset
Fif = 10E6;                 % IF frequency
f =  Fif + Fs/length(xn);      % Frequency in Hz
t1 = length(d)/Rb;
t = 0 : 1/(Ns*Rb) : t1 -1/(Ns*Rb);

% introducing a phase offset of n degrees
phiDeg = 30;
phiRad = phiDeg*pi/180;

% Limit the input magnitude
yn = yn*sqrt(Ns);
yn = yn*(408/512);

% Modulate signal as it comes from ADC
I = yn.*cos(2*pi*f*t + phiRad);
Q = yn.*sin(2*pi*f*t + phiRad);

% Generate data files to be used in simulation
I = round(I*512);
I = I + 512;

Q = round(Q*512);
Q = Q + 512;

q = quantizer('ufixed', 'nearest', 'saturate', [10 10]);
```

```
data_i = num2bin(q, I/1024);
data_q = num2bin(q, Q/1024);
data = strcat(data_i, data_q);
memInit('TestBPSKSignal', data, 20, Ns*length(d));
```

Appendix C: MATLAB code for PLL testing

```
clear all;
close all;

EbNo = 10^(10/10);
sigma = sqrt(1/2/EbNo);

Rb = 1E6;       % Bit rate
Fs = 40E6;      % Sampling frequency
Fif = 10E6;     % IF freqeuncy

d = mseq([1 0 0 1 1 1], [1 1 1 1 1 1], 63, 0);

Ns = Fs/Rb;                             % Ns samples/bit
[xn, pulse, Es] = PulseShape(d, 'SQAR', Ns);
xn = xn(1:length(d)*Ns);

%n = sigma*randn(1, length(xn)) + 1i*sigma*randn(1, length(xn));
n = 0;

% Add noise and normalize it
yn = xn + n;

% introducing a phase offset of n degrees
phiDeg = 30;
phiRad = phiDeg*pi/180;

% Phase offset
yn = yn*exp(1i*phiRad);

% Add frequency offset
f  = 5000;              % Frequency in Hz
t1 = length(d)/Rb;
t = 0 : 1/(Ns*Rb) : t1 -1/(Ns*Rb);
k = exp(-1i*2*pi*f.*t);       % Generate frequency offset
yn = yn.*k;

yn = yn*sqrt(Ns);
yn = yn*(408/512);

% PLL variables
z1 = 0;
z2 = 0;
z3 = 0;
z4 = 0;
```

```
 % Phase error
phiHat = 0;

for ii = 1:length(yn)

    % Derotate the phase
    gen = cos(phiHat) + 1i*sin(phiHat);
    yn(ii) = yn(ii)*gen;

    % Decision directed PLL loop
    e = yn(ii)*sign(real(yn(ii)));
    e = atan2(imag(conj(e)), real(e));

    % Loop filter output (Proportional-Integral)
    phiHat = 0.0170535*e + 0.000785*z1 - 0.0162685*z2 + z3*2 - z4;

    % Make sure phase is within [-2pi, 2pi]
    if(phiHat >= 2*pi)
        phiHat = phiHat - 2*pi;
    elseif(phiHat <= -2*pi)
        phiHat = phiHat + 2*pi;
    end

    % Update states
    z2 = z1;
    z1 = e;
    z4 = z3;
    z3 = phiHat;
end

figure;
plot(real(yn)); hold on; plot(imag(yn), 'r');
```

Appendix D: Command file for hex6x utility

```
Debug/RX_TX_BIOS.out              /* input COFF file */
-map RX_TX_BIOS_hex.map           /* generate hex map map file */
-a                                /* ASCII HEX format */
-memwidth  8                      /* 8-bit wide ROM */

-boot                             /* create a boot table for all initialized sects*/
-bootorg 0xB0000400               /* address of the boot/copy-table */
-bootsection .boot_load 0xB0000000 /* section containing our asm boot routine */

ROMS
{
FLASH: org = 0xB0000000, len = 0x80000, romwidth = 8, files = {RX_TX_BIOS.hex}
}
```

Appendix E: Source Code

The full source code for the project can be found at http://antenna.ece.vt.edu/~v_podosinov/SourceCode_Podosinov_Thesis2011.zip or by contacting author at v_podosinov@vt.edu or the author's advisor at manteghi@vt.edu. Latest ComBlock files can be found at their website at www.comblock.com. Source code tree structure is provided below

- SourceCode_Podosinov_Thesis2011
    - DSP_Code
        - RX_TX_BIOS
            - edma.h // Header file and code for the EDMA functions
            - edma.c // based on the LLD library
            - FlashBurnConfig.cdd // Configuration file for FlashBurn
            - FlashInit.cmd // Command file to convert DSP object file to HEX file used by FlashBurn. Shown in Appendix D
            - main.h // Main header file containing all variables used and declarations
            - RX_TX_main.c // Main program file implementing all functions and threads
            - network_driver.h // Header file and code for working with network communication.
            - network_driver.c
            - userlinked.cmd // User linker file with additional memory references in addition to the BIOS ones
            - rx_tx_bios.tcf // BIOS configuration file
            - RX_TX_BIOS.pjt // Code Composer Studio 3.3 project file
        - bootDSK6455.asm // Bootloader for Flash memory operation. Code is similar to the one shown in [20]
        - box_muller_table.h // Box-Muller variable sqrt(-2*ln(U1))
        - complexIQmath.c // Functions implementing complex math operations and a slicer function for DD-PLL in DSP

        - complexIQmath.h // Header file containing function prototypes and expected symbols for DD-PLL in Q2.14 format
        - noise_valsI.h
        - noise_valsQ.h // Normally distributed random values with 0 mean and 1 standard deviation in Q2.14 format
    - FPGA_Code
        - Source Code // Source Verilog and VHDL files
            - AGC.v // Automatic Gain Control with DC Filter and Memory
              AWGN.v // Unused AWGN generator for FPGA
            - capture_8_bit_words.vhd
            - capture_16_bit_words.vhd // Files used by ComScope
            - CICFilter.v // CIC Decimator
            - COM1400.ucf // File describing all FPGA connections to pins
            - com1400.vhd // TOP MODULE for FPGA
            - COMSOPE.VHD // ComScope implementation from ComBlock
            - DAC.vhd // AGC DAC Communication block
            - DAC_TEST.vhd // Test block for AGC DAC from ComBlock
            - DCFilt.v // DC filter used in the AGC block
            - DDPLL.v // Unused Decision Directed PLL for FPGA
            - DECIMATE.VHD // Decimation modules used by ComScope
            - DelayAdjust.v // Adjuster of the delay for TED
            - DSPComm.v // FPGA to DSP communication using HPI block
            - MF.v // Matched filter implemented as running average
            - proc_clk.vhd // Main FPGA clock DCM
            - RXClockGen.v // Receiver clock generator module
            - slicer.v // Unused slicer module for DD-PLL
            - TED.v // Timing error detector module
            - TestGen.v // Module that replays signal from the memory
            - USB20.NGC // USB 2.0 connected used in the project
        - Coregen files // Cores generated by Xilinx ISE 11.5 for the project

- MATLAB
  - apsk16sim.mat // Simulation results of 16-APSK from EbNo = 3-12 dB in 0.5 dB increments
  - ccorr.m // Cross-correlation calculator for PRBS
  - MA_DCFilter.m // DC Filter simulator
  - memInit.m // Script to write COE and MEM files
  - ModGen.m // Signal generation to be put into memory
  - mseq.m // m-sequence generator
  - PLL.m // PLL simulator for loop filter tests
  - PulseShape.m // Script to generate different pulse shapes (Dr. Buehrer code)
- VB_GUI
  - // Visual Studio 2010 Visual Basic .NET Project files and source code based on .NET Framework version 4.