Digital Phase Detection
In a Variable Frequency RF System

By

Adam Molzahn

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

December 2005

ABSTRACT

Digital Phase Detection
In a Variable Frequency RF System

By

Adam Molzahn

In cyclotron control systems, accurate phase and amplitude information derived from the radio frequency voltages applied to the accelerating electrodes (dees) is crucial to the successful operation of the accelerator. A small tolerance of $\pm 0.1\%$ in amplitude jitter and $\pm 0.05^{\circ}$ in phase jitter of the sinusoidal radio frequency drive signal is required for the measurements. This thesis focuses on the design and implementation of an FPGA-based phase meter module with a discussion regarding further additions to convert the module to a fully functional phase and amplitude control system.

Using inphase and quadrature (I and Q) vector data gathered by digitizing the electrode waveforms, the phase and amplitude are calculated and compared to a reference signal.

The phase information from each module is used in the existing cyclotron control system to replace the obsolete analog vector voltmeters and provide a display for each dee station.

This thesis could not have happened without the love and support of Destinee and Ashton, who have helped keep me sane through all of life’s twists and turns.

ACKNOWLEDGMENTS

Many people helped me get where I am today. I'd like to thank John Vincent, first and foremost, for his unending support and cynicism which drove me to try and meet and exceed his expectations. Thanks for holding me to a higher standard. I would also like to thank Dale Smith for giving me the opportunity to learn from some of the best technicians and engineers in the accelerator field. For board layout and design problems, I have to thank Brian Drewyor for teaching me the ropes. I would not have made any progress on the research for this project had it not been for the discussions I had with Michael O'Farrell, he was a great sounding board with lots of excellent feedback. Without the help of Mark Davis the programming side of this project would have been a bear. And Larry Doolittle, whose work was the seed that started this project, deserves kudos for putting up with my relentless questions. Finally, I'd like to thank Leo Kempel for getting me here and being undyingly optimistic.

On a special note, I'd like to thank my wife Destinee and my son Ashton. They have helped me more than they could possibly know. You are my motivation.

Thank you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

The successful operation of the superconducting cyclotrons at Michigan State University depends heavily on the ability of the RF controls system to precisely regulate the voltage applied to the accelerating electrodes (dees, beam buncher, etc). For acceleration, the cyclotron employs three electrodes, called dees, which are nominally 120° out of phase with each other. High accuracy phase measurements are necessary to allow the cyclotron operators to precisely set the phase between the dees to tune the beam[1].

The purpose of this thesis project is to develop a digital phase meter to accurately read the phase between the three dees and the beam buncher on the K500 and K1200 cyclotrons. This thesis presents a detailed discussion of the theory, hardware and software desired to create a high quality phase meter.

In the cyclotron control system's current incarnation, one station is set up to regulate each of the three dees in the cyclotron and one station is set up to regulate the beam buncher. Three external phase meters read the phase of the RF between the A and B stations, the A and C stations and the A station and the beam buncher on each cyclotron[2]. This module is meant to replace those three obsolete analog phase meters with a digital module that will read each phase and report the readings back to the control system. Beyond this project, additional technologies have been added to facilitate replacing the multiple existing analog

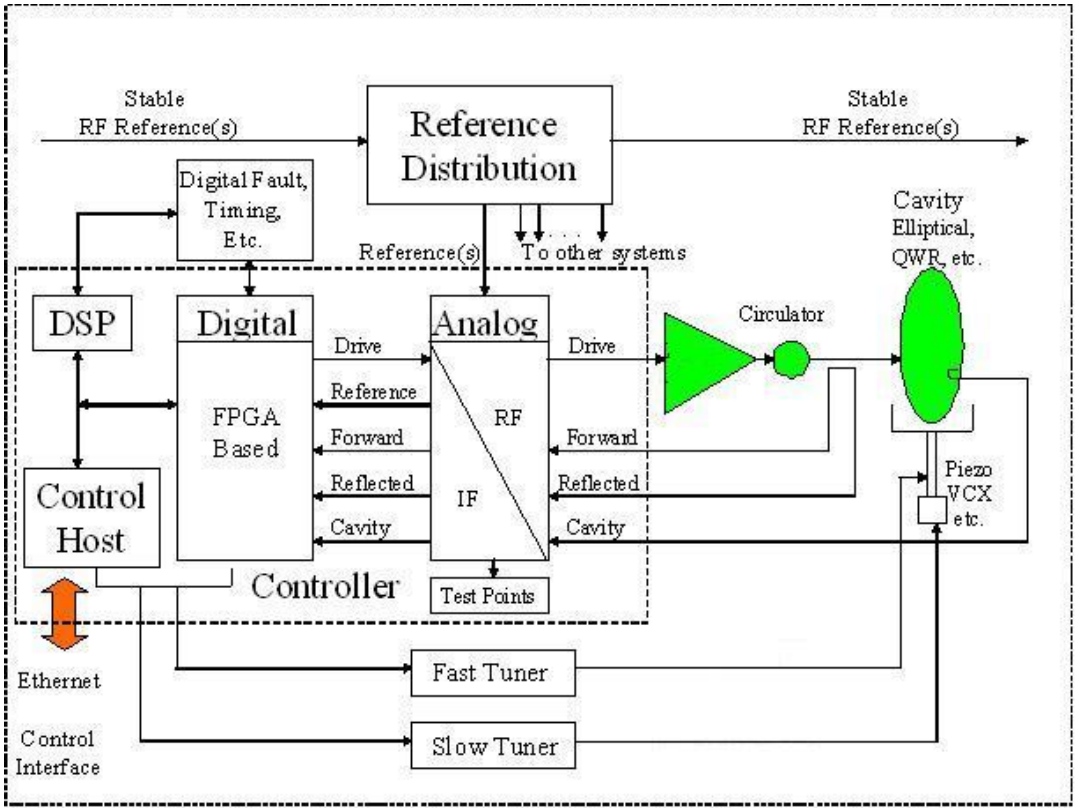

Figure 1.1. System Overview.

tuning and regulation modules with one digital system per dee that monitors, tunes and regulates all within a single module. Figure 1.1 shows an overview of the entire control system. This thesis will detail the components contained within the dotted lines labeled Controller.

Construction of a phase meter requires seven main stages: (1) a fixed attenuator to deal with the high maximum voltages of the low level radio frequency (LLRF) signals from the cyclotron, (2) a mixer stage where the RF signals are mixed from their variable frequencies to an intermediate frequency, (3) a variable digital attenuation stage to handle the wide range of LLRF signal levels, (4) a fixed amplification stage to condition the signal to the full scale of the analog to digital converters (ADCs), (5) an ADC stage to sample and digitize

the waveforms, (6) a field programmable gate array (FPGA) stage to process the raw data, and (7) a microprocessor stage to deal with communications, calibration and controls. The remaining components are intended to extend the functionality of the module beyond a simple phase meter and will be discussed but not fully utilized in the final implementation. For a complete schematic overview see Appendix D, Figure D.1.

Five RF inputs are included on the front of the module. One is for the reference, which is used to control the phase lock loop and synchronize the clocks that control the FPGA and ADC sample timing. The reference can be any frequency, f_{ref}, as long as it can be related to the ADC sampling frequency, f_s, according to the following formula,

$$f_{ref} = \frac{N}{R} * f_s \tag{1.1}$$

where N and R are both integers[3]. The reference signal is digitized; therefore it is beneficial to make

$$f_{ref} = \frac{f_s}{4} \tag{1.2}$$

to generate the correct type of vector data[4], which will be discussed later. Three of the other inputs are the RF signals from which phase information is to be extracted. In the cyclotron, these signals range from 9MHz to 27MHz with voltage levels from -7dBm up to +33dBm. This module is designed to handle all of these signals without modification. However, by changing the mixers, some filters and the fixed input attenuators, it can be extended to work in virtually any system. The final input is a local oscillator (LO) signal that is used in conjunction with the mixers to shift the frequency of the RF signals whose phases are desired to a common intermediate frequency (IF).

The front panel also includes a high speed DAC output that is for use as the RF control signal once that capability has been added to the module. An RF On/Off input and a reset input are included as interlock signals from the control system. A fault signal output and a fast tuner signal output make up the last two connections on the front panel. The back panel connections include an Ethernet plug, a miscellaneous connector and a NIM crate power connector. A CPU reset button on the back panel is connected to the microprocessor to allow the module to be reset manually. LEDs are included to indicate CPU activity, RF status and module readiness are included to give quick feedback as to the state of the module.

The maximum RF voltages presented to the input of the module are too high for the mixers to handle without being overdriven. Therefore, a fixed high power attenuator using standard surface mount resistors is designed to match the maximum RF voltage to the maximum input voltage of the mixers. This makes the attenuator reconfigurable for any system specifications while easily handing the power requirements

In the mixer stage, three mixers are used to mix the LO with the three RF inputs to create the IF, which is sampled and manipulated digitally. A high quality frequency synthesizer phase locked to the reference signal creates the LO frequency. For this system, the 10MHz phase reference on the back of the signal generator is the module reference and is already locked to the LO.

Due to the frequency dependence and operating requirements of the cyclotron, the voltage levels at the output of the mixers are not constant. Therefore, after mixing, the signals enter the digital attenuation stage to condition them for amplification. Variable digital attenuators are controlled by a 6-bit word from the FPGA.

The signals pass through a fixed amplification stage to condition them to be sampled by the ADCs. A chain of RF amplifiers and attenuators condition the signal levels to match them to the input requirements of the ADCs. It is imperative they are matched as closely as possible to full-scale to utilize all of the precision of the ADCs without overdriving the inputs. Low amplitudes result in a loss of sensitivity and accuracy with regard to changes in the signal while overdriving the inputs distorts the waves and corrupts the vector data being taken.

The ADC stage digitizes the signals in such a way that the samples taken can be considered the in-phase (I) and quadrature (Q) values of the vector representing the RF input signal [5, 6, 7]. I and Q map to the polar coordinate system as the real and imaginary axes, respectively, and can be used to directly calculate the phase and magnitude of the sampled signal.

The digitized signals are read by the FPGA in real time. A history buffer keeps track of past inputs and outputs and is used for filtering and storing samples as I and Q data. This data is sent to the microprocessor to determine the phase between the RF inputs. The FPGA also takes care of interlocks, digital to analog converter output and various other housekeeping tasks that will be detailed later.

The microprocessor is the heart of the system, manipulating the data and handling communication. The bus controller allows data to be shared asynchronously with the FPGA so that phase and magnitude can be calculated and other components on the PC board can be configured. Various user interface panels display board parameters and chip settings over a telnet connection hosted here as well. Configuration data is either generated or loaded by the microprocessor and sent via the serial programming interface to set up the rest of the

supplemental chips.

The software used to do the work detailed in this thesis includes Xilinx Integrated Software Environment 7.1i for Verilog code development and compilation, Protel DXP 8.3 SP 3 for schematic capture and printed circuit board layout and design, AutoCAD 2005 for layouts and designs, Chipscope Pro 6.3i for FPGA verification, MATLAB R14 for graphing and numerical manipulation, NMAKE 6.00 for C-code compliation and Dynamic C 8.61 for ZWorld C-code development.

Test equipment included a Rhode and Schwarz 3.3GHz signal generator, 2 PTS 250 frequency synthesizers, a Hewlett Packard 8508A vector voltmeter and a Hewlett Packard E4402B spectrum analyzer.

CHAPTER 2

Module Input and Mixing

2.1 Description of the Input Stage

This phase meter must be able to accept a wide number of input frequencies and voltage levels. Specifically designing for the cyclotron, a front end was developed that could accept frequencies from 9 to 27MHz at amplitudes varying from 100mVRMS (-7dBm) up to 10VRMS (+33dBm), but also be configurable to other ranges[1, 7]. This was accomplished using an attenuator and mixer stage at the input of each RF channel of the module (Appendix D, Figure D.2). The mixer is used to convert the radio frequency (RF) input signal to a common intermediate frequency (IF). The maximum signal level that can be handled on the RF port of the mixer is +1dBm. Therefore, a fixed high power attenuator is necessary to match the maximum input voltage to that of the mixer. With a maximum of +33dBm coming into the module, -32dB of attenuation is required. A -29dB PI style attenuator (Figure 2.1) for a 50Ω system can be easily constructed using values for R_1, R_2 and R_3 based on the following equations:

$$R_1 = R_2 = \frac{1}{\frac{10^{\frac{dB}{10}}+1}{50*10^{\frac{dB}{10}}-1} - \frac{1}{R_3}} = 53.5\Omega \tag{2.1}$$

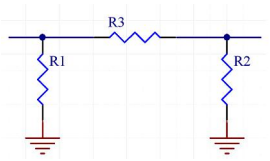

Figure 2.1. PI Attenuator.

$$R_3 = \frac{1}{2} * (10^{\frac{dB}{10}} - 1)\sqrt{\frac{50 * 50}{10^{\frac{dB}{10}}}} = 704\Omega \tag{2.2}$$

where dB is the amount of attenuation required for the attenuator[8]. This style attenuator allows for easy modification and can be designed to handle the large amount of power dissipated by using a high power resistor for R_1. The attenuator reduces the maximum level down to +4dBm and a standard low power 3dB RF attenuator can be used to match to the desired +1dBm.

2.2 Mixer Theory

In an ideal mixer, the RF input signal is multiplied by a local oscillator (LO) signal to create a new signal with sidebands equal to the sum and difference of the RF and LO frequencies [9, 10]. Specifically,

$$RF = A_1 sin(\omega_1 t + \theta_1) \tag{2.3}$$

$$LO = A_2 sin(\omega_2 t + \theta_2) \tag{2.4}$$

$$RF * LO = \frac{A_1 A_1}{2} cos((\omega_1 + \omega_2)t + \theta_1 + \theta_2) - \frac{A_1 A_1}{2} cos((\omega_1 - \omega_2)t + \theta_1 - \theta_2) \tag{2.5}$$

$$RF * LO = sum - difference \tag{2.6}$$

The phase of the RF signal, θ_1, is preserved and offset by $+\theta_2$ and $-\theta_2$ even though the frequency has changed. Hence, there is a one to one correspondence between the phase of the new signal and the phase of the old signal. The resultant waveform is filtered to select the desired sideband (IF). The recommended input signal level for the LO port on the mixer is +7dBm (0.5VRMS). With three input channel mixers and one output channel mixer, the signal generator connected to the LO port on the module must supply +19dBm (2.0VRMS). Standard signal generators cannot supply this much voltage, so a single stage amplifier and attenuator chain is employed. The gali-51 RF amplifiers used in this design have a 1dB compression point of +18.3dBm and a gain of +18dB. For this reason, the required input level to the module is reduced to +8dBm and is immediately attenuated by -9dB to -1dBm. The signal is then amplified up to +17dBm and distributed to the four mixers using a matched resistive voltage divider circuit. +17dBm is within the range of linear operation for the gali-51 amplifiers and supplies enough current to drive the mixers correctly by delivering +5dBm (0.4VRMS) at each mixer LO input. It is important that the signal path lengths for each LO trace are equalized on the PCB for each of the mixers on the RF input channels to make θ_2 the same. By ensuring that θ_2 is equal for each of the RF inputs, the resultant multiplication of sine waves produces three signals that are all offset by the same value. In this manner, the channel-to-channel phase is independent of the θ_2's.

2.3 Mixing and Harmonic Interference

Harmonic frequencies on the RF input can adversely affect the mixing process and the subsequent filtering. Assume the input signal contains harmonic frequencies (ω_{RF}, $2 * \omega_{RF}$,

Table 2.1. Harmonic Mixing for RF=9MHz, IF=50MHz, LO^-=IF+RF=59MHz, LO^+=IF-RF=41MHz.

Harmonic Freq.	$LO^- + h*RF$	$LO^- - h*RF$	$LO^+ + h*RF$	$LO^+ - h*RF$
9	68	50	50	32
18	77	41	59	23
27	86	32	68	14
36	95	23	77	5
45	104	14	86	4
54	113	5	95	13
63	122	4	104	22

$3*\omega_{RF}$, $4*\omega_{RF}$, ...$h*\omega_{RF}$) and that the LO contains only the fundamental frequency without harmonics. Any harmonic frequency that may mix back near the IF will be hard to filter out and will distort the phase. In the interest of finding the highest and lowest harmonic frequencies that might be a problem for this system in the cyclotron, it is advantageous to look at the low end of the cyclotron frequency scale which will have the closest spaced harmonics and will be the hardest to filter. Using a 9MHz RF and analyzing both the RF+LO and RF-LO frequencies table 2.1 can be generated. No matter the input frequency, the lowest harmonic that could potentially create a filtering problem is at a higher frequency for IF=LO-RF than for IF=LO+RF. Note that negative frequencies simply fold back into the positive realm with a 180° phase shift. A simple analysis of the frequencies created during mixing shows that certain RF harmonics can have sums or differences that will fold directly onto the IF frequency corrupting the true signal and causing phase error. Assume a signal enters the module containing the RF and the 2^{nd} harmonic of the RF. The harmonic mixes with the LO onto the IF and is then ideally filtered so that only the IF passes giving a signal of the form,

$$IF = A_1 e^{j(\omega_{IF}t+\phi_1)} + A_2 e^{j(\omega_{IF}t+\phi_2)} \tag{2.7}$$

with ϕ_1 the desired phase of the signal and ϕ_2 the phase due to the unwanted harmonic frequency. Manipulating this equation to determine the phase and amplitude of the IF gives

$$IF = e^{j\omega_{IF}t}(A_1\cos(\phi_1) + A_2\cos(\phi_2) + j(A_1\sin(\phi_1) + A_2\sin(\phi_2) \tag{2.8}$$

Rotating ϕ_1 and ϕ_2 so that $\phi_1 = 0$ and $\phi_2^{'} = \phi_2 - \phi_1$ yields an equation for the phase equal to

$$\theta_{IF} = \tan^{-1}(\frac{Im}{Re}) = \tan^{-1}(\frac{A_2\sin(\phi_2^{'})}{A_1 + A_2\cos(\phi_2^{'})}) \tag{2.9}$$

2.4 Evaluating the Effects of Harmonic Interference

Evaluating this expression for different values of $\phi_2^{'}$ shows that $\theta_{IF} = 0^\circ$ when $\phi_2^{'} = 0^\circ$ and $\theta_{IF} = \tan^{-1}(A_2/A_1)$ when $\phi_2^{'} = 90^\circ$. The argument of the inverse tangent will always be between 0 and $\pm A_2/A_1$, therefore θ_{IF} will be between 0 and $\pm\tan^{-1}(A_2/A_1)$. If $\phi_2^{'}$ does not vary with time, as is the case with a pure harmonic, then the phase read by the module will include a static error added in equal to θ_{IF}. However if $\phi_2^{'}$ varies with time, it will show up in the reading as phase noise, varying with time.

Further analysis of the magnitude gives the equation

$$Mag_{IF} = A_1\sqrt{1 + \frac{A_2}{A_1}^2 + \frac{A_2}{A_1}\cos(\phi_2^{'})} \tag{2.10}$$

If the ratio of the harmonic to the RF is low, the magnitude of the signal is nearly equal to the magnitude of the fundamental component. Otherwise, the magnitude of the signal varies based on the phase $\phi_2^{'}$ and the ratio of the amplitudes. Once again, if $\phi_2^{'}$ is a time

varying signal the magnitude will fluctuate. It is important to determine which harmonics will mix to the IF frequency so they can be prefiltered out before the mixing process. There are two different ways to select a LO frequency to produce the desired IF. In the first approach, the LO frequency is chosen so that IF=LO+RF. This will be referred to as the sum. In the second approach, the LO frequency is chosen so that IF=LO-RF. This will be referred to as the difference. Let

$$IF = RF + LO \tag{2.11}$$

Assume some harmonic of RF = h*RF can mix with the LO onto the IF where,

$$IF = h * RF - LO \tag{2.12}$$

IF can be rewritten as,

$$IF = h * RF - IF + RF \tag{2.13}$$

and certain RF frequencies at,

$$RF = \frac{2IF}{h+1} \tag{2.14}$$

will have problematic harmonics at,

$$h * RF = \frac{h}{h+1} * 2IF \tag{2.15}$$

The lowest frequency will be (h=2),

$$h * RF = \frac{4}{3} IF \tag{2.16}$$

The highest will be at (h$\rightarrow \infty$),

$$h * RF = 2 * IF \tag{2.17}$$

Using an LO such that the sum is kept and the difference is filtered out (IF = RF+LO) certain frequencies image to the IF following the equation,

$$h * RF = \frac{2 * h}{h + 1} * IF \tag{2.18}$$

which leads to

$$h = \frac{2 * IF}{RF} - 1 \tag{2.19}$$

with h=1 equal to the fundamental RF frequency, h=2 equal to the 2^{nd} harmonic, h=3 equal to the 3^{rd} harmonic and so on. The solution for h corresponds to a harmonic image of the RF that will mix exactly to the IF and cause phase noise or phase error, depending on its origin. If h is not an integer, then no harmonic image of the RF will mix exactly to the IF. Using an LO such that the difference is kept and the sum is filtered out (IF = RF-LO) yields a center band image frequency of

$$h * RF = \frac{2 * h}{h - 1} * IF \tag{2.20}$$

which leads to

$$h = \frac{2 * IF}{RF} + 1 \tag{2.21}$$

Examining the limits of both equations shows at h=2 the problematic harmonic frequency for the sum is $(4/3) * IF$ and for the difference is $4 * IF$. For $h = \infty$ both equations

converge on $2 * IF$. Using a low pass filter to remove any harmonic frequency greater than or equal to $(4/3) * IF$ would guarantee that no image of a harmonic would mix back onto the IF. This filtering is left up to the individual users because of the wide range of input filters that would be required for the module to work in a broad range of applications.

After mixing, a bandpass filter centered on the IF is required to condition the mixer output because certain harmonics that may have passed through the initial filtering process could mix to frequencies both higher and lower than the IF. In an ideal system with a band pass filter that only allows the center frequency to pass and sufficiently attenuates any other frequencies, this analysis would be complete, however in a non-ideal system the filter has some finite bandwidth. For this module, the bandwidth is defined as the frequency band in which the frequency content is attenuated by less than -20dBc (dB to carrier) as this would sufficiently reduce any unwanted signals to the point they would no longer cause significant errors. The pass band sets the minimum frequency for the RF because any frequency lower than the bandwidth of the filter will produce a signal whose two frequency components both mix into the pass band and cannot be filtered. It is not sufficient to simply filter out harmonics greater than or equal to $(4/3) * IF$ because some lower frequencies may mix into the pass band of the post-mixer filter as well. For example, using a 50MHz IF and a 33MHz RF, the second, third and fourth harmonics of the fundamental are 66MHz, 99MHz and 132MHz (Table 2.2). The filter used to condition the mixer output is a four pole Bessel bandpass filter with constant phase delay and sharp sidebands (Figure 2.2). The -3dBc frequencies of the filter are 48.75MHz and 51.25MHz. The -40dBc frequencies are 42.01MHz and 59.01MHz. The desired -20dBc attenuation occurs around 44MHz and 56MHz for a bandwidth of about 6MHz. According to Table 2.2, the 66MHz harmonic

Table 2.2. Harmonic Mixing for RF=33MHz, IF=50MHz, LO^-=IF+RF=83MHz, LO^+=IF-RF=17MHz.

Harmonic Freq.	$LO^- + h*RF$	$LO^- - h*RF$	$LO^+ + h*RF$	$LO^+ - h*RF$
33	116	50	50	16
66	149	17	83	**49**
99	182	16	116	82
132	215	**49**	149	115

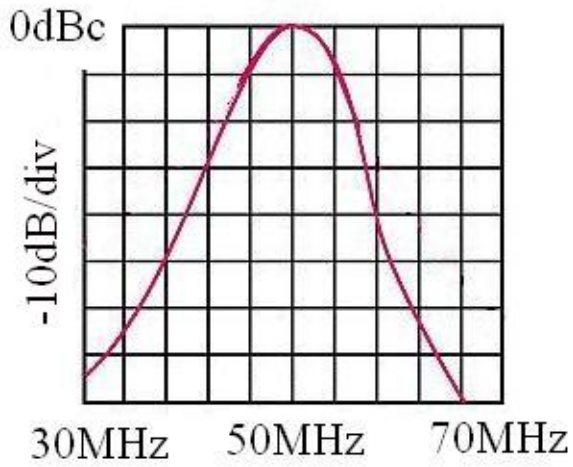

Figure 2.2. Bessel Bandpass Filter Response.

will mix with the sum LO of 17MHz to create 49MHz and 83MHz. If that harmonic is not filtered out before mixing, the 49MHz signal will pass through the filter and be digitized along with the desired 50MHz signal causing phase noise to show up in the reading. The same type of problem arises from choosing the difference LO of 83MHz, however the frequency of the harmonic increases to 132MHz. Adding in the bandwidth of the filter, F_{BW}, changes the equations for the harmonic frequencies that will mix into the pass band to

$$h * RF = \frac{h}{h+1}(2IF \pm F_{BW}) \tag{2.22}$$

for the sum LO and to

$$h * RF = \frac{h}{h-1}(2IF \pm F_{BW}) \tag{2.23}$$

for the difference LO. The lower limit of the sum equation moves to

$$\frac{4IF - 2F_{BW}}{3} \tag{2.24}$$

and the upper limit moves to

$$2IF + F_{BW} \tag{2.25}$$

The lower limit of the difference equation move to

$$2IF - F_{BW} \tag{2.26}$$

and the upper limit moves to

$$4IF + F_{BW} \tag{2.27}$$

The higher minimum problematic harmonic makes it easier to filter the input before mixing when using the difference LO frequency because there is more separation between the RF and the frequencies that need to be filtered out. For instance, for the cyclotron running at between 9MHz and 27MHz, using a 50MHz IF and a LO such that IF=LO-RF, a low pass filter on the input would be require -20dB of attenuation at 94MHz to catch any harmonics that might mix into the passband. This same filter would not work for a system used in the Rare Isotope Accelerator (RIA) where the RF runs at 805MHz[7]. Therefore, the filtering is left up to the user so a suitable filter can be used without limiting the application of the module.

The mixing process is one of the most crucial steps in designing a phase meter module because the ability to design for one IF given a number of RF inputs makes the system much less complicated and more versatile. However, care must be taken in the preparation of the signals because, as has been shown, any inputs that are not sufficiently clear of harmonics and other types of noise can adversely affect the module measurements.

CHAPTER 3

Conditioning the Input Channels

Once the RF input signals have been mixed and conditioned to a common IF frequency, the signal must be matched to the input levels required by the analog to digital converters (ADCs). The closer to full scale these signals are the more accurate the digitization and subsequent measurement. For an input voltage range of +33dBm to -7dBm the output of the mixer stage should be between -1dBm and -41dBm. In order to condition this variable signal to a constant full-scale ADC input signal, a digital attenuator is placed in series with a fixed chain of amplifiers and attenuators.

3.1 Dealing with Variable Input Levels

First, the low level signal is amplified using a gali-51 +18dB amplifier to separate it from the noise floor before it is attenuated again. The signal passes into the digital attenuator section where, depending on the input RF frequency and voltage level, it is variably conditioned to a constant value. 6 bit digital attenuators are used to give an attenuation range from 2.5dB to 31.5dB in 0.5dB steps (Table 3.1). To accommodate the wide range of levels that may be encountered, two digital attenuators are cascaded to provide a minimum attenuation of -5.0dB (-2.5dB insertion loss). Both attenuators are connected to the same control bits,

Table 3.1. Digital Attenuator Control Bits.

Digital Control Bits	Attenuation (dB)
000000	2.5
000001	3.0
101000	22.5
111110	31.0
111111	31.5

so a one bit change on the control lines is equal to a 1dB change in attenuation. The result is that any signal within the specified levels can be conditioned to within 1dBm of the target constant value of -28dBm. The signal is then amplified up to +17dBm to match to near the full-scale value of the ADCs and stay within the tolerances of the gali-51 amplifiers (+18.3dBm 1dB compression point). A chain of static attenuator pads and gali-51 amplifiers (Appendix D, Figure D.3) is used to condition to the desired levels. The attenuator pads are used to increase the stability of the gain stage by decoupling the inputs of cascaded amplifiers. By adding a lossy component between amplifiers, the interaction between them is dampened.

Each amplifier is biased to around 4.2V using +12V and a 120W resistor. A 4.7μH inductor is placed in series with the DC biasing circuit to reduce the RF from the amplifier so that it does not couple to the DC bias network. A 0.1μF capacitor to ground between the resistor and the inductor provides a RF ground to further limit the effects of the amplifier on the bias network[10]. The DC blocking capacitor values were chosen such that their reactance is low enough so as not to attenuate the IF as it passes into the 50 Ω input of the next amplifier.

3.2 Noise and Interference Considerations

Due to the large amount of attenuation and amplification needed to handle the variable inputs, internally generated noise created by the amplifiers, mixers, attenuators and interference from other RF signals on the board could pose a real problem if not handled properly. Interference occurs when other RF signals couple either capacitively from trace to trace or as bleed-through in the case of mixers and filters. In general, thermal noise, also known as Johnson noise, shot noise and flicker noise make up the sources of internally generated noise in a system. Amplifiers and resistances introduce noise as a result of the random thermal motion of electrons following the equation[10, 11],

$$\bar{v}^2 = 4kTR(f)\Delta f \tag{3.1}$$

where k is Boltzman's constant, T is the absolute temperature, Δf is the bandwidth and R(f) is the frequency dependent resistance. The noise factor (F) of a part is defined as the ratio of the signal to noise ratio (SNR) at the input to the signal to noise ratio at the output (Equation 3.2)[10, 11].

$$F = \frac{SNR_{IN}}{SNR_{OUT}} \tag{3.2}$$

Converting the noise factor to dB yields the noise figure (NF). The noise figure of an attenuator is equal to the attenuation and the noise figure of an amplifier is typically given in the specifications sheet. The loss through the mixer is approximately 4.7dB, therefore the noise figure of the input section including the attenuators is 37.7dB. Using the noise figures

of cascaded parts, the noise factor of a section of circuitry can be calculated [10, 11] using,

$$F_0 = F_1 + \frac{F_2 - 1}{G_{A1}} + \frac{F_3 - 1}{G_{A1}G_{A2}} + ... + \frac{F_N - 1}{G_{A1}G_{A2}...G_{A(N-1)}} \tag{3.3}$$

where F_1, F_2, ..., F_N are the noise factors of each stage and G_{A1}, G_{A2}, ... G_{AN} are the gains of each stage converted from dB. From the specifications sheet, the noise factor of the gali-51 amplifiers is 3.5dB and the gain is 63.1. The noise factor of the attenuators in the chain is 3dB and the gain is 0.5. Using the formula above, the noise figure of the amplifier chain is 2.29, which corresponds to a noise factor of 3.6dB. This means the input stage reduces the SNR by 37.7dB and the amplifiers add enough noise to the system to reduce the SNR by another 3.6dB. Taking readings for each channel using a spectrum analyzer (HP Model E4402B) shows that the noise floor at the input to the ADCs is much lower than the signal, although signal coupling is a problem at certain frequencies. This leads to the fair assumption that the signal to noise ratio at the input is much higher than can be read by the spectrum analyzer and that most of the fluctuations in the signals being digitized are a result of interference.

3.3 Interference Analysis

Figures 3.1a, b and c show the fast Fourier transform (FFT) of the signal being sampled on the module input channel 1 at 9, 18 and 27 MHz. Figures 3.2 and 3.3 show the same for the module inputs on channels 2 and 3. Notice the amount of interference on channel 2 is much higher than that of channels 1 or 3. This is mostly due to the physical layout of the channel with respect to the LO traces and the FPGA filter capacitors. For channels 1 and 3,

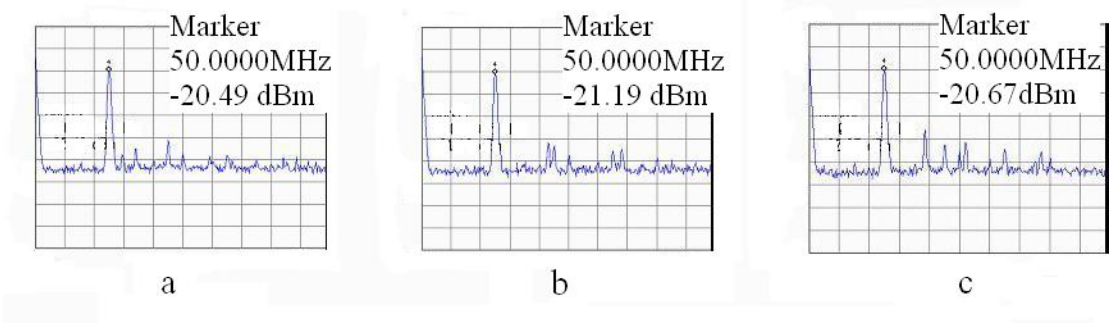

Figure 3.1. Channel 1 FFT Plots at 9MHz(a), 18MHz(b) and 27MHz(c) at the input to the ADC

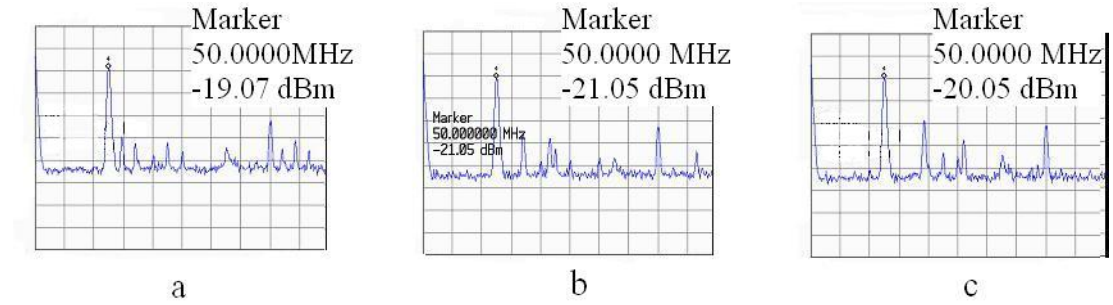

Figure 3.2. Channel 2 FFT Plots at 9MHz(a), 18MHz(b) and 27MHz(c) at the input to the ADC

any interference frequency is at least 30dB down from the fundamental 50MHz signal so it is safe to say that the signals are relatively clean when being sampled and that the noise and interference for those channels is negligible.

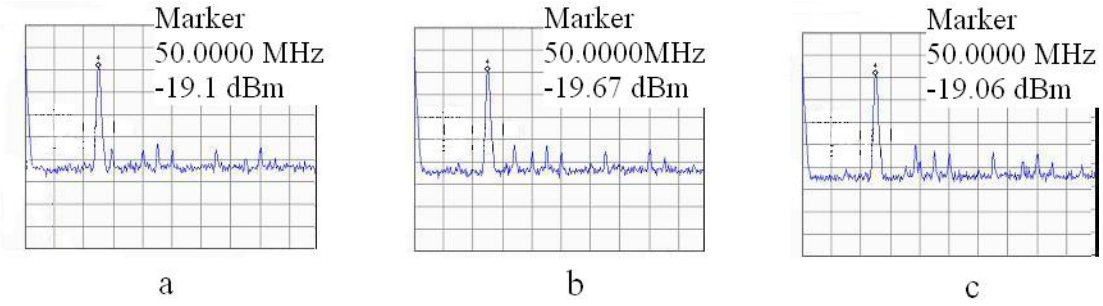

Figure 3.3. Channel 3 FFT Plots at 9MHz(a), 18MHz(b) and 27MHz(c) at the input to the ADC

CHAPTER 4

Phase Lock Loop

For this phase meter to be useful in an environment where multiple modules are to be compared against each other, a stable reference must be used to synchronize the internal clocks in each module. The phase lock loop IC (PLL) compares the phase of the generated clock to the phase of the reference signal, providing a control signal to the voltage controlled crystal oscillator (VCXO) to adjust the frequency of the clock and synchronize them[12] (Appendix D, Figure D.5). Connecting the reference signal to each separate module ensures the modules are locked to each other. In this way the phases calculated by each module are synchronized to the same reference.

4.1 General Operation

The main PLL chip, the ADF 4001, has two RF inputs. The first input is the reference signal generated by a high quality signal generator. The reference signal will is used to lock the phase of each of the clocks generated by the VCXO on the board. The second input port on the PLL chip is the feedback from the output of the VCXO. The PLL compares the two

signals by dividing them in such a way that [13],

$$f_{VCXO} = \frac{R}{N} * f_{reference} \tag{4.1}$$

R is a 14-bit programmable number that can take integer values from 1 to 16,383. N is a 13-bit programmable number that can take integer values between 1 and 8,192.

4.2 Modulating the VCXO

A phase frequency detector (PFD) runs at the frequency of the divided signals and compares them to generate a current output based on the amount of phase variation. The chip can be programmed for either a positive or negative current output, where a positive current output means when the reference phase lags the VCXO phase the current pulse will be positive and vice versa. The ADF 4001 modulates using a bipolar pulse width modulated (PWM) current output which has a maximum frequency equal to that of the PFD. The higher the PFD frequency the faster the PLL can control the VCXO. The modulation limit of the VCXO is 10kHz and the typical design rule is to set the loop bandwidth of the inverting integrator to be $1/3$ of the modulation limit to get good performance [3, 12], leading to Equation 4.2.

$$BW_{loop} = \frac{1}{2\pi RC} = \frac{1}{2\pi 15k\Omega 3nF} = 3.54kHz \tag{4.2}$$

The output of the integrator is used as the control voltage to the VCXO. 3.5kHz is well within the specified modulation bandwidth of the VPLD54TE VCXO, so to remove any high frequency noise or interference from the control voltage line, a low pass filter with a

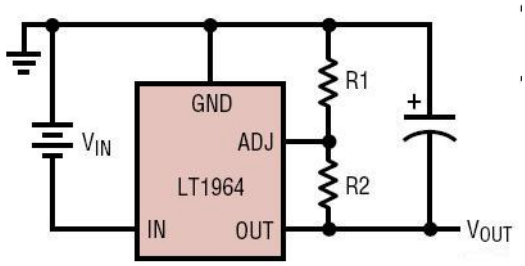

Figure 4.1. Power Supply for PECL Compatibility

3dB corner frequency of around 7kHz limits the oscillations at its input. The slow response of the VCXO to the control voltage keeps the clock signal very stable once it has locked. The VCXO on this board has a center frequency of 80MHz and a pull range of around 8kHz for an input voltage range of 0 to 3.3V.

4.3 Creating the Clock Signals

The differential output is positive emitter coupled logic (PECL) compliant and requires that each pair be terminated into VCC minus 2.0V[14]. The LT1964 produces an output voltage of,

$$V_{out} = -1.22V(1+\frac{R_2}{R_1}) \tag{4.3}$$

Referring to Figure 4.1, setting R2 = 12kΩ and R1 = 18kΩ this power supply maintains the -2V necessary to run the PECL outputs correctly. The VCXO differential outputs are run to a PECL clock divider with both f_{VCXO} and $f_{VCXO}/2$ outputs that converts from the sine wave output of the VCXO to square wave clock signals. Since all of the outputs are generated from the same input, the phase of each clock output is locked. A f_{VCXO} output

and a $f_{VCXO}/2$ output are converted from differential to single ended signals to clock the DAC and the FPGA. The other $f_{VCXO}/2$ output is routed to a 1 to 1 RF transformer whose primary center tap is terminated into -2V and whose secondary center tap is terminated to ground. This signal is routed to each ADC to initiate sampling. Any variation in sampling times will show up as a phase error, therefore the PCB traces must be closely matched from the transformer to each ADC to ensure that all of them sample at the same instant. One of the $f_{VCXO}/2$ lines must also be fed back to the PLL to make sure the phase of the clock signals is locked to the phase of the reference. Using one reference for multiple meters locks the sampling in each module to the same reference and ensures the phase data collected in each module is coherent.

CHAPTER 5

Signal Digitization

The RF input signals have been mixed to a common IF and conditioned to a level at or near the full scale input of the analog to digital converters (ADCs) where they will be digitized and transferred to the FPGA. Digitizing signals can sometimes yield unwanted effects if the sample frequency is not sufficient to recover the entire signal. However, with careful manipulation, it may be possible to recover all of the information that is required.

5.1 Nyquist Zones

Generally speaking, when the frequency content of a signal is not known explicitly, Nyquist criterion states that to recover all of the frequency content within the signal without losing any information you must sample at a minimum of two times the highest frequency that may exist in the signal. Frequencies that lie within the band starting at 0 and going up to one half of the sampling frequency (f_s) are contained within the 1^{st} Nyquist zone. From $f_s/2$ to f_s is the 2^{nd} Nyquist zone, f_s to $3 * f_s/2$ the 3^{rd} Nyquist zone and so on. Setting f_s sets which zone a frequency will be contained in. Sampling a signal creates images of the frequency, f, at,

$$f_{image} = |\pm m f_s \pm f| \tag{5.1}$$

where $m = 1, 2, 3...$

Given a frequency spectrum in the 1^{st} Nyquist zone, the orignal signal may have been contained in an even Nyquist zone, in which case the original frequency spectrum is a mirror image of the 1^{st} Nyquist zone with frequencies equal to,

$$f_n = \frac{(n-1)f_s}{2} - f \tag{5.2}$$

whereas an original signal contained in an odd Nyquist zones will have frequency content equal to,

$$f_n = \frac{(n-1)f_s}{2} + f \tag{5.3}$$

where n is the Nyquist zone in question and f is the frequency content of the signal in the 1^{st} Nyquist zone.

When a signal is sampled using an f_s such that some of the frequency content is outside the 1_{st} Nyquist zone, aliasing occurs. Aliasing is the method by which frequency content contained in higher Nyquist zones folds back as an image into the 1st Nyquist zone. Images of the higher frequencies appear at,

$$f_{image} = |m f_s - f_{high}| \tag{5.4}$$

where m is the integer required to bring f_{image} into the 1^{st} Nyquist zone. This method is called undersampling and is useful in certain applications when the frequency content of the signal is known. For more information, see [7, 15].

5.2 Vector Data Using Nyquist Zone Manipulation

A vector modulation/demodulation technique is applied to map the frequency of interest (IF) to the complex plane to facilitate setting/reading of the phase and amplitude. The In Phase (I) value and the Quadrature (Q) value map to the real and imaginary axis of the complex plane. Using Euler's identity the signal may be cast in the following form,

$$V_{IF}(t) = Re\{|V_p|e^{j(\omega_{IF}t+\phi)} = I + jQ\} \tag{5.5}$$

The magnitude and phase can be determined from the I and Q values of the vector using the equations[5],

$$Magnitude(M) = \sqrt{I^2 + Q^2} \tag{5.6}$$

and

$$Phase(\theta) = \tan^{-1}(\frac{Q}{I}) \tag{5.7}$$

$V_{IF}(t)$ may be written as,

$$V_{IF}(t) = V_p \cos(\omega_{IF}t + \phi) \tag{5.8}$$

Using a sampling frequency $\omega_s = 4\omega IF$ yields a sampling interval that is periodic with a time step of,

$$\Delta t = \frac{2\pi k}{\omega_s} = \frac{2\pi\, k}{4\omega_{IF}} = \frac{k\pi}{2\omega_{IF}} \tag{5.9}$$

with $k = 0,1,2,3,0,1,2,3...$

This gives a rotation of $\frac{\pi}{2}$ or 90° between each sample. $V_{IF}(k)$ can be recast into the

form,

$$V_{IF}(k) = V_p \cos(\phi - \frac{k\pi}{2}) + V_{offset} \tag{5.10}$$

where sequential values of k correspond to sequential sampled values which may have a slowly varying offset of V_{offset} and a coordinate plane rotated by 90° steps such that $\Delta\theta = -\frac{k\pi}{2}$. The following values are further defined,

$$V_{IF}(0) = V_p \cos(\phi) + V_{offset} \equiv I^+ \tag{5.11}$$

$$V_{IF}(1) = V_p \cos(\phi - \frac{\pi}{2}) + V_{offset} = V_p \sin(\phi) + V_{offset} \equiv Q^+ \tag{5.12}$$

$$V_{IF}(2) = V_p \cos(\phi - \pi) + V_{offset} = -V_p \cos(\phi) + V_{offset} \equiv I^- \tag{5.13}$$

$$V_{IF}(3) = V_p \cos(\phi - \frac{3\pi}{2}) + V_{offset} = -V_p \sin(\phi) + V_{offset} \equiv Q^- \tag{5.14}$$

$$\tag{5.15}$$

These samples repeat to form a recurring set of four values that are used by downstream microprocessors to create the I and Q values where[4, 5],

$$I = \frac{I^+ - I^-}{2} = V_p \cos(\phi) \tag{5.16}$$

and

$$Q = \frac{Q^+ - Q^-}{2} = V_p \sin(\phi) \tag{5.17}$$

By taking the subtraction, V_{offset} will be removed leaving only the magnitude of the IF multiplied by either a cosine, for I, or sine, for Q. The digitized channels each have a set of I and Q values which were all taken simultaneously so that they can be used to calculate

the phase between channels.

To gather samples 90° apart requires an ADC that can sample at a rate of $f_s = 4f_{IF}$, which could pose problems for higher IFs since conventional ADCs are limited to around 105 mega samples per second (MSPS). By moving the IF to a different Nyquist zone, undersampling can be used to lower the sampling frequency required while still retaining sequential I, Q, -I and -Q values according to the equation[4, 7],

$$f_s = \frac{4IF}{2n-1} \tag{5.18}$$

where $n = 1, 3, 5...$

By forcing the IF into a higher Nyquist zone (n 1), the same I,Q, -I and -Q values will be sampled but the bandwidth will change according to the equation,

$$BW = \frac{f_s}{4} = \frac{1}{2}Nyquist_{BW} \tag{5.19}$$

The bandwidth refers to the image frequency created based on the IF and the sampling frequency and will always be $\frac{1}{2}$ the minimum Nyquist bandwidth because four samples per cycle are required instead of the minimum two as defined by Nyquist[15].

5.3 ADC Implementation

The phase module uses the ADS5542 ADCs with a maximum sample rate of 80MSPS. According to Equation 5.18, to gather I/Q data when the 50MHz IF is in the 1^{st} Nyquist zone, a sampling rate of 200MHz is required, well beyond the maximum sampling rate of

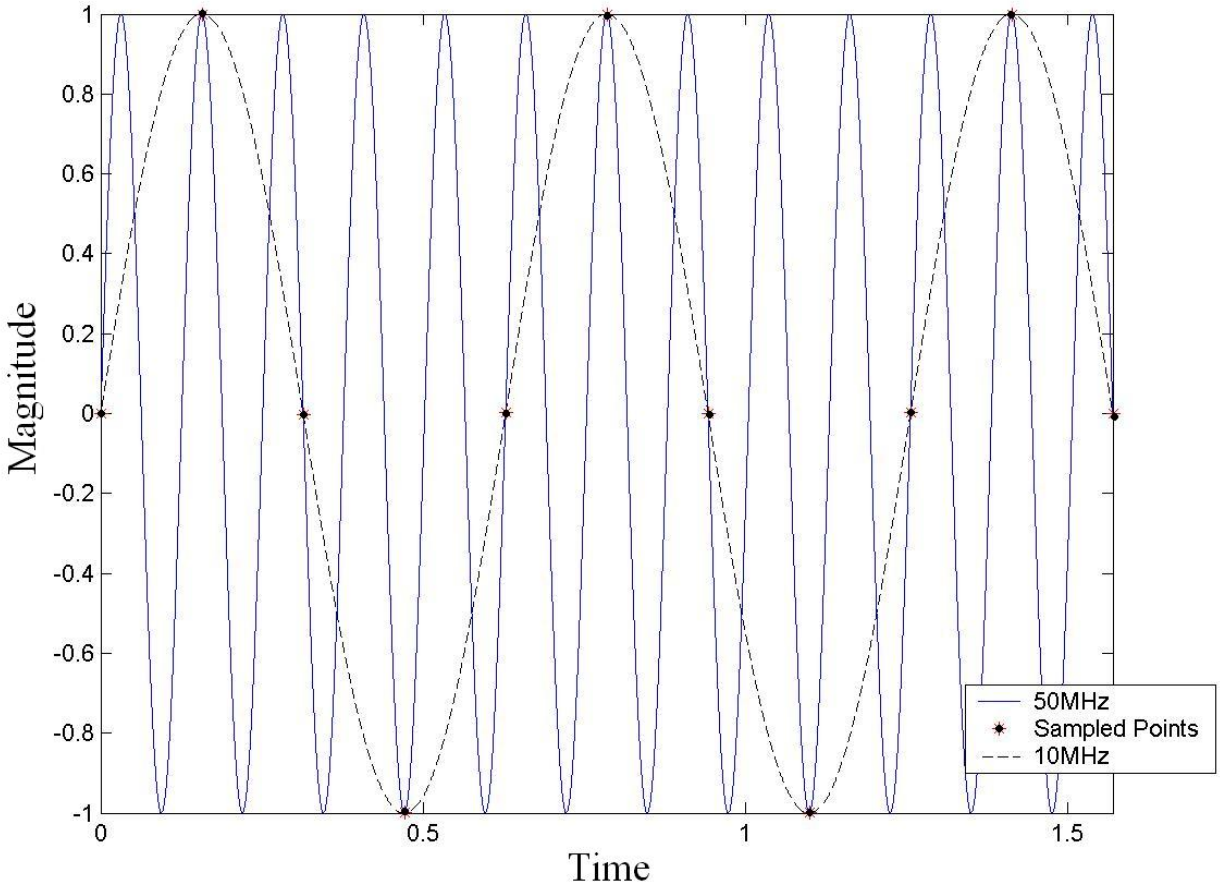

Figure 5.1. Undersampling 50MHz using f_s=40MSPS

conventional technology. Nyquist zone manipulation using an undersampling technique is used to shift the IF into a frequency realm that can be easily handled by readily available technology. Choosing n=3 in Equation 5.18 to put the 50MHz IF into the 3rd Nyquist zone, an f_s of 40MSPS will be necessary. This creates a 10MHz image of the 50MHz IF (Figure 5.1). 40MHz is also four times 10MHz, meaning each sample is 90° delayed from the one before it. These samples will correspond exactly to the samples taken from the 50MHz signal sampled at 200MHz, thereby reducing the required sampling frequency

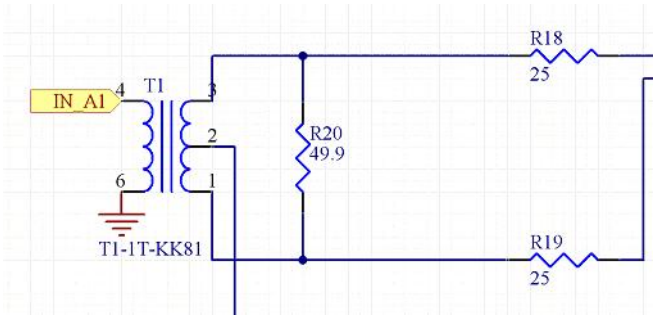

Figure 5.2. RF Transformer Section

without losing any phase information.

5.4 Signal Preparation

To prepare the signals for the ADS5542,the single ended IF signals are passed through a 1:1 RF transformer (Figure 5.2), which converts them to differential signals as per the requirements of the ADCs (Appendix D, Figure D.4). The center tap on the secondary side of the RF transformer is connected to the common mode pin of the ADC to put a DC bias of 1.65V ($\frac{VCC}{2}$) on each signal branch. The common mode voltage generated by the ADC must be very clean to ensure stable signals, therefore a 10Ω resistor in series with the center tap and two filter capacitors, in parallel and connected to ground, are required. A 49.9Ω resister between the positive and negative paths matches the impedance of the transmission line and the 25Ω resisters in series with the inputs to the ADC help to dampen any reflected signals and ringing due to the sample and hold nature of the chip[16].

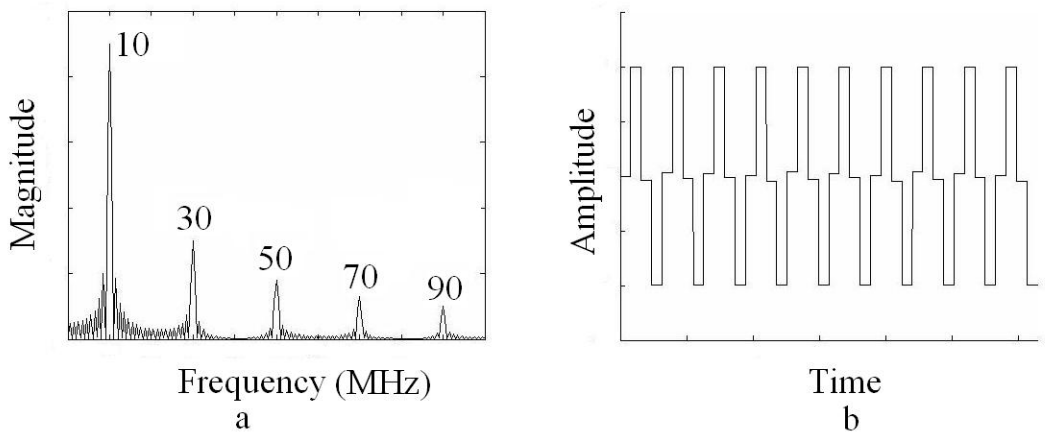

Figure 5.3. The figure on the right (b) shows the DAC I/Q square wave output and the figure on the left (a) is the FFT of the square wave

5.5 High Speed DAC Output

This module has been designed so that it can be extended to replace the existing phase controller and do all of the regulation necessary to run a cavity or cyclotron dee. The output of the module is set by a high speed digital to analog (DAC) converter that is controlled by the FPGA (Appendix D, Figure D.9). Clocking the output at f_s, the FPGA repeatedly sends 14-bit I, Q, -I and -Q values sequentially to the DAC (Figure 5.3). These values are set by the microprocessor and will dynamically update based on the phase that is desired and the phase that is being read off the RF input channels. The $f_s/4$ square wave output that this method creates contains spectral lines at the fundamental $f_s/4$ and at all odd harmonics of $f_s/4$[4]. It is important to keep the sampling frequency as high as possible so that the IF is in the lowest Nyquist zone that can be maintained. The higher the Nyquist zone the higher the odd harmonic required to get back to the IF. Since the harmonic levels fall of as a function of $1/f^2$, the lower the starting frequency the lower the level at the IF[4]. The

DAC output is then filtered through a bandpass filter set at the IF to remove the higher and lower harmonics, leaving a clean IF signal. This IF signal is mixed with the LO frequency to recreate the original RF. The LO+IF is filtered out using a low pass filter with a corner frequency such that the maximum RF may pass without much attenuation but the minimum LO+IF will be filtered out. The required level at the output is +13dBm (1VRMS). An amplifier and attenuator chain using two gali-51 amplifiers with +18dB of gain and multiple attenuators of various sizes are used to condition the signal to the correct level while maintaining stability in the same fashion as before. This feature is meant to be implemented at a later date and is documented to be extend the module for cavity and cyclotron control.

CHAPTER 6

The Field Programmable Gate Array

The field programmable gate array (FPGA) is the data collection hub and is used to read and store the data sampled by the ADCs. Samples are separated into I, Q, -I and -Q values and transferred to the microprocessor for phase and magnitude calculations. The FPGA is also used to set the digital attenuators in the amplifier/attenuator chain to condition the IF and for sending I/Q data to the high speed DAC for RF control (Appendix D, Figure D.6). Pins are made available for connecting a DSP card to expand into the control realm in a future project[5].

6.1 FPGA Connections

The high speed and large number of pins configurable as inputs and outputs makes the FPGA a prime candidate for collecting and routing all of the information to the correct places. Each ADC in this module is connected in parallel, each using 14-bits for data and a 1-bit as an RF over range indicator. A high speed DAC also has a 14-bit data bus and a 1-bit power down control line connected to the FPGA. A 16-bit data bus connects the FPGA to the microprocessor, using 8-bits configured as inputs to the FPGA from the microprocessor and 8-bits configured as outputs from the FPGA to the microprocessor. Two handshaking

Table 6.1. I/Q Determination.

Sample #	Counter Value	I/Q Value
1,5,9,13...	0	I
2,6,10,14...	1	Q
3,7,11,15...	2	-I
4,8,12,16...	3	-Q

bits, one set by the FPGA and one set by the microprocessor, synchronize the data transfer between. A 4-bit command bus controlled by the microprocessor is used to indicate to the FPGA what data is required and how it is to be utilized.

6.2 Collecting I/Q Vector Data

The ADCs are configured to sample the IF on the rising edge of the sampling clock, and the digital information is ready and stable on the data bus at the falling edge of the clock. The clock is used as an input trigger for the FPGA and an event is set to trigger on its falling edge to read the input values from each of the four ADC data buses. As was discussed before, the IF is undersampled to yield digital data that is periodic with a frequency of $f_s/4$. Each input is run through a first order digital band pass filter that has a center frequency of $f_s/4$ corresponding to the difference equation,

$$y[n] = \frac{x[x]}{2} - \frac{y[n-2]}{2} \tag{6.1}$$

to remove any noise that may have been picked up by the ADC during sampling[15]. A 2-bit counter casts the sample as being either I, Q, -I or -Q, naming the first sample I, the second Q, the third -I, the fourth -Q and then repeating as in Table 6.1. Each channel is

latched in parallel, so on the first falling edge of the clock the FPGA reads and stores I_A, I_B, I_C and I_{REF} simultaneously. On the next falling edge of the clock, it reads and stores Q_A, Q_B, Q_C, Q_{REF} and so on. In this manner, the channel to channel phase is conserved.

6.3 Conditioning the Inputs

If the phase of the inputs is not changing, the I/Q values should be constant as well. So, to reduce the effect of interference and digital noise on any of the ADC inputs, a low pass filter is implemented on the raw data that is sent to the microprocessor. To increment(decrement) the value currently held in an I, Q, -I or -Q register by 1-bit, the current input must be greater(less) than the stored value for some specified number of clock cycles. If the value of the input dips below(raises above) the stored value for one clock cycle, the process is restarted.

The maximum number of clock cycles required to increase by 1-bit is specified by the variable 'center' and is referred to as the filter factor. The number of clock cycles required to move by 1-bit can be set anywhere from 1 up to 'center'. The more cycles required to move the stored value, the less fluctuation the phase measurement will have. However, the system response to a real phase change will be slower following the equation,

$$\Delta t(deg/s) = \frac{\frac{1}{4} * f_s cycles/s}{filter\,factor} * \frac{360^\circ}{2^{14} bits} \tag{6.2}$$

For example, with $f_s = 40MHz$ and a filter factor of 1,000 cycles, the result will allow a maximum rotation of 219.7° per second. An 180° shift would take 0.82 seconds to settle to the correct phase. Requiring a high number of cycles may cause problems with accuracy

on a signal that contains a large amount of interference that is not completely random. If the sample value fluctuates around some median number due to random noise, eventually the I, Q, -I and -Q values will be accurate.

6.4 Creating the DAC Output

Two branches of code are run on the falling edge of the clock. The first branch of code creates an RF output by sending I/Q data to the high speed DAC. In the current implementation of the code, I/Q data from one of the four input channels is selected and directly fed through to the DAC. Eventually, the I/Q data will be set by a control processor that calculates the phase desired and compares it to the phase that is being read. The control processor will transfer the four I/Q values to the FPGA and they will be continuously cycled to the DAC until a new phase or magnitude is required.

6.5 Buffering the Inputs

The second and most important branch of code to the phase meter is the double buffering of the I/Q data for transfer to the microprocessor. After every fourth sample is taken, the values stored in the registers for I, Q, -I and -Q for each input are shifted to another set of registers that are read directly by the microprocessor. If the FPGA is in the process of sending the I/Q values to the microprocessor the buffers will not be updated. This allows the FPGA to continue latching data from the ADCs in real time without affecting the data that is being read over multiple cycles by the microprocessor. Once the transfer has completed, the FPGA is able to shift the I/Q data into the buffers again.

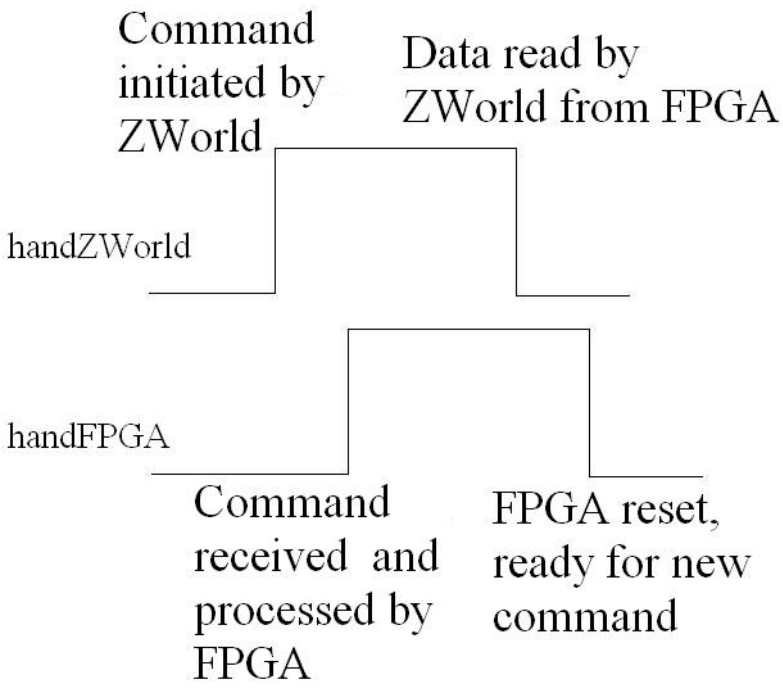

Figure 6.1. Handshaking Timing

6.6 Data Bus Transfers

Transferring the data to the microprocessor requires handshaking, which is also handled on the rising edge of the clock. This ensures that the commands are sent and received by the FPGA at a periodic rate and synchronized to the rest of the operations. The FPGA samples the microprocessor handshaking bit to determine if a command is waiting to be executed on the command bus. When the handshaking bit transitions, it triggers a command bus read and initiates command processing. The FPGA sets its own handshaking bit to relay to the microprocessor as to the status of the command processing. The timing of the handshaking is illustrated in figure 6.1.

On a positive transition of the microprocessor handshaking bit the FPGA reads the command bits and determines the action to take according to Table 6.2. Any data that

Table 6.2. FPGA Commands.

Command	Action
0000	Set digital attenuators
0001	Set ADC channel to DAC
0010	Send an I value to ZWorld
0011	Send a Q value to ZWorld
0100	Send a -I value to ZWorld
0101	Send a -Q value to ZWorld
0110	Read high 8-bits of filter factor
0111	Read low 2-bits of filter factor
1000	unfreeze I, Q, -I, -Q buffer update

Table 6.3. ADC Mode Select.

ADC Mode Value	DAC Output Source
00	Channel 1
01	Channel 2
10	Channel 3
11	Reference Channel

must be read from the microprocessor is latched and any data that needs to be sent to the microprocessor is set up on the data bus. The FPGA responds that it has finished processing the command and waits for the microprocessor to read any data it needs and release the data bus.

Setting the digital attenuators and the ADC mode are simple reads from the data bus. The 6-bit attenuator value requires a single read and the value is shuffled directly to the output pins connected to the chips. The ADC mode selects the input channel that is passed through to the DAC output according to Table 6.3.

The 8-bit data bus requires that the values be broken up and sent in two sections. The bits on the command bus determine whether the I, Q, -I or -Q values are to be read, while the data bus from the microprocessor tells the FPGA the channel to send and whether to

send the high or the low byte for that channel (See Appendix A for a complete description). The FPGA loads the required data onto the data bus to the microprocessor and toggles its handshaking bit high to indicate the command has been processed and the data is available. Once the 8-bits have been read, the FPGA goes back into normal operation and waits for the next command from the microprocessor. However, the I/Q buffer updates do not resume until all of the I/Q values have been read and the command has been issued to begin again.

The final command allows the microprocessor to change the filter factor to change the number of cycles required to increase and decrease the I/Q values stored in the FPGA. It is defined as a 10-bit number and requires two bus transfers to transmit the entire value.

6.7 Conclusion

The high speed and parallel processing of the FPGA makes it a robust solution for routing and storing massive amounts of data in real time. Without these capabilities, the techniques used to make this phase meter work would not be possible. For future exploration, the FPGA could be integrated into more of the control and data processing algorithms the expand on its role in the phase meter.

CHAPTER 7

The Microcomputer

7.1 The ZWorld Microcomputer

The ZWorld Rabbit Core 3200 microcomputer is the CPU of choice for this project. Nearly any microcomputer could be used as long as it has the ability to communicate over Ethernet and has a serial programming interface (SPI). Changing the microcomputer would require changing the connector on the board and rewiring the new connector to the existing peripherals. The ZWorld was used because of the vast amount of code already developed for Ethernet and Telnet communication here at the lab making integration into the cyclotron control system much easier. The code specifically written for this thesis is included in the Appendix B.

The ZWorld handles all of the external communication and configures the chips on the board. It provides the initialization routines and data to get the module up and running. Through the ZWorld telnet interface, the user can set and change all of the configurable options on the board. Most importantly, the ZWorld microcomputer is responsible for reading in the raw data from the FPGA and calculating phase and magnitude information. For a schematic of the ZWorld connections see Appendix D, Figure D.2. Lastly, it manages all bus communication with the FPGA to initiate data transfers.

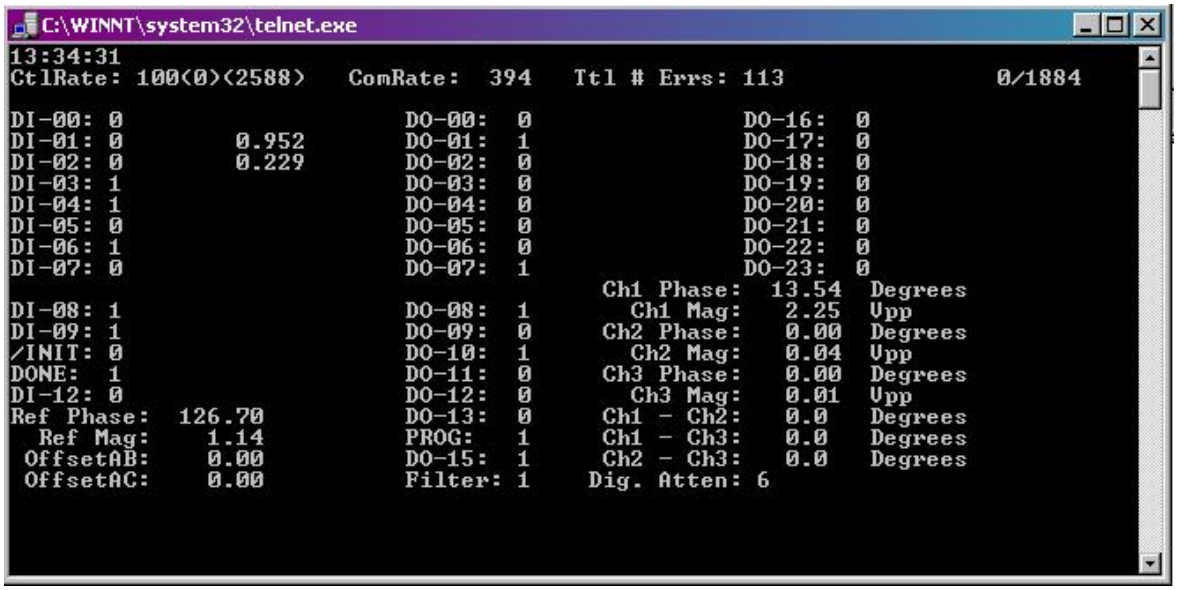

Figure 7.1. Telnet Interface

7.2 Interacting with the ZWorld

There are a number of digital I/O ports on the ZWorld microcomputer that are used to communicate with the various chips on the board. These include four sets of serial transmit and receive ports, 13 digital inputs to the ZWorld and 24 digital outputs from the ZWorld. A complete list of I/O port configurations can be found in Appendix C.

7.2.1 The User Interface

For this thesis, the front end (Figure 7.1) displayed by the ZWorld over the telnet connection contains most of the pertinent information as to the status of the module Table 7.1.

Table 7.1. Telnet Interface Description.

Telnet Label	Description
DI-XX	Displays the value being read on the XX digital input pin
DO-XX	Displays the value being sent to the XX digital output pin
/INIT	Initialization pin on the FPGA, used for configuration timing
DONE	Done pin on the FPGA, indicates the FPGA has been programmed
PROG	Program pin on the FPGA, used to initiate FPGA programming
Filter	Displays the stored filter factor
Dig. Atten	Displays the digital attenuation (dB)
ChX Phase	Displays the phase as $\theta_{ChX} - \theta_{ref}$
ChX Mag	Displays the magnitude of channel X
OffsetAB	Displays the static offset from Channel 1 to Channel 2
OffsetAC	Displays the static offset from Channel 1 to Channel 3
Ref Phase	Displays the phase of the reference
Ref Mag	Displays magnitude of the reference

7.2.2 Serial Programming

The ZWorld is responsible for setting up all of the chips on the phase meter board. The phase lock loop, the FPGA, the fast ADCs, the slow ADC, the slow DAC, the digital attenuators and the EEPROM are all configured using the serial programming interface. The EEPROM and the FPGA are both serially programmed with files stored on the network[17, 18]. The ZWorld can transfer a configuration file from an Ethernet connection to the FPGA or to the EEPROM. It can also write the file from the EEPROM to the FPGA when there is no network connection.

7.2.3 User Commands

The Telnet interface allows the user to input commands to configure everything from the fast ADC's mode to the PLL divide ratios. The FPGA and EEPROM configurations are both initiated by the EPICS control system. The commands in Table 7.2 are implemented

Table 7.2. ZWorld Telnet Command List.

Telnet Command	Description
init #	Initialize the PLL (values 0-7)
config #	Reconfigure the PLL without initializing (values 0-7)
ncount #	set the N-Counter Register for the PLL (values 1-1023)
rcount #	set the Reference Counter Register for the PLL (values 1-1023)
dig # #	set output bit (0-23) to either 1 or 0
setatten #	set the digital attenuators (values 5-45)
setadc #	set the ADS5542 ADC mode (values 0-3)
read #	set the channel that is fed through to the DAC (values 1-4)
filter #	set the filter factor
offsetab #	set the phase offset between Channel 1 and Channel 2
offsetac #	set the phase offset from Channel 1 to Channel 3

Table 7.3. Telnet Interface Description.

M3	M2	M1	Output
0	0	0	Three-State Output
0	0	1	Digital Lock Detect
0	1	0	N-Divider Output
0	1	1	AVDD
1	0	0	R-Divider Output
1	0	1	N-Channel Open Drain Lock Detect
1	1	0	Serial Data Output
1	1	1	DGND

in the current incarnation of the ZWorld program. The PLL can be configured for diagnostics so that pin 14 on the ADF 4001 outputs various internal signals that would not normally be available for probing (Figure 7.3). The bits in the initialization register and the configuration register are the same. The initialization register must be loaded first after power is applied to the chip to reset the inner workings. For changes after the chip has been initialized, the configuration register should be modified. Only the diagnostic values M3, M2 and M1 are configurable by the user. The rest of the bits are set by the ZWorld program to put the ADF 4001 in normal operation mode with a current output of 5mA

and a phase frequency detector timeout of three cycles. The fastlock is turned off and the phase frequency detector polarity is set to positive so that when the VCXO phase leads the reference phase the charge pump output is positive and when the VCXO phase lags the reference phase the charge pump output is negative. These settings are applied when both config and init commands are run. The rcount command sets the VCXO frequency divide ratio to the the user specified value, sets the antibacklash pulse width to 1.3ns and sets the lock detect precision to 3 cycles. The ncount command sets the reference frequency divide to the value specified by the user. For a complete description of these parameters see the [13].

An interrupt routine, running once every 10ms, handles processing commands from the user interface (UI) as well as data transfer to and from the FPGA and phase calculations. On its first run, the interrupt routine sets up the PLL to run with MUXOUT configured as digital lock detect, sets the VCXO divide ratio to 4 and sets the reference frequency divide ratio to 1. This assumes a 10MHz reference and a 40MHz sampling frequency. The IF sampling ADCs are configured to run in normal operation mode.

The ZWorld must initiate all bus transfers by toggling a handshaking bit to trigger the FPGA. Before the handshaking bit is toggled high, the command to be executed and any data pertaining to that command must be written to the outputs. Next, the handshaking bit is toggled. Setting the command and data bits first allows them to settle before the FPGA reads them. The ZWorld waits for a response from the FPGA that the command has been read and processed. Once that response has been received, the ZWorld reads any data the FPGA has set and releases the bus.

The main task of the interrupt routine is to retrieve the I/Q data from the FPGA and

calculate the phase. The I/Q values are stored as 16-bit numbers and require two 8-bit bus transfers for retrieval. The process of reading the entire set of I/Q data and calculating the phase requires 67 passes through the interrupt. The interrupt is triggered every 10ms so the phases are updated at a rate of,

$$\tau_{update} = 67 cycles * 10^{-3} s/cycle = 0.67 s \tag{7.1}$$

The phase of each channel is calculated using,

$$\theta_i = \tan^{-1}(\frac{Q_i}{I_i}) \tag{7.2}$$

The phase between each channel is determined using,

$$\theta_{ch1} = \theta_1 - \theta_{ref} \tag{7.3}$$

$$\theta_{ch2} = \theta_2 - \theta_{ref} \tag{7.4}$$

$$\theta_{ch3} = \theta_3 - \theta_{ref} \tag{7.5}$$

which leads to channel to channel phase to be taken as

$$\theta_{ch1-ch2} = \theta_{ch1} - \theta_{ch2} = \theta_1 - \theta_2 \tag{7.6}$$

$$\theta_{ch1-ch3} = \theta_{ch1} - \theta_{ch3} = \theta_1 - \theta_3 \tag{7.7}$$

$$\theta_{ch2-ch3} = \theta_{ch2} - \theta_{ch3} = \theta_2 - \theta_3 \tag{7.8}$$

The magnitude is calculated using,

$$Mag_{ch1} = \sqrt{I_1^2 + Q_1^2} \tag{7.9}$$

$$Mag_{ch2} = \sqrt{I_2^2 + Q_2^2} \tag{7.10}$$

$$Mag_{ch3} = \sqrt{I_3^2 + Q_3^2} \tag{7.11}$$

$$Mag_{ref} = \sqrt{I_{ref}^2 + Q_{ref}^2} \tag{7.12}$$

The ZWorld microcomputer does an excellent job of handling the module configuration, processing the data and acting as the front end for the user interface. The low phase update rate requirements of the phase meter make the ZWorld an ideal chip for calculating phase, however, when control is implemented, the task of calculating phase will rest on a much faster DSP.

CHAPTER 8

Signals and Interlocks

A final section of interlocks, monitoring systems and supplemental hardware allow this module to be practically useful to the existing cyclotron system. These systems monitor the status of the external RF control system to control the functionality of the module and monitor the functionality of the module to relay the module status to the external system.

8.1 External Signals and Status Indicators

A set of LEDs provides vital information at a glance regarding the operation of the module. The microprocessor controls the status of these three LEDs (Appendix D, Figure D.8). Located on the back of the module, the activity LED lights when the CPU is busy being updated. On the front of the module, the RF On LED relays the status of the RF in the module and the Ready LED lights when the configuration process is done and the module is in working order.

Two signals from external systems are buffered and connected to the FPGA as interlocks. An RF enable signal connects to the front of the module and is generated by the RF control system to turn the RF on and off. A module reset signal is also generated by the cyclotron control systems. Both signals tie directly to the FPGA and are used as inputs,

responding quickly to any change on either one.

The FPGA creates a high-speed fault signal, which is used to indicate a problem with the RF anywhere inside the phase module. The fault can be tied to any type of internal workings including a loss of RF or an overload signal from an ADC that has an RF level that is out of range, to name a few. This signal will, again, be more useful once the cavity control has been instantiated.

8.2 Housekeeping Circuits

In addition to status LEDs and interlocks, there are a couple of other housekeeping circuits that monitor the inner workings of the module and report the information back to the microprocessor (Appendix D, Figure D.10). An ADS7825 16-bit 4-channel serial ADC is used to read slowly varying voltages. The full-scale input of the ADC is 10V with a conversion time of 20μs and an acquisition time of 5μs. The maximum sample frequency of the ADC is 40kSPS, which is plenty fast to sample the aforementioned signals since they are expected to be slowly changing and their values need only be monitored periodically. Two of the four channels are connected to signals on the board, with the other two left as spares for future use. An analog temperature sensor monitors the temperature of the board and outputs a voltage of 250mV at 25C with a slope of 10mV/C. An amplifier is used to condition the voltage output of the temperature sensor to utilize more of the full-scale input of the ADC to reduce digitization error and give a more accurate reading. The control voltage for the VCXO is also conditioned and sampled so the lock status of the PLL can be monitored. The conditioning is done by four op amp circuits, which are set up as non-inverting

amplifiers and follow the equation,

$$V_{out} = V_{in}(1 + \frac{R_2}{R_1}) \tag{8.1}$$

where R_1 and R_2 are chosen to try and use as much of the full scale of the ADC as possible.

The Spartan II XC2S150 FPGA used on this board requires a configuration file that is 130,012 bytes, which is larger than the available flash memory on the microprocessor. The FPGA configuration memory is volatile meaning that it loses the information when power is removed from the chip. An AT25P1024 1Mbit serial EEPROM with 131,072 bytes of available storage space is used to store the configuration file when the power is off[19]. The EEPROM communicates with the microprocessor using the Serial Programming Interface (SPI). On power up, the microprocessor can pull the configuration data from the EEPROM and send it to the FPGA or transfer it over Ethernet. This allows the module to work even if it is not connected to a network from which it can download the latest FPGA configuration data.

The last housekeeping circuit is the 4-output 12-bit serial DAC model MAX5742. The four outputs are individually configurable and are intended to output a voltage proportional to the different phase readings taken by the module. To be compatible with the current system running the cyclotron, the DAC voltage outputs must be bipolar. Each output is connected to an op amp according to the schematic shown in Figure 8.1. This gives a swing range of $\pm$Vref, which is $\pm$2.5V for this module. Each instruction must be sent to the DAC serially on the SPI bus as a 16-bit string of values where the least significant 12 bits correspond to the output voltage (Table 8.1). The most significant four bits of the

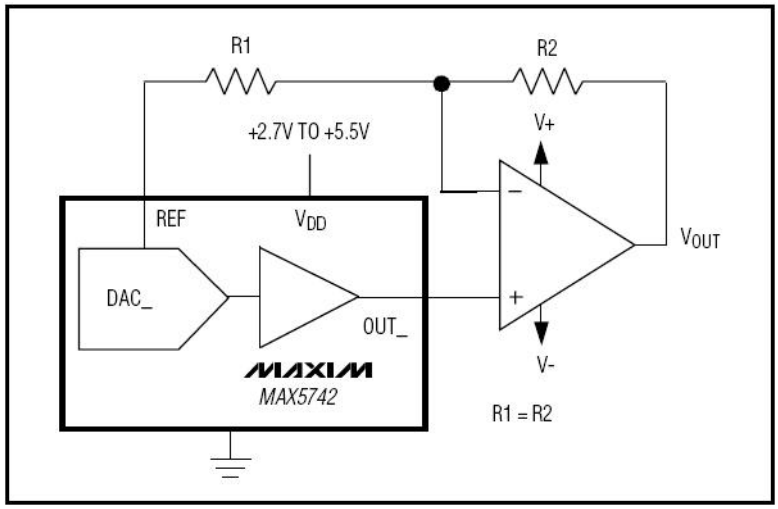

Figure 8.1. Bipolar DAC Configuration

Table 8.1. DAC Binary Output Chart.

DAC Contents	Analog Output
1111 1111 1111	$+V_{ref}(\frac{2047}{2048})$
1000 0000 0001	$+V_{ref}(\frac{1}{2048})$
1000 0000 0000	0
0111 1111 1111	$-V_{ref}(\frac{1}{2048})$
0000 0000 0001	$-V_{ref}(\frac{2047}{2048})$
0000 0000 0000	$-V_{ref}$

instruction are the control bits and tell the DAC which outputs are going to be changed and how to change them.

The DAC outputs are accessible on the connector on the back of the module and can be connected to the existing phase control modules to display the phase for each station in the cyclotron.

The supplemental hardware and interlock system is designed to monitor the status of the overall system and is mostly intended to facilitate implementing the next phase of the project by making it easier to add control elements to the phase meter.

CHAPTER 9

Phase Meter Performance

The purpose for developing this module was to replace the obsolete analog vector voltmeters (Model HP 8508A) currently being used in the cyclotron. The specifications for the existing voltmeter claim an absolute accuracy of $\pm 1^{\circ}$ in the frequency range of 1MHz to 100MHz[20]. Two test setups were used to determine the accuracy of the newly constructed phase meter.

9.1 Determining Module Channel Offsets

In the first test setup (Figure 9.1), a single signal generator (Rohde&Schwarz Model 1090.3000.13) is split using a Janel Laboratories 2-50MHz four-way splitter (Model PD7905) and run into the three RF input channels on the phase meter. Another signal generator (PTS Model 250) is connected to the LO port. The vector voltmeter (HP8508A) is connected in parallel with the module across channels 1 and 3 to take reference phase information. Using the phase control on the Rohde&Schwarz signal generator, the phase of the output RF is rotated with respect to the 10MHz reference signal from 0° to 360° in 10° steps. The channel 1 to channel 3 phase, θ_{VM}, is measured by the vector voltmeter and compared to the channel 1 to channel 3 phase as measured by the module, θ_{module} . The

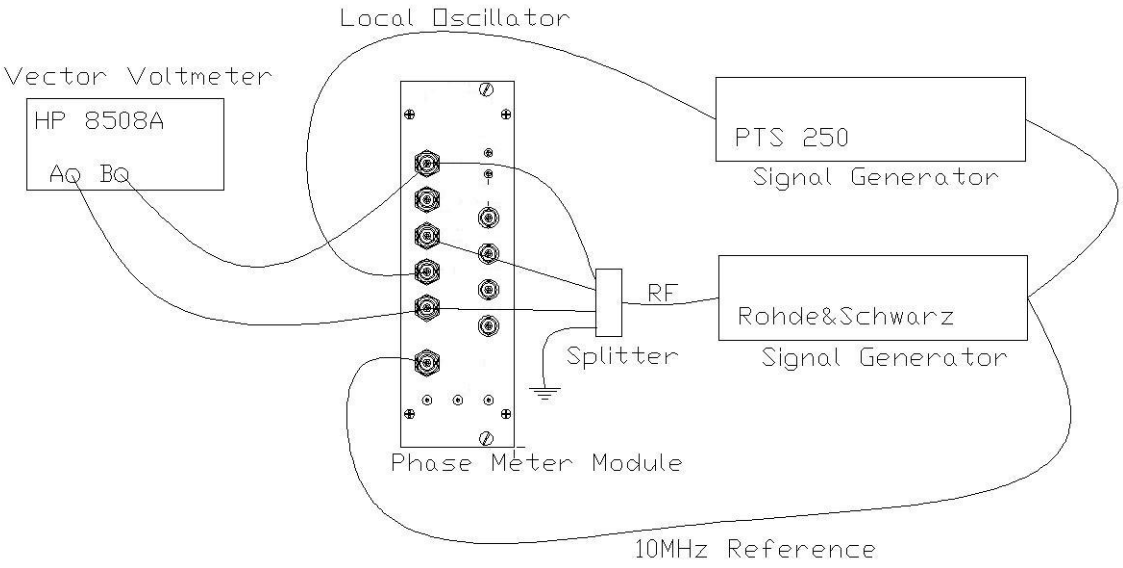

Figure 9.1. Test Setup 1

static offset,θ_{OS} , incurred by the filtering is calculated as,

$$\theta_{OS} = \theta_{VM} - \theta_{module} \tag{9.1}$$

for each reading. The average offset over all of the readings is calculated and subtracted from the module readings. In general the average offset was between 14° and 17°, depending on the frequency of the input. The adjusted module measurements are subtracted from the vector voltmeter measurements and plotted against the channel 1 phase reading on the module (Figure 9.2).

The process is repeated for 9, 15, 21, 25 and 27MHz input signals. The results show that, minus a fixed offset, the module readings deviate less than $\pm 0.6°$ from the vector voltmeter, well within the specified limits of the absolute accuracy. The largest deviations occurred when channel 1 read 0°, 180° and 270°. This could be due to errors in the digital

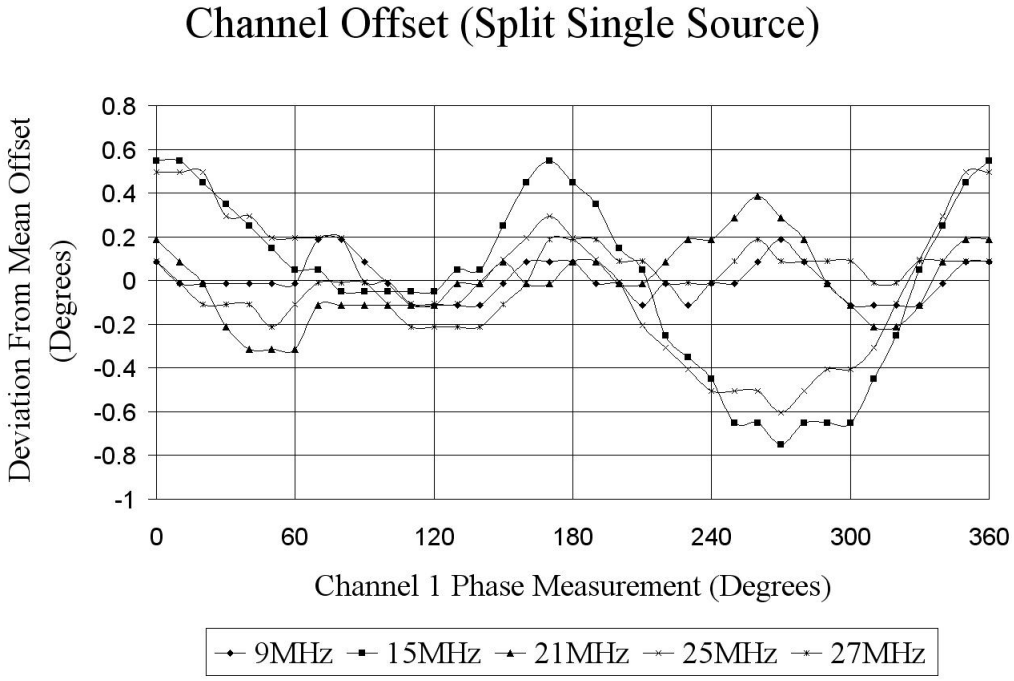

Figure 9.2. Measured Channel-to-Channel Offset

calculation as either I or Q become very close to zero and the argument of the inverse tangent goes to either 0 or ∞. Since the channel 1 and channel 3 readings are offset by approximately 15°, half of the digitization error could be attributed to each channel and the large offset could be a result of the errors adding. This would suggest that 90° should also be a problematic area, although the error appears to cancel itself out instead of adding at 90°. However, since the errors are entirely within the specified absolute accuracy of the vector voltmeter, there is no way to discern which module is giving the most accurate phase reading. These results suggest that the phase meter module is at least as good as the vector voltmeter.

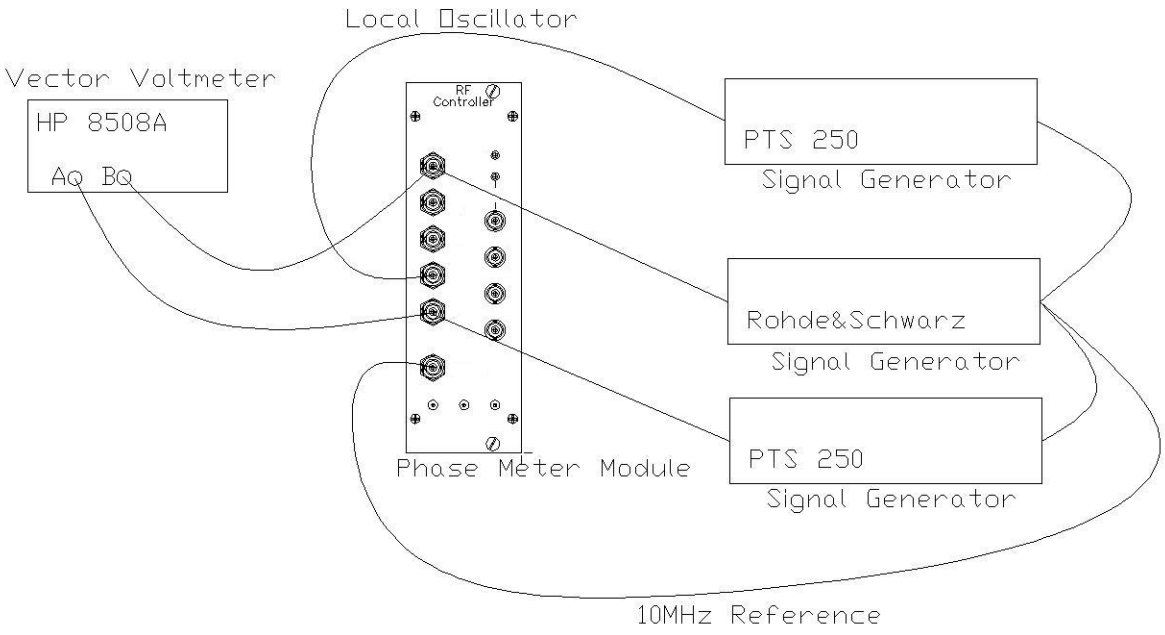

Figure 9.3. Test Setup 2

9.2 Determining Phase Accuarcy

Test setup 2 (Figure 9.3) adds another signal generator (PTS Model 250) connected to the channel 3 RF input of the module and forgoes the splitter in favor of connecting the Rohde&Scwarz signal generator directly to the channel 1 RF input. The vector voltmeter is connected in parallel with the module to measure the phase between channels 1 and 3. The signal generator connected to channel 1 is rotated through phases from 0° to 360° using 10° steps, but this time the phase of the second signal generator is held constant. The phase rotation of the signal generator is only accurate to approximately ±0.6°, so a single step can be between 9.4° and 10.6°. The vector voltmeter measurements of channel 1 to channel 3 are taken as the baseline readings and are recorded and compared to the channel 1 to channel 3 measurements displayed by the module. The average offset of the module

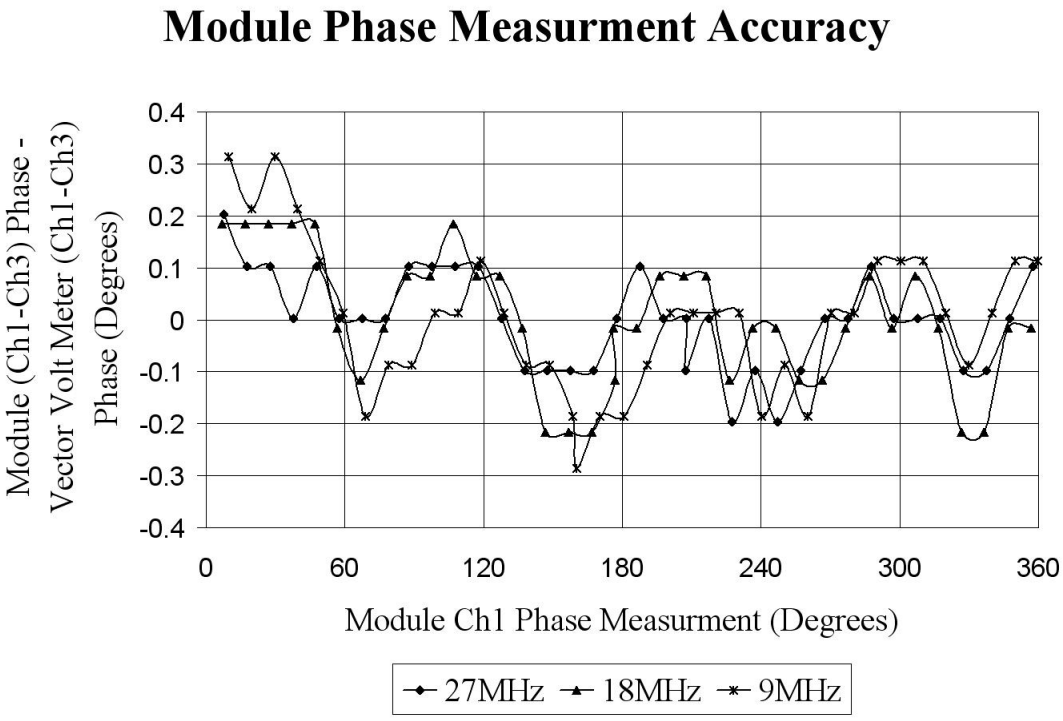

Figure 9.4. Module Phase Accuracy

reading to the vector voltmeter is computed and subtracted from the module reading. The adjusted reading is subtracted from the voltmeter reading and plotted to determine the phase accuracy (Figure 9.4). As the results show, for input frequencies of 9, 18 and 27MHz, the maximum deviation from the vector voltmeter reading is $\pm 0.3^{\circ}$. Once again, this is well within the $\pm 1^{\circ}$ accuracy of the voltmeter and the source of the error cannot be ascertained.

9.3 Calculation Accuracy Dependence on Amplitude

The previous tests were run with the ADCs sampling waveforms at full scale, which is the ideal situation when dealing with digitization errors. To determine the susceptibility of the module's phase and amplitude measurements to channel input amplitude variations, the

test setup is left as it was in the previous test. The vector voltmeter phase is recorded and compared to the module phase measurement as the channel 1 input amplitude is stepped from +13dBm to -10dBm using -1dB steps. The channel 3 amplitude is set to +13dBm and held constant. The digital attenuators are set to provide the ADCs with a full-scale signal given +13dBm on the input. The specifications for the vector voltmeter require at least -7dBm to guarantee the stated accuracy. The test is again run at 9, 18 and 27MHz. Plotting the data vs. signal input amplitude (Figure 9.5) shows that to maintain an accuracy of $\pm 0.5^{\circ}$ requires the input amplitude be within 5dB of full-scale. To maintain an accuracy of $\pm 1^{\circ}$, which is the absolute accuracy of the vector voltmeter, the amplitude can be as low as 13dB down from full scale.

The amplitude is determined from the same samples as the phase, therefore a loss in accuracy of the signal amplitude measurements is expected as well. The amplitude calculated by the module is recorded as a function of the channel 1 input amplitude. The calculated change in amplitude is compared to the actual amplitude change of the signal generator and plotted (Figure 9.6). The amplitude calculations were less susceptible to the change in input than the phase measurements, maintaining an error of around +/-0.1dBm from an input of +13dBm down to near +3dBm.

Taking $\pm 0.5^{\circ}$ and ± 0.2dBm to be accurate measurements in both phase and amplitude, an input can be up to 5dB down from full scale on the ADCs and still be considered correct. The amplitude dependence tests were run at a fixed digital attenuator setting, however the digital attenuators can be always be changed to allow the internal amplification chain to match the signal to the full scale of the ADCs as long as that signal is within the specifications of the module. This will change the range of inputs that will maintain an

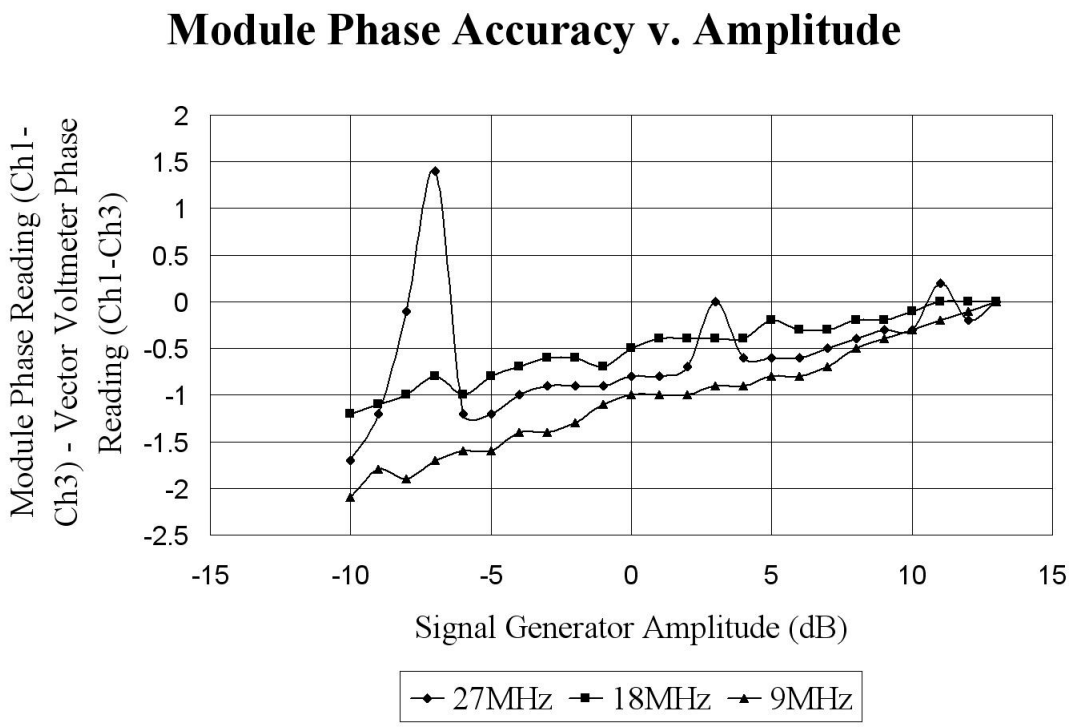

Figure 9.5. Phase Accuracy Amplitude Dependence

acceptable level of accuracy. For instance, to run at 0dBm, the digital attenuators can be set so that 0dBm on the input will still be full-scale at the ADCs and the range for accurate measurements will shift to 0dBm to -5dBm.

9.4 Performance Analysis

The results of these experiments show that the digital phase meter does meet the required specifications set forth by the obsolete analog vector voltmeters and could be a viable alternative for use in any RF system and specifically for the cyclotron. The prototype board used to generate this data still has some interference issues that are known and are to be addressed in future builds. It is worth noting that channel 2 had a very high amount of noise

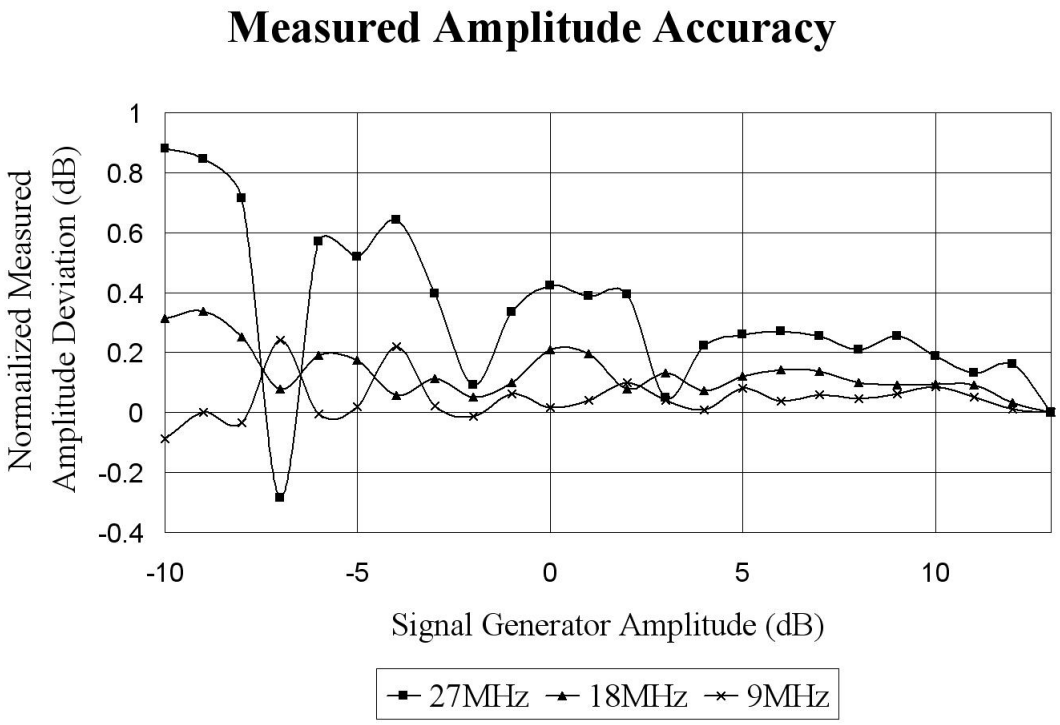

Figure 9.6. Calculated Amplitude Accuracy

due to its close proximity to the FPGA bypass capacitors and the LO channel. Because of the excess interference, measurements were only taken on channels 1 and 3. Steps were taken to isolate the RF channels in the latest board design; unfortunately it was not available for testing in time. Even given the interference issues present, this module could be implemented without further modification as a high accuracy phase meter.

Table 9.1. Phase Meter Specifications

Voltage Input Range (dBm)	-7 to +33
Frequency Input Range (MHz)	9 to 31
Phase Accuracy (Degrees)	±0.4 (+13dBm to +10dBm)
(+13dBm Input Full-Scale)	±.7 (+13dBm to +8dBm)
	±1.2 (+13dBm to 0dBm)
Phase Resolution (Degrees)	0.088
Amplitude Accuracy (dBm)	±0.1 (+13dBm to +3dBm)
(+13dBm Input Full Scale)	±0.2 (+13dBm to -3dBm)
	±0.5 (+13dBm to -10dBm)
Amplitude Resolution (mV)	0.5

*Channel crosstalk was calculated by connecting one input and terminating the rest. The level of the signal was measured at that input (dBm) and the signal level at the input of the ADC (dBm) for other inputs was subtracted from that level to determine the interference.

Table 9.2. Channel to Channel Cross-Talk

CH1 +13dBm Input		9MHz	18MHz	27MHz
	CH1-CH2	80dB	80dB	80dB
	CH1-CH3	80dB	80dB	80dB
	CH1-Ref	80dB	80dB	80dB
CH2 +13dBm Input		9MHz	18MHz	27MHz
	CH2-CH1	80dB	80dB	80dB
	CH2-CH3	80dB	80dB	80dB
	CH2-Ref	80dB	80dB	80dB
CH3 +13dBm Input		9MHz	18MHz	27MHz
	CH3-CH1	80dB	80dB	80dB
	CH3-CH2	80dB	80dB	80dB
	CH3-Ref	80dB	80dB	80dB
LO +7.5dBm Input		59MHz	68MHz	77MHz
	LO-CH1	40dB	47dB	37dB
	LO-CH2	30dB	32dB	25dB
	LO-CH3	40dB	53dB	32dB
	LO-Ref	44dB	46dB	50dB

APPENDICES

APPENDIX A

FPGA Code in Verilog

```
module Phase(clk40,clk80,DigIn,DigOut,Attenuators,handZWorld,handFPGA,DAC,
        ADC_A,ADC_B,ADC_C,ADC_D,command);
/*
  List of inputs:

   Input:          # of bits       Purpose

   clk40             1            used to sync to events triggered by the 40MHz clock generated by the PLL on the PCB, for the
                                  purposes of this code this allows the FPGA to grab the samples from the ADCs which sample at 40MHz

   clk80             1            used to sync to events triggered by the 80MHz clock generated by the PLL on the PCB, for the
                                  purposes of this code this allows the FPGA to send data points to the DAC which samples at 80MHz

   DigIn             8            used as the data bus to receive information from the ZWorld Microcontroller in 8bit chunks. Works in
                                  conjuntion with DigOut, handZWorld and handFPGA to transmit data back and forth between FPGA and CPU

   handZWorld        1            used to signal that a command is ready from the ZWorld Microcontroller and can be read off of the
                                  data bus.    When handZWorld goes high a command is ready to be read and the appropriate data is available
                                  at DigIn.  When handZWorld is low the data on the bus is not guaranteed valid, also signifies the ZWorld
                                  has read the data from the FPGA on DigOut.  Works in conjuction with DigIn, DigOut and handFPGA to
                                  transmit data back and forth between FPGA and CPU

   ADC_A,B,C,D   14               14-bit data input from the four on board ADCs sampled by the FPGA on the falling edge of clk40. Data
                                  format is straight binary from 14'b0 (most negative) to 14'b16383 (most positive)

   command           4            used by the FPGA to receive commands from the ZWorld Microcontroller (see below for details)
*/
  input clk40;
  input clk80;
  input [7:0] DigIn;
  input handZWorld;
  input [13:0] ADC_A, ADC_B, ADC_C, ADC_D;
  input [3:0] command;

/*
```

List of outputs:

Output:	# of bits	Purpose
DigOut	8	used as the data bus to send information to the ZWorld Microcontroller in 8bit chunks. Works in conjunction with DigIn, handZWorld and handFPGA to transmit data back and forth between FPGA and CPU
Attenuators	6	used to set the digital attenuator value from 5dB (6b'000000) to 68dB(6'b111111) in 1dB steps and assuming 5dB insertion loss
handFPGA	1	used to signal that the command from the ZWorld that was initiated by handZWorld has been read and processed. When handFPGA goes high the previous command has been read and completed and the data on the data bus is valid and may be read back by the ZWorld. When handFPGA is low the FPGA is either not working on a command if one was not initiated by handZWorld or the command has not been completed. Works in conjunction with DigIn, DigOut and handFPGA to transmit data back and forth between FPGA and CPU
DAC	14	14-bit data output to the high speed DAC on the PCB. Data is written on the rising edge of clk40 and considered valid for the falling edge of clk40. Data is straight binary 14'b0 (most negative) to 14'b16383 (most positive)

*/

```
output [7:0] DigOut;
output [7:0] Attenuators;
output handFPGA;
output [13:0] DAC;

reg [1:0] counter;  //used to cycle through I(n Phase) and Q(uadrature) samples, placing them in the correct register
                    //00 -> I, 01 -> Q, 10 -> -I, 11 -> -Q then repeat
reg handZWorldReg;  //register reads and stores the handZWorld input value on every positive edge of clk40.
                    //used to trigger all data bus reads and writes
reg freeze;         //used to freeze the I/Q values to be read by the ZWorld so that none are updated in the middle of a read

/*
  icount*,qcount*,imcount* and qmcount* are used to keep track of the number of times in a row a given I,Q,-I or -Q sample is greater
  than or less than the previously stored value (I[n] compared to I[n-1], Q[n] compared to Q[n-1] and so on).  This is individually
  tracked for each I,Q,-I and -Q value for each ADC input.
```

```
*/
reg [10:0] icountA, qcountA, imcountA, qmcountA, icountB, qcountB, imcountB, qmcountB, icountC, qcountC, imcountC, qmcountC;
reg [10:0] icountRef, qcountRef, imcountRef, qmcountRef;

reg [13:0] IVal_A, IVal_B, IVal_C, IValRef;          //accumulated I value for each of four channels (A,B,C and Ref), updated on negedge clk40
reg [13:0] QVal_A, QVal_B, QVal_C, QValRef;      //accumulated Q value for each of four channels (A,B,C and Ref), updated on negedge clk40
reg [13:0] ImVal_A, ImVal_B, ImVal_C, ImValRef;          //accumulated -I value for each of four channels (A,B,C and Ref), updated on negedge clk40
reg [13:0] QmVal_A, QmVal_B, QmVal_C, QmValRef;      //accumulated -Q value for each of four channels (A,B,C and Ref), updated on negedge clk40

/*
  Each IValOut*,QValOut*,ImValOut* and QmValOut* is updated simultaneously on the posedge of clk40 when counter is equal to 2'b11 to ensure all of
  the values correspond to the same cycle and to ensure that each channel's IQ information corresponds to the others.  These values are the ones to
   be read by the ZWorld and are only updated if they are not being read.
*/
reg [15:0] IValOut_A, IValOut_B, IValOut_C, IValRefOut;    //I value read by the ZWorld for each of four channels (A,B,C and Ref)
reg [15:0] QValOut_A, QValOut_B, QValOut_C, QValRefOut;          //Q value read by the ZWorld for each of four channels (A,B,C and Ref)
reg [15:0] ImValOut_A, ImValOut_B, ImValOut_C, ImValRefOut; //Im value read by the ZWorld for each of four channels (A,B,C and Ref)
reg [15:0] QmValOut_A, QmValOut_B, QmValOut_C, QmValRefOut; //Qm value read by the ZWorld for each of four channels (A,B,C and Ref)

/*
  ADC input history buffers for each channel (A,B,C and Ref).  Values are latched on the negitive edge of clk40. x*4 is delayed one cycle
  from x*3 which is delayed one cycle from x*2 which is delayed one cycle from x*1 which latches the current input. Gives x[n], x[n-1],
        x[n-2] and x[n-3]
*/
reg [13:0] xA1, xA2, xA3, xA4;
reg [13:0] xB1, xB2, xB3, xB4;
reg [13:0] xC1, xC2, xC3, xC4;
reg [13:0] xRef1, xRef2, xRef3, xRef4;

/*
  Output buffers for each channel (A,B,C and Ref). Values are latched on the negitive edge of clk40.  y*4 is delayed one cycle from y*3
  which is delayed one cycle from y*3 which is delayed one cycle from y*2 which is delayed one output from y*1 which is delayed one cycle
  from y*Out which is the current output.  Gives y[n], y[n-1], y[n-2], y[n-3] and y[n-4
*/
reg [13:0] yA1, yA2, yA3, yA4;
reg [13:0] yB1, yB2, yB3, yB4;
```

```
reg [13:0] yC1, yC2, yC3, yC4;
reg [13:0] yRef1, yRef2, yRef3, yRef4;
reg [15:0] yAOut, yBOut, yCOut, yRefOut;

reg [7:0] attenOut; //register holds the value for the digital attenuators. updated by the command bus triggered by the command 4'b0000
reg [1:0] ADC_Select; //register holds the value for the ADC mode. updated by the command bus triggered by the command 4'b0001
reg [7:0] DigitalOut; //register holds the DigOut value.  updated by the command bus triggered by commands 4'b0010 through 4'b0101
reg [13:0] ADC_Out; //register hold the value destined for DAC. updated on each positive edge of clk40
reg [10:0] high_reg, low_reg; //registers hold the high and low values for the low pass filter (10'd20480 to 10'd0)
reg [7:0] filter_high_byte; //temporary register holds the high byte of the filter value because of a two part load for a 10 bit number
reg command_received; //register hold the value to be transfered to command_received_reg. updated by the command bus to signify completed command
reg command_received_reg; //clocked register holds the value destined for handFPGA. updated on positive edge of clk40 to gaurantee timing
                          //and signify completed command

//Wires used to connect input history buffers together for each channel (A,B,C and Ref)
wire [13:0] xA1w, xA2w, xA3w, xA4w;
wire [13:0] xB1w, xB2w, xB3w, xB4w;
wire [13:0] xC1w, xC2w, xC3w, xC4w;
wire [13:0] xRef1w, xRef2w, xRef3w, xRef4w;

//Wires used to connect output history buffers together for each channel (A,B,C and Ref)
wire [13:0] yA1w, yA2w, yA3w, yA4w;
wire [13:0] yB1w, yB2w, yB3w, yB4w;
wire [13:0] yC1w, yC2w, yC3w, yC4w;
wire [13:0] yRef1w, yRef2w, yRef3w, yRef4w;

//Center specifies the total maximum number that can be accumulated in icount*,qcount*,imcount* and qmcount*.  high and low are configurable by
//the ZWorld to specify what the actual accumulated values will be
wire [10:0] center, high, low;

/*************************   Triggered Events   *************************/

/*
  Negative Edge of clk40

  Overview:  The input from the ADCs is sampled and shuffled into the history buffers.  The output is also generated and shuffled into history
```

```
 buffers.  The I,Q,-I and -Q values are generated by comparing the current input vs. the previous output of the same I,Q,-I or -Q value.  If the
 current input is greater than or less than the last output a counter is incremented or decremented.  Once the counter reaches the value
 stored in either high or low the output for that I,Q,-I or -Q value is incremented or decremented.  High and low are specified by the ZWorld
 by giving an offset from center, which is set at 1000.  This gives a maximum offset of 1000 counts before any of the I,Q,-I or -Q values will
 be changed.  The currently read input must be higher or lower consecutively for the specified number of cycles otherwise the counter will reset.
 This gives the system a response time of 10,000,000*0.02/(high - center) deg/sec (360/2^14 = .021 deg/bit)
*/
always @(negedge clk40)
begin
//Shuffle the input history buffers. Continuously assigned wires are registerd on negedge of clk40
 xA1 <= xA1w; xA2 <= xA2w; xA3 <= xA3w; xA4 <= xA4w;
        xB1 <= xB1w; xB2 <= xB2w; xB3 <= xB3w; xB4 <= xB4w;
        xC1 <= xC1w; xC2 <= xC2w; xC3 <= xC3w; xC4 <= xC4w;
        xRef1 <= xRef1w; xRef2 <= xRef2w; xRef3 <= xRef3w; xRef4 <= xRef4w;

//Shuffle the output history buffers. Continuously assigned wires are registerd on negedge of clk40
        yA1 <= yA1w; yA2 <= yA2w; yA3 <= yA3w; yA4 <= yA4w;
        yB1 <= yB1w; yB2 <= yB2w; yB3 <= yB3w; yB4 <= yB4w;
        yC1 <= yC1w; yC2 <= yC2w; yC3 <= yC3w; yC4 <= yC4w;
        yRef1 <= yRef1w; yRef2 <= yRef2w; yRef3 <= yRef3w; yRef4 <= yRef4w;

//Generate the current output by using a 10MHz bandpass filter y[n] = 0.5*x[n] - 0.5*y[n-2].  This is done by a logical right shift of x[n] by
//1 bit and inverting y[n-2] and logically shifting left by 1 and then adding 1
        yAOut <= {1'b0,xA1w[13:1]} + {1'b0,(~yA2w[13:1])} + 1'b1;
        yBOut <= {1'b0,xB1w[13:1]} + {1'b0,(~yB2w[13:1])} + 1'b1;
        yCOut <= {1'b0,xC1w[13:1]} + {1'b0,(~yC2w[13:1])} + 1'b1;
        yRefOut <= {1'b0,xRef1w[13:1]} + {1'b0,(~yRef2w[13:1])} + 1'b1;
//This 2-bit counter keeps track of I,Q,-I and -Q samples 2'b00 = I, 2'b01 = Q, 2'b10 = -I, 2'b11 = -Q
        counter = counter + 1;
/*
 Case statement adjusts the I,Q,-I and -Q values based on the state of counter.  Each idividual counter must be equal to either high or low to
 either increment or decrement the I,Q,-I or -Q value for each channel (A,B,C and Ref) Ex. icountA must be equal to high for IVal_A to increment
 by 1. icountA must be equal to low for IVal_A to decrement by 1.  icountA will reset to center if any consecutive cycles are not either both
 greater than the previous output or less than the previous output.
*/
        case (counter[1:0])
```

```
2'b00:  //I Section
begin
//This conditional assignment reads: if y[n-2] is greater than the current value being sent to the ZWorld and icountA is greater than or equal to
//the center value and icountA is not greater than the high value then increase icountA by 1, otherwise reset icountA to center.  If y[n-2] is
//less thant the current value being sent to the ZWorld and icountA is less than or equal to the center value and icountA is not less than low
//then decrease icountA by 1, otherwise reset it to center.
  icountA <= (yA2w > IVal_A) ? ( (icountA >= center && icountA < high) ? icountA + 1 : center ) :
                               ( (icountA <= center && icountA > low) ? icountA - 1 : center );
//if icountA is equal to high then increase IVal_A, elseif icountA is equal to low then decrease IVal_A, else leave it alone.
//This is the same for each channel and each I,Q,-I and -Q
  IVal_A <= (icountA == high) ? IVal_A + 1'b1 : (icountA == low) ? IVal_A - 1'b1 : IVal_A;
  icountB <= (yB2w > IVal_B) ? ( (icountB >= center && icountB < high) ? icountB + 1 : center ) :
                               ( (icountB <= center && icountB > low) ? icountB - 1 : center );
  IVal_B <= (icountB == high) ? IVal_B + 1'b1 : (icountB == low) ? IVal_B - 1'b1 : IVal_B;
  icountC <= (yC2w > IVal_C) ? ( (icountC >= center && icountC < high) ? icountC + 1 : center ) :
                               ( (icountC <= center && icountC > low) ? icountC - 1 : center );
  IVal_C <= (icountC == high) ? IVal_C + 1'b1 : (icountC == low) ? IVal_C - 1'b1 : IVal_C;
  icountRef <= (yRef2w > IValRef) ? ( (icountRef >= center && icountRef < high) ? icountRef + 1 : center ) :
                               ( (icountRef <= center && icountRef > low) ? icountRef - 1 : center );
  IValRef <= (icountRef == high) ? IValRef + 1'b1 : (icountRef == low) ? IValRef - 1'b1 : IValRef;
end
2'b01:  //Q Section
begin
  qcountA <= (yA2w > QVal_A) ? ( (qcountA >= center && qcountA < high) ? qcountA + 1 : center ) :
                               ( (qcountA <= center && qcountA > low) ? qcountA - 1 : center );
  QVal_A <= (qcountA == high) ? QVal_A + 1'b1 : (qcountA == low) ? QVal_A - 1'b1 : QVal_A;
  qcountB <= (yB2w > QVal_B) ? ( (qcountB >= center && qcountB < high) ? qcountB + 1 : center ) :
                               ( (qcountB <= center && qcountB > low) ? qcountB - 1 : center );
  QVal_B <= (qcountB == high) ? QVal_B + 1'b1 : (qcountB == low) ? QVal_B - 1'b1 : QVal_B;
  qcountC <= (yC2w > QVal_C) ? ( (qcountC >= center && qcountC < high) ? qcountC + 1 : center ) :
                               ( (qcountC <= center && qcountC > low) ? qcountC - 1 : center );
  QVal_C <= (qcountC == high) ? QVal_C + 1'b1 : (qcountC == low) ? QVal_C - 1'b1 : QVal_C;
  qcountRef <= (yRef2w > QValRef) ? ( (qcountRef >= center && qcountRef < high) ? qcountRef + 1 : center ) :
                               ( (qcountRef <= center && qcountRef > low) ? qcountRef - 1 : center );
  QValRef <= (qcountRef == high) ? QValRef + 1'b1 : (qcountRef == low) ? QValRef - 1'b1 : QValRef;
end
```

```
        2'b10:  //-I Section
        begin
          imcountA <= (yA2w > ImVal_A) ? ( (imcountA >= center && imcountA < high) ? imcountA + 1 : center ) :
                                         ( (imcountA <= center && imcountA > low) ? imcountA - 1 : center );
          ImVal_A <= (imcountA == high) ? ImVal_A + 1'b1 : (imcountA == low) ? ImVal_A - 1'b1 : ImVal_A;
          imcountB <= (yB2w > ImVal_B) ? ( (imcountB >= center && imcountB < high) ? imcountB + 1 : center ) :
                                         ( (imcountB <= center && imcountB > low) ? imcountB - 1 : center );
          ImVal_B <= (imcountB == high) ? ImVal_B + 1'b1 : (imcountB == low) ? ImVal_B - 1'b1 : ImVal_B;
          imcountC <= (yC2w > ImVal_C) ? ( (imcountC >= center && imcountC < high) ? imcountC + 1 : center ) :
                                         ( (imcountC <= center && imcountC > low) ? imcountC - 1 : center );
          ImVal_C <= (imcountC == high) ? ImVal_C + 1'b1 : (imcountC == low) ? ImVal_C - 1'b1 : ImVal_C;
          imcountRef <= (yRef2w > ImValRef) ? ( (imcountRef >= center && imcountRef < high) ? imcountRef + 1 : center ) :
                                         ( (imcountRef <= center && imcountRef > low) ? imcountRef - 1 : center );
          ImValRef <= (imcountRef == high) ? ImValRef + 1'b1 : (imcountRef == low) ? ImValRef - 1'b1 : ImValRef;
        end
        2'b11:  //-Q Section
        begin
          qmcountA <= (yA2w > QmVal_A) ? ( (qmcountA >= center && qmcountA < high) ? qmcountA + 1 : center ) :
                                         ( (qmcountA <= center && qmcountA > low) ? qmcountA - 1 : center );
          QmVal_A <= (qmcountA == high) ? QmVal_A + 1'b1 : (qmcountA == low) ? QmVal_A - 1'b1 : QmVal_A;
          qmcountB <= (yB2w > QmVal_B) ? ( (qmcountB >= center && qmcountB < high) ? qmcountB + 1 : center ) :
                                         ( (qmcountB <= center && qmcountB > low) ? qmcountB - 1 : center );
          QmVal_B <= (qmcountB == high) ? QmVal_B + 1'b1 : (qmcountB == low) ? QmVal_B - 1'b1 : QmVal_B;
          qmcountC <= (yC2w > QmVal_C) ? ( (qmcountC >= center && qmcountC < high) ? qmcountC + 1 : center ) :
                                         ( (qmcountC <= center && qmcountC > low) ? qmcountC - 1 : center );
          QmVal_C <= (qmcountC == high) ? QmVal_C + 1'b1 : (qmcountC == low) ? QmVal_C - 1'b1 : QmVal_C;
          qmcountRef <= (yRef2w > QmValRef) ? ( (qmcountRef >= center && qmcountRef < high) ? qmcountRef + 1 : center ) :
                                         ( (qmcountRef <= center && qmcountRef > low) ? qmcountRef - 1 : center );
          QmValRef <= (qmcountRef == high) ? QmValRef + 1'b1 : (qmcountRef == low) ? QmValRef - 1'b1 : QmValRef;
        end
        endcase
end

/*
  Positive Edge of clk40
```

```
        Overview:  Based on the value of the 2-bit mux ADC_Select each channel can be sent to the DAC output. The handshaking bit from the ZWorld
board
        is sampled and stored in handWorldReg and the command received output register is updated to reflect a completed command.  Finally, as long as
        the update is not frozen, every 4 clk40 cycles all of the I,Q,-I and -Q values are updated for each channel (A,B,C and Ref) simultaneously so
        that they may be read by the ZWorld.
    */
    always @(posedge clk40)
    begin
        //case statement to set the DAC output 2'b00 -> channel 1, 2'b01 -> channel 2,
        //2'b10 -> channel 3, 2'b11 -> Reference Channel.
      case (ADC_Select)
        2'b00:
          ADC_Out <= yAOut[13:0];
      2'b01:
          ADC_Out <= yBOut[13:0];
        2'b10:
          ADC_Out <= yCOut[13:0];
        2'b11:
          ADC_Out <= yRefOut[13:0];
        endcase

        //check for a command being sent by the ZWorld by sampling the handshaking pin
        handZWorldReg <= handZWorld;
        //update the command received reg to reflect the status of command processing to the ZWorld
        command_received_reg <= command_received;

        //if not frozen and on the 4th cycle, update the values read by the ZWorld
        if (freeze == 1'b0 && counter == 2'b11)
        begin
       IValOut_A <= {2'b00,IVal_A[13:0]};
          IValOut_B <= {2'b00,IVal_B[13:0]};
          IValOut_C <= {2'b00,IVal_C[13:0]};
          IValRefOut <= {2'b00,IValRef[13:0]};
          QValOut_A <= {2'b00,QVal_A[13:0]};
          QValOut_B <= {2'b00,QVal_B[13:0]};
          QValOut_C <= {2'b00,QVal_C[13:0]};
```

```
            QValRefOut <= {2'b00,QValRef[13:0]};
            ImValOut_A <= {2'b00,ImVal_A[13:0]};
            ImValOut_B <= {2'b00,ImVal_B[13:0]};
            ImValOut_C <= {2'b00,ImVal_C[13:0]};
            ImValRefOut <= {2'b00,ImValRef[13:0]};
            QmValOut_A <= {2'b00,QmVal_A[13:0]};
            QmValOut_B <= {2'b00,QmVal_B[13:0]};
            QmValOut_C <= {2'b00,QmVal_C[13:0]};
            QmValRefOut <= {2'b00,QmValRef[13:0]};
    end
  end

  /*
    Either Edge of handZWorldReg

        Overview: Whenever the handshaking pin from the ZWorld board toggles the FPGA needs to handle the event and respond to any commands.  The
        handshaking works as follows:  handZWorldReg goes from low to high indicating a command is being sent to the FPGA.  The FPGA reads the
command
        pins and the DigIn pins and processes the command, setting any needed outputs and toggles command_received from 0 to 1.  The ZWorld reads
        the command_received pin and reads in any data on DigOut from the FPGA then toggles handZWorldReg from 1 to 0 at which point the FPGA
responds
        by toggling command_received from 1 to 0.  This section sets the digital attenuators, the ADC modes, the filter offset and transmits all of the
        I,Q,-I and -Q data to the ZWorld
  */
  always @(handZWorldReg)
  begin
    //if handZWorldReg is equal to 1 read a command and process it
    if (handZWorldReg == 1'b1)
        begin
          command_received <= 1'b1;
          /*
            Commands:
                4'b0000         Set the digital attenuator with the value on DigIn (6'b000000 to 6'b111111)
                4'b0001         Set the ADC mode to the value on DigIn (2'b00 to 2'b11)
                4'b0010         Read each channel's I value, freeze the the I,Q,-I,-Q value update
                4'b0011         Read each channel's Q value, freeze the the I,Q,-I,-Q value update
```

```
                4'b0100                 Read each channel's -I value, freeze the the I,Q,-I,-Q value update
                4'b0101                 Read each channel's -Q value, freeze the the I,Q,-I,-Q value update
                4'b0110                 Read the high 8 bits of the 10 bit filter value
                4'b0111                 Read the low 2 bits of the 10 bit filter value and create the 10 bit value
                4'b1000                 unfreeze the the I,Q,-I,-Q value update, only called after 4'b0010 through 4'b0101 are compeleted
        */
case (command)
        4'b0000:        //Set digital Attenuator values
          begin
                freeze <= 1'b0;
    attenOut <= DigIn;
          end
              4'b0001: //Set the ADC mode 2'b00 -> standard sampling, 2'b01 -> output all 0s, 2'b10 -> output all 1s,
                begin    //2'b11 -> output string of 1s and 0s
                  freeze <= 1'b0;
                  ADC_Select <= DigIn[1:0];
          end
              4'b0010:          //Output the I value based on DigIn, output the high byte or low byte of either channels A,B,C or Ref
                begin             //If DigIn[3] is 1 output Ref values, elseif DigIn[2] is 1 output channel C, elseif DigIn[1] is 1 output channel B,
                  freeze <= 1'b1; //else output channel A.  If DigIn[0] is 1 output low byte else output high byte.
                  DigitalOut = DigIn[3] ? (DigIn[0] ? IValRefOut[7:0] : IValRefOut[15:8]) :
                    (DigIn[2] ? (DigIn[0] ? IValOut_C[7:0] : IValOut_C[15:8]) :
                            (DigIn[1] ? (DigIn[0] ? IValOut_B[7:0] : IValOut_B[15:8]) :
                              (DigIn[0] ? IValOut_A[7:0] : IValOut_A[15:8])));
                end
              4'b0011: //Same as I, but output Q values
                begin
                  freeze <= 1'b1;
                  DigitalOut = DigIn[3] ? (DigIn[0] ? QValRefOut[7:0] : QValRefOut[15:8]) :
                    (DigIn[2] ? (DigIn[0] ? QValOut_C[7:0] : QValOut_C[15:8]) :
                            (DigIn[1] ? (DigIn[0] ? QValOut_B[7:0] : QValOut_B[15:8]) :
                              (DigIn[0] ? QValOut_A[7:0] : QValOut_A[15:8])));
                end
              4'b0100: //Same as I, but output -I values
                begin
                  freeze <= 1'b1;
```

```
                DigitalOut = DigIn[3] ? (DigIn[0] ? ImValRefOut[7:0] : ImValRefOut[15:8]) :
                  (DigIn[2] ? (DigIn[0] ? ImValOut_C[7:0] : ImValOut_C[15:8]) :
                        (DigIn[1] ? (DigIn[0] ? ImValOut_B[7:0] : ImValOut_B[15:8]) :
                          (DigIn[0] ? ImValOut_A[7:0] : ImValOut_A[15:8])));
              end
            4'b0101:  //Same as I, but output -Q values
              begin
                freeze <= 1'b1;
                DigitalOut = DigIn[3] ? (DigIn[0] ? QmValRefOut[7:0] : QmValRefOut[15:8]) :
                  (DigIn[2] ? (DigIn[0] ? QmValOut_C[7:0] : QmValOut_C[15:8]) :
                        (DigIn[1] ? (DigIn[0] ? QmValOut_B[7:0] : QmValOut_B[15:8]) :
                          (DigIn[0] ? QmValOut_A[7:0] : QmValOut_A[15:8])));
              end
            4'b0110:  //Read in the high 8 bits of the 10 bit filter value
              begin
                freeze <= 1'b0;
                filter_high_byte <= DigIn;
              end
            4'b0111:  //Read in the low 2 bits of the 10 bit filter value and create high and low
              begin
                freeze <= 1'b0;
                high_reg <= center + {filter_high_byte,DigIn[1:0]};
                low_reg <= center - {filter_high_byte,DigIn[1:0]};
              end
    4'b1000:      //unfreeze the I,Q,-I and -Q ZWorld value update
              begin
                freeze <= 1'b0;
              end
        endcase
 end
        else
        begin                   //if handZWorldReg is equal to 0 change command_received to 0
          command_received <= 1'b0;
        end
end
```

```
/*************************  Continuous Assignments  *************************/

    /*
     This section contains all of the contiually assigned wires and outputs that are connected to the outputs of registers.  As soon
      as the registers used to assign these values are updated, the values at the output or on the wire will be updated
    */

    assign Attenuators = attenOut;       //assign the 6 bit output Attenuators with the value stored in the attenOut register
    assign DigOut = DigitalOut;          //assign the 8 bit output DigOut with the value stored in the DigitalOut register
    assign DAC = ADC_Out;                          //assign the 14 bit output DAC with the value stored in the ADC_Out register
    assign handFPGA = command_received_reg; //assign the 1 bit output handFPGA with the value stored in the command_received_reg register

    //assign the 11 bit wire center with the default value 11'd1000. Assign the wires high and low with the appropriate register value
    assign center = 11'd1000;            assign high = high_reg; assign low = low_reg;

    //assign the input history wires to the appropriate inputs and registers.  x*1w is assigned to the 14 bit ADC input and the others are assigned
    //to the outputs of the registers before them.  This is done for each channel (A,B,C and Ref) which retains the last 4 input values (x[n], x[n-1]
    //x[n-2], x[n-3])
    assign xA1w = ADC_A; assign xA2w = xA1; assign xA3w = xA2; assign xA4w = xA3;
    assign xB1w = ADC_B; assign xB2w = xB1; assign xB3w = xB2; assign xB4w = xB3;
    assign xC1w = ADC_C; assign xC2w = xC1; assign xC3w = xC2; assign xC4w = xC3;
    assign xRef1w = ADC_D; assign xRef2w = xRef1; assign xRef3w = xRef2; assign xRef4w = xRef3;

    //assign the output history wires to the appropriate registers.  y*1w is assigned to the least significant 14 bits of the current output and the
    //others are assigned to the outputs of the registers before them.  This is done for each channel (A,B,C and Ref) which retains the last 5 output
    //values (y[n], y[n-1], y[n-2], y[n-3], y[n-4])
    assign yA1w = yAOut[13:0]; assign yA2w = yA1; assign yA3w = yA2; assign yA4w = yA3;
    assign yB1w = yBOut[13:0]; assign yB2w = yB1; assign yB3w = yB2; assign yB4w = yB3;
    assign yC1w = yCOut[13:0]; assign yC2w = yC1; assign yC3w = yC2; assign yC4w = yC3;
    assign yRef1w = yRefOut[13:0]; assign yRef2w = yRef1; assign yRef3w = yRef2; assign yRef4w = yRef3;

endmodule
```

APPENDIX B

ZWorld C-Code

```
//Variable Initialization
int initcount, config, setAtten, ADCcommand, ADCVal, get_IQ, PLL_Setup,
    ReadIA, ReadIB, ReadQA, ReadQB, ReadImA, ReadImB, ReadQmA, ReadQmB,
     ReadIC, ReadQC, ReadImC, ReadQmC, IA_Val_Read, IB_Val_Read, QA_Val_Read,
      QB_Val_Read, ImA_Val_Read, ImB_Val_Read, QmA_Val_Read, QmB_Val_Read,
       IC_Val_Read, QC_Val_Read, ImC_Val_Read, QmC_Val_Read, IRef_Val_Read,
        QRef_Val_Read, ImRef_Val_Read, QmRef_Val_Read, ReadIRef, ReadQRef,
         ReadImRef, ReadQmRef, filterVal;


float ReadIA_Mag, ReadIB_Mag, ReadIC_Mag, ReadQA_Mag, ReadQB_Mag, ReadQC_Mag,
    ReadImA_Mag, ReadImB_Mag, ReadImC_Mag, ReadQmA_Mag, ReadQmB_Mag,
     ReadQmC_Mag, phaseA, IValA, QValA, magnitudeA, phaseB, IValB, QValB,
      magnitudeB, phaseC, IValC, QValC, magnitudeC, phaseAB, phaseAC, phaseBC,
       divide_val, phaseRef, IValRef, QValRef, magnitudeRef, ReadIRef_Mag,
        ReadQRef_Mag, ReadImRef_Mag, ReadQmRef_Mag, offsetAB, offsetAC;

ulong  rcount, ncount, OutVal;

bool  writeSPIData, writeInit, writeRCount, writeNCount, writeConfig,
     changeAtten, newAtten, fpgaBusy, writeADC, newOutVal, changeADC,
     getting_IQ, firstRun, New_IQ, changeFilter, changeFilter2, filtered;

//Telnet User Interface Declaration
xstring DevStateStr {
"DI-00: .              DO-00:  .          DO-16:  .",
"DI-01: .              DO-01:  .          DO-17:  .",
"DI-02: .              DO-02:  .          DO-18:  .",
"DI-03: .              DO-03:  .          DO-19:  .",
"DI-04: .              DO-04:  .          DO-20:  .",
"DI-05: .              DO-05:  .          DO-21:  .",
"DI-06: .              DO-06:  .          DO-22:  .",
"DI-07: .              DO-07:  .          DO-23:  .",
"                              Ch1 Phase:       Degrees ",
"DI-08: .              DO-08:  .      Ch1 Mag:        Vpp ",
"DI-09: .              DO-09:  .     Ch2 Phase:       Degrees ",
"/INIT: .              DO-10:  .      Ch2 Mag:        Vpp ",
"DONE:  .              DO-11:  .     Ch3 Phase:       Degrees ",
"DI-12: .              DO-12:  .      Ch3 Mag:        Vpp ",
"Ref Phase:            DO-13:  .     Ch1 - Ch2:       Degrees ",
"  Ref Mag:            PROG:   .     Ch1 - Ch3:       Degrees ",
" OffsetAB:            DO-15:  .     Ch2 - Ch3:       Degrees ",
" OffsetAC:                        Dig. Atten: ",
""
};

//---------------------------------------------------------------------------
//  process a multi-character command from the diagnostic terminal/console
//---------------------------------------------------------------------------
void ProcessStrCmd(char *cmd, LinkProc *lp)
{
  char  *ptr;
  int  i, chan, state, digState;
  ulong  mask, OutValLong;
  float  volts;
  bool  validCmd;
  char  *msg, *temp, *cmdStr;
```

```
  StackPtr();

  msg = getBuf(90 + 90 + 90);  temp = msg + 90;  cmdStr = temp + 90;

  strcpy(cmdStr, cmd);  ptr = strtok(cmd, " ");
  state = 0;  validCmd = false;
  strcpy(msg, "cmd parser error");

  while (ptr AND (state != 999))  {

//select parse the user input command
    switch (state)  {

      case 0:  // which command did they type?
        if (strcmpi(ptr, "offsetab") == 0) {state = 101; break; }
        if (strcmpi(ptr, "setadc") == 0) {state = 102; break;}
        if (strcmpi(ptr, "dig") == 0)  { state = 103;  break; }
        if (strcmpi(ptr, "init") == 0) {state = 104; break; }
        if (strcmpi(ptr, "rcount") == 0) {state = 105; break; }
        if (strcmpi(ptr, "ncount") == 0) {state = 106; break; }
        if (strcmpi(ptr, "config") == 0) {state = 107; break; }
        if (strcmpi(ptr, "setatten") == 0) {state = 108; break;}
        if (strcmpi(ptr, "Read") == 0) {state = 109; break;}
        if (strcmpi(ptr, "filter") == 0) {state = 110; break;}
        if (strcmpi(ptr, "offsetac") == 0) {state = 111; break; }
        sprintf(msg, "Unrecognized command: %s", ptr);
        state = 999;  break;

      case 101:  //change the offset value stored for channel 1 to channel 2
        offsetAB = atof(ptr);
        if ((offsetAB < -180) OR (offsetAB > 180)) {
          sprintf(msg, "Offset must be between -180 and 180 degrees");
          state = 999; break;  }
        sprintf(msg, "Offset has been set to %d ", offsetAB);
        validCmd = true; state = 999; break;

      case 102: //set the ADC mode
        ADCcommand = atoi(ptr);
        if ((ADCcommand < 0) OR (ADCcommand > 3))  {
          sprintf(msg, "%s is an Invalid ADC Setting", ptr); state = 999;
          break;  }
        ++criticalSection;
        writeSPIData = writeADC = true;
        --criticalSection;
        sprintf(msg, "Configuring the ADCs to test mode %d", ADCcommand);
        validCmd = true; state = 999; break;


      case 103:  // set state of a Digital output channel
        chan = atoi(ptr);
        if ((chan < 0) OR (chan > 23))  {
          sprintf(msg, "Invalid DigOut chan #: %s", ptr);  state = 999;
          break; }
        if (! (ptr = strtok(NULL, " ")))  break;
        digState = atoi(ptr);
        sprintf(msg, "Setting DigOut bit %d to: %s",
              chan, digState ? "On" : "Off");
        mask = 1L << chan;
        ++criticalSection;
        if (digState)  digitalOut |= mask;  else  digitalOut &= ~mask;
        --criticalSection;
        validCmd = true;  state = 999;  break;
```

```
case 104: //configure initialization register
  initcount = atoi(ptr);
  if ((initcount < 0) OR (initcount > 7 ))  {
    sprintf(msg, "Initialization command out of valid range: %s", ptr);
    state = 999;  break;
  }
  ++criticalSection;
  writeSPIData = writeInit = true;
  --criticalSection;
  sprintf(msg, "Configuring Initilization Register");
  validCmd = true; state = 999; break;

case 105: //configure the R-Count Register
  rcount = atoi(ptr);
  if ((rcount < 1) OR (rcount > 1023))  {
    sprintf(msg, "Divide value out of valid range: %s", ptr);
    state = 999;  break;
  }
  ++criticalSection;
  writeSPIData = writeRCount = true;
  --criticalSection;
  sprintf(msg, "Setting the R-Count Register to %d", rcount);
  validCmd = true; state = 999; break;

case 106: //configure the N-Count Register
  ncount = atoi(ptr);
  if ((ncount < 1) OR (ncount > 1023))  {
    sprintf(msg, "Multiply value out of valid range: %s", ptr);
    state = 999;  break;
  }
  ++criticalSection;
  writeSPIData = writeNCount = true;
  --criticalSection;
  sprintf(msg, "Setting the N-Count Register to %d", ncount);
  validCmd = true; state = 999;  break;

case 107: //configure the configuration register
  config = atoi(ptr);
  if ((config < 0) OR (config > 7))  {
    sprintf(msg, "Configuration command out of valid range:  %s", ptr);
    state = 999; break;
  }
  ++criticalSection;
  writeSPIData = writeConfig = true;
  --criticalSection;
  sprintf(msg, "Configuring register");
  validCmd = true; state = 999;  break;

case 108: //change the digital attenuator value
  setAtten = atoi(ptr);
  if ((setAtten < 5) OR (setAtten > 45)) {
    sprintf(msg, "Atteunation setting out of range: %s", ptr);
    state = 999; break;
  }
  sprintf(msg, "Setting the digital attenuators to %s db", ptr);
  ++criticalSection;  newAtten = true;  --criticalSection;
  validCmd = true; state = 999; break;

case 109: //set the DAC source to channel 1-4
  ADCVal = atoi(ptr) - 1;
  if ((ADCVal < 0) OR (ADCVal > 3))  {
```

```
        sprintf(msg, "%d is not a valid ADC.  Select ADC 1-4", ADCVal + 1);
        state = 999; break;  }
      sprintf(msg, "Now reading from ADC %d", ADCVal + 1);
      ++criticalSection; changeADC = true; --criticalSection;
      validCmd = true; state = 999; break;

    case 110: //set the filter factor
      filterVal = atoi(ptr);
      if ((filterVal < 0) OR (filterVal > 1000)) {
        sprintf(msg, "%d is not a valid filter value.  Select 0-1000",
          filterVal);
        state = 999; break; }
      sprintf(msg, "Setting Filter Value to %d", filterVal);
      ++criticalSection; changeFilter = true; --criticalSection;
      validCmd = true; state = 999; break;

    case 111: //set the channel offset between channels 1 and 3
      offsetAC = atof(ptr);
      if ((offsetAC < -180) OR (offsetAC > 180)) {
        sprintf(msg, "Offset must be between -180 and 180 degrees");
        state = 999; break;  }
      sprintf(msg, "Offset has been set to %d ", offsetAC);
      validCmd = true; state = 999; break;
   }
   if (ptr)  ptr = strtok(NULL, " ");
  }
  strCmdState = false;



//--- what state were we in when we ran out of cmd string to parse? ---
  switch (state)  {

    case 999:  // msg supplied by cmd parser state machine
      break;

    default:
      strcpy(msg, "Incomplete command");
  }

  if (! validCmd)
    ProcessStrCmdCommon(cmdStr, msg, lp);  // not valid dev cmd - check generic ones
  else
    if (*msg)
      if (StreamDisp)
        ShowMsg(msg);
      else  {
        respShownTime = MS_TIMER - 5000;
        DispStr(0,23, msg, true);
        respShownTime = MS_TIMER;
      }

  freeTo(msg);
}

//---------------------------------------------------------------------------
//  Update data based on digital and analog input values and determine new
//  values for the digital and analog outputs
//
//  !!!!! WARNING !!!!!
//
//  This function is called by the timer-interrupt driven function that reads
```

```
//  and writes the Analog and Digital I/O values.  It should be limited to
//  processing current input values and updating output values that will become
//  the active values on the next interrupt (currently, this interuupt occurs
//  every 10ms, so this routine must take considerably less time than that to
//  do EVERYTHING it needs to).
//---------------------------------------------------------------------------
nodebug void UpdateDeviceState()
{
  ulong  bit, secTime, mask, attenBits, OutValLong, ADCValLong, filterValLong;
  int  i, rate;
  char  buf[4];

  divide_val = 8192.0;
//Verify the ADC chip select is high to prohibit SPI loads on the ADCs
  digitalOut = digitalOut | 0x008000; WriteDigOutputs();

//on initial power up, configure the PLL and ADCs to run in default mode
  if (firstRun)
  {
    switch (PLL_Setup)
    {
      case 0:
        writeSPIData = writeInit = true; initcount = 1; PLL_Setup++; break;
      case 1:
        writeSPIData = writeConfig = true; config = 1; PLL_Setup++; break;
      case 2:
        writeSPIData = writeRCount = true; rcount = 4; PLL_Setup++; break;
      case 3:
        writeSPIData = writeNCount = true; ncount = 1; PLL_Setup++; break;
      case 4:
        writeSPIData = writeADC = true; ADCcommand = 0; firstRun = false; break;
      default:
        firstRun = false; break;
    }
  }

//if a new set of I/Q values have been read from the FPGA, calculate the phase
if (New_IQ)
  {
//Convert the 14-bit integer values read in for I, Q, -I and –Q for each channel to
//floating point numbers between –1 and 1
    ReadIA_Mag = ((float)ReadIA/divide_val)-1;
    ReadIB_Mag = ((float)ReadIB/divide_val)-1;
    ReadIC_Mag = ((float)ReadIC/divide_val)-1;
    ReadIRef_Mag = ((float)ReadIRef/divide_val)-1;
    ReadImA_Mag = ((float)ReadImA/divide_val)-1;
    ReadImB_Mag = ((float)ReadImB/divide_val)-1;
    ReadImC_Mag = ((float)ReadImC/divide_val)-1;
    ReadImRef_Mag = ((float)ReadImRef/divide_val)-1;
    ReadQA_Mag = ((float)ReadQA/divide_val)-1;
    ReadQB_Mag = ((float)ReadQB/divide_val)-1;
    ReadQC_Mag = ((float)ReadQC/divide_val)-1;
    ReadQRef_Mag = ((float)ReadQRef/divide_val)-1;
    ReadQmA_Mag = ((float)ReadQmA/divide_val)-1;
    ReadQmB_Mag = ((float)ReadQmB/divide_val)-1;
    ReadQmC_Mag = ((float)ReadQmC/divide_val)-1;
    ReadQmRef_Mag = ((float)ReadQmRef/divide_val)-1;

//Determine the I and Q values for each channel
    IValA = (ReadIA_Mag - ReadImA_Mag)/2;
    QValA = (ReadQA_Mag - ReadQmA_Mag)/2;
    IValB = (ReadIB_Mag - ReadImB_Mag)/2;
```

```
    QValB = (ReadQB_Mag - ReadQmB_Mag)/2;
    IValC = (ReadIC_Mag - ReadImC_Mag)/2;
    QValC = (ReadQC_Mag - ReadQmC_Mag)/2;
    IValRef = (ReadIRef_Mag - ReadImRef_Mag)/2;
    QValRef = (ReadQRef_Mag - ReadQmRef_Mag)/2;

  //Determine the magnitude of each channel
    magnitudeA = sqrt(QValA*QValA + IValA*IValA);
    magnitudeB = sqrt(QValB*QValB + IValB*IValB);
    magnitudeC = sqrt(QValC*QValC + IValC*IValC);
    magnitudeRef = sqrt(QValRef*QValRef + IValRef*IValRef);

  //calculate the phase for each channel, if there is no data being read from the //FPGA, output 0
  //Refernence Channel phase
    if ((QValRef == 0 AND IValRef == 0) | (magnitudeRef < 0.01))
    {
      phaseRef = 0.0;
    }
    else
    {
      phaseRef = atan2(QValRef,IValRef)/PI*180.0;
      if (phaseRef < 0) phaseRef = 360 + phaseRef;
    }
  //Channel 1 phase
    if ((QValA == 0 AND IValA == 0) | (magnitudeA < 0.05))
    {
      phaseA = 0.0;
    }
    else
    {
      phaseA = atan2(QValA, IValA)*180.0/PI-phaseRef;
      if (phaseA < 0) phaseA = 360 + phaseA;
    }
  //Channel 2 phase
    if ((QValB == 0 AND IValB == 0) | (magnitudeB < 0.05))
    {
      phaseB = 0.0;
    }
    else
    {
      phaseB = atan2(QValB,IValB)*180.0/PI-phaseRef;
      if (phaseB < 0) phaseB = 360 + phaseB;
    }
  //channel 3 phase
    if ((QValC == 0 AND IValC == 0) | (magnitudeC < 0.05))
    {
      phaseC = 0.0;
    }
    else
    {
      phaseC = atan2(QValC,IValC)*180.0/PI-phaseRef;
      if (phaseC < 0) phaseC = 360 + phaseC;
    }

  //calculate the phases between the sets of channels
  //channel 1 to channel 2
    if (phaseA == 0 OR phaseB == 0)
      phaseAB = 0;
    else
      phaseAB = phaseA - phaseB - offsetAB;
  //channel 1 to channel 3
    if (phaseA == 0 OR phaseC == 0)
```

```
      phaseAC = 0;
    else
      phaseAC = phaseA - phaseC - offsetAC;
//channel 2 to channel 3
    if (phaseB == 0 OR phaseC == 0)
      phaseBC = 0;
    else
      phaseBC = phaseB - phaseC;
//convert the phase to –180 to 180 scale
    if (phaseAB > 180) phaseAB = phaseAB - 360;
    else if (phaseAB < -180) phaseAB = 360 + phaseAB;

    if (phaseAC > 180) phaseAC = phaseAC - 360;
    else if (phaseAC < -180) phaseAC = 360 + phaseAC;

    if (phaseBC > 180) phaseBC = phaseBC - 360;
    else if (phaseBC < -180) phaseBC = 360 + phaseBC;

//allow new I/Q data to be collected
    New_IQ = false;
  }

//--- if we don't have "possesion" of the SPI connection to the FPGA, then ---
//--- try to get it again (the ComLoop() process can take it away when ---
//--- temporarily needed by a remote client) ---
  if (! fpgaLink)  fpgaLink =
     (ComLink *) SPILink_(FPGA_SPIPort, CLK_1MHz, NORM_LOW_LATCH_RISE);

//--- use the SPI link (If we currently own it) to talk to the FPGA ---
  if (writeSPIData AND fpgaLink)  {
//  Set up SPI data for the High Speed ADCs
    if (writeADC)
    {
      SPILink_reconfig((SPILink *)fpgaLink, CLK_1MHz, NORM_HIGH_LATCH_FALL);
      buf[0] = 0xe0 | (0x06 & (ADCcommand << 1));
      buf[1] = 0x00;
    }
//set up SPI data for the PLL to set the Initialization Register
    else if (writeInit)
    {
      buf[0] = 0x1f;
      buf[1] = 0x80;
      buf[2] = (0x70 & initcount << 4) | 0x83;
    }
//set up SPI data for the PLL to set the R-Count Register
    else if (writeRCount)
    {
      buf[0] = 0x01;
      buf[1] = (char)(0x00f & rcount >> 6);
      buf[2] = (char)(0x0fc & rcount << 2);
    }
//set up SPI data for the PLL to set the N-Count Register
    else if (writeNCount)
    {
      buf[0] = (char)(0x003 & ncount >> 8);
      buf[1] = (char)(0x0ff & ncount);
      buf[2] = 0x01;
    }
//set up SPI data for the PLL to set the Configuration Register
    else if (writeConfig)
    {
      buf[0] = 0x1f;
```

```
    buf[1] = 0x80;
    buf[2] = (0x70 & config << 4) | 0x82;
   }
   if (writeADC)
   {
   //toggle the chip select low on the ADCs to allow an SPI load of 16 bits
    digitalOut = digitalOut & 0xff7fff; WriteDigOutputs();

   //Write 2 bytes out the SPI output to the ADCs on the falling edge of sclk
    fpgaLink->write(fpgaLink, buf, 2);

   //toggle the chip select high on the ADCs to prohibit SPI loads to the ADCs
    digitalOut = digitalOut | 0x008000; WriteDigOutputs();
   }
   else
   {
   //Write 3 bytes out the SPI output to the PLL on the rising edge of sclk
    SPILink_reconfig((SPILink *)fpgaLink, CLK_1MHz, NORM_LOW_LATCH_RISE);
    fpgaLink->write(fpgaLink, buf, 3);

   //toggle the PLL Load Enable pin to load the internal initialization register
    digitalOut = digitalOut | 0x010000;  WriteDigOutputs();

   //Toggle the PLL Load Enable pin to prepare for the next register load
    digitalOut = digitalOut & 0xfeffff;  WriteDigOutputs();
   }
   writeInit = writeRCount = writeNCount = writeConfig = writeSPIData = false;
   writeADC = false;
  }
//as long as the FPGA is not being read for I/Q data, transmit data on the data bus to set the digital
 //attenuators,  the DAC source ADC and the filter factor value
 if (!FPGA_Cmd_Read)
{
//send new digital attenuator values
  if (newAtten AND !getting_IQ)  {
   attenBits = (ulong)setAtten - 5;
//set up the data on the bus
   attenBits = ((attenBits << 14) & 0x300000) | ((attenBits << 8) & 0x003f00);
   digitalOut = ((digitalOut & 0xcfc0f0) | attenBits) | 0x000000;
   WriteDigOutputs();
//toggle handshaking bit
   digitalOut = digitalOut | 0x080000; WriteDigOutputs();
   FPGA_Cmd_Read = true;
   newAtten = false;
  }
//change the DAC source ADC
  else if (changeADC AND !getting_IQ)
  {
//set up the data on the data bus
   ADCValLong = ((ulong)ADCVal << 8) & 0x003f00;
   digitalOut = ((digitalOut & 0xcfc0f1) | ADCValLong) | 0x000001;
   WriteDigOutputs();
//toggle the handshaking bit
   digitalOut = digitalOut | 0x080000; WriteDigOutputs();
   FPGA_Cmd_Read = true;
   changeADC = false;
  }
//change the filter factor value
  else if (changeFilter AND !getting_IQ)
  {
//set up the first byte of data on the data bus
   filterValLong = (((ulong)filterVal << 12) & 0x300000) |
```

```
    (((ulong)filterVal << 6) & 0x003f00);
   digitalOut = ((digitalOut & 0xcfc0f0) | filterValLong) | 0x000006;
   WriteDigOutputs();
//toggle the handshaking bit
   digitalOut = digitalOut | 0x080000; WriteDigOutputs();
   changeFilter = false; changeFilter2 = true;
  }
//set up the last 2 bits of data on the data bus
  else if (changeFilter2 AND !getting_IQ)
  {
   filterValLong = ((ulong)filterVal << 8) & 0x000300;
   digitalOut = ((digitalOut & 0xcfc0f0) | filterValLong) | 0x000007;
   WriteDigOutputs();
//toggle the handshaking bit
   digitalOut = digitalOut | 0x080000; WriteDigOutputs();
   changeFilter2 = false;
   filtered = true;
  }
//gather the I/Q values from the FPGA
  else
  {
   getting_IQ = true;
   switch (get_IQ)
   {
    case 0:  //Give the command to send out the I value from the FPGA
      digitalOut = (digitalOut & 0xffc0f0) | 0x000002; WriteDigOutputs();
      digitalOut = digitalOut | 0x080000; WriteDigOutputs();
      FPGA_Cmd_Read = true; get_IQ++; break;
    case 1:  //Read the high byte of I from the FPGA
      ReadDigInputs();
      IA_Val_Read = (int)(digitalIn & 0x00ff) << 8;
     //Give the command to send out the I value from the FPGA
      digitalOut = (digitalOut & 0xffc0f0) | 0x000102; WriteDigOutputs();
      digitalOut = digitalOut | 0x080000; WriteDigOutputs();
      FPGA_Cmd_Read = true; get_IQ++; break;
    case 2:  //Read the low byte of I from the FPGa
      ReadDigInputs();
      ReadIA = (IA_Val_Read | (int)(digitalIn & 0x00ff)) & 0x3fff;
     //Give the command to send out the Q value from the FPGA
      digitalOut = (digitalOut & 0xffc0f0) | 0x000003; WriteDigOutputs();
      digitalOut = digitalOut | 0x080000; WriteDigOutputs();
      FPGA_Cmd_Read = true; get_IQ++; break;
    case 3:  //Read the high byte of Q from the FPGA
      ReadDigInputs();
      QA_Val_Read = (int)(digitalIn & 0x00ff) << 8;
  //Give the command to send out the Q value from the FPGA
      digitalOut = (digitalOut & 0xffc0f0) | 0x000103; WriteDigOutputs();
      digitalOut = digitalOut | 0x080000; WriteDigOutputs();
      FPGA_Cmd_Read = true; get_IQ++; break;
    case 4:  //Read the low byte of Q from the FPGA
      ReadDigInputs();
      ReadQA = QA_Val_Read | (int)(digitalIn & 0x00ff);
  //Give the command to send out the -I value from the FPGA
      digitalOut = (digitalOut & 0xffc0f0) | 0x000004; WriteDigOutputs();
      digitalOut = digitalOut | 0x080000; WriteDigOutputs();
      FPGA_Cmd_Read = true; get_IQ++; break;
    case 5:  //Read the high byte of -I from the FPGA
      ReadDigInputs();
      ImA_Val_Read = (int)(digitalIn & 0x00ff) << 8;
  //Give the command to send out the -I value from the FPGA
      digitalOut = (digitalOut & 0xffc0f0) | 0x000104; WriteDigOutputs();
      digitalOut = digitalOut | 0x080000; WriteDigOutputs();
```

```
      FPGA_Cmd_Read = true; get_IQ++; break;
    case 6: //Read the low bye of -I from the FPGA
      ReadDigInputs();
      ReadImA = (ImA_Val_Read | (int)(digitalIn & 0xff)) & 0x3fff;
//Give the command to send out the -Q value from the FPGA
      digitalOut = (digitalOut & 0xffc0f0) | 0x000005; WriteDigOutputs();
      digitalOut = digitalOut | 0x080000; WriteDigOutputs();
      FPGA_Cmd_Read = true; get_IQ++; break;
    case 7: //Read the high byte of -Q from the FPGA
      ReadDigInputs();
      QmA_Val_Read = (int)(digitalIn & 0xff) << 8;
 //Give the command to send out the -Q value from the FPGA
      digitalOut = (digitalOut & 0xffc0f0) | 0x000105; WriteDigOutputs();
      digitalOut = digitalOut | 0x080000; WriteDigOutputs();
      FPGA_Cmd_Read = true; get_IQ++; break;
    case 8: //Read the low byte of -Q from the FPGA and reset the sequence
      ReadDigInputs();
      ReadQmA = QmA_Val_Read | (int)(digitalIn & 0xff);
//Give the command to send out the I value from the FPGA
      digitalOut = (digitalOut & 0xffc0f0) | 0x000202; WriteDigOutputs();
      digitalOut = digitalOut | 0x080000; WriteDigOutputs();
      FPGA_Cmd_Read = true; get_IQ++; break;
    case 9:  //Read the high byte of I from the FPGA
      ReadDigInputs();
      IB_Val_Read = (int)(digitalIn & 0x00ff) << 8;
  //Give the command to send out the I value from the FPGA
      digitalOut = (digitalOut & 0xffc0f0) | 0x000302; WriteDigOutputs();
      digitalOut = digitalOut | 0x080000; WriteDigOutputs();
      FPGA_Cmd_Read = true; get_IQ++; break;
    case 10:  //Read the low byte of I from the FPGa
      ReadDigInputs();
      ReadIB = (IB_Val_Read | (int)(digitalIn & 0x00ff)) & 0x3fff;
   //Give the command to send out the Q value from the FPGA
      digitalOut = (digitalOut & 0xffc0f0) | 0x000203; WriteDigOutputs();
      digitalOut = digitalOut | 0x080000; WriteDigOutputs();
      FPGA_Cmd_Read = true; get_IQ++; break;
    case 11:  //Read the high byte of Q from the FPGA
      ReadDigInputs();
      QB_Val_Read = (int)(digitalIn & 0x00ff) << 8;
//Give the command to send out the Q value from the FPGA
      digitalOut = (digitalOut & 0xffc0f0) | 0x000303; WriteDigOutputs();
      digitalOut = digitalOut | 0x080000; WriteDigOutputs();
      FPGA_Cmd_Read = true; get_IQ++; break;
    case 12:  //Read the low byte of Q from the FPGA
      ReadDigInputs();
      ReadQB = QB_Val_Read | (int)(digitalIn & 0x00ff);
//Give the command to send out the -I value from the FPGA
      digitalOut = (digitalOut & 0xffc0f0) | 0x000204; WriteDigOutputs();
      digitalOut = digitalOut | 0x080000; WriteDigOutputs();
      FPGA_Cmd_Read = true; get_IQ++; break;
    case 13:  //Read the high byte of -I from the FPGA
      ReadDigInputs();
      ImB_Val_Read = (int)(digitalIn & 0x00ff) << 8;
 //Give the command to send out the -I value from the FPGA
      digitalOut = (digitalOut & 0xffc0f0) | 0x000304; WriteDigOutputs();
      digitalOut = digitalOut | 0x080000; WriteDigOutputs();
      FPGA_Cmd_Read = true; get_IQ++; break;
    case 14: //Read the low bye of -I from the FPGA
      ReadDigInputs();
      ReadImB = (ImB_Val_Read | (int)(digitalIn & 0xff)) & 0x3fff;
       //Give the command to send out the -Q value from the FPGA
      digitalOut = (digitalOut & 0xffc0f0) | 0x000205; WriteDigOutputs();
```

```
  digitalOut = digitalOut | 0x080000; WriteDigOutputs();
  FPGA_Cmd_Read = true; get_IQ++; break;
case 15: //Read the high byte of -Q from the FPGA
  ReadDigInputs();
  QmB_Val_Read = (int)(digitalIn & 0xff) << 8;
  //Give the command to send out the -Q value from the FPGA
  digitalOut = (digitalOut & 0xffc0f0) | 0x000305; WriteDigOutputs();
  digitalOut = digitalOut | 0x080000; WriteDigOutputs();
  FPGA_Cmd_Read = true; get_IQ++; break;
case 16: //Read the low byte of -Q from the FPGA and reset the sequence
  ReadDigInputs();
  ReadQmB = QmB_Val_Read | (int)(digitalIn & 0xff);
   //Give the command to send out the I value from the FPGA
  digitalOut = (digitalOut & 0xffc0f0) | 0x000402; WriteDigOutputs();
  digitalOut = digitalOut | 0x080000; WriteDigOutputs();
  FPGA_Cmd_Read = true; get_IQ++; break;
case 17:  //Read the high byte of I from the FPGA
  ReadDigInputs();
  IC_Val_Read = (int)(digitalIn & 0x00ff) << 8;
    //Give the command to send out the I value from the FPGA
  digitalOut = (digitalOut & 0xffc0f0) | 0x000502; WriteDigOutputs();
  digitalOut = digitalOut | 0x080000; WriteDigOutputs();
  FPGA_Cmd_Read = true; get_IQ++; break;
case 18:  //Read the low byte of I from the FPGa
  ReadDigInputs();
  ReadIC = IC_Val_Read | (int)(digitalIn & 0x00ff);
  //Give the command to send out the Q value from the FPGA
  digitalOut = (digitalOut & 0xffc0f0) | 0x000403; WriteDigOutputs();
  digitalOut = digitalOut | 0x080000; WriteDigOutputs();
  FPGA_Cmd_Read = true; get_IQ++; break;
case 19:  //Read the high byte of Q from the FPGA
  ReadDigInputs();
  QC_Val_Read = (int)(digitalIn & 0x00ff) << 8;
    //Give the command to send out the Q value from the FPGA
  digitalOut = (digitalOut & 0xffc0f0) | 0x000503; WriteDigOutputs();
  digitalOut = digitalOut | 0x080000; WriteDigOutputs();
  FPGA_Cmd_Read = true; get_IQ++; break;
case 20:  //Read the low byte of Q from the FPGA
  ReadDigInputs();
  ReadQC = QC_Val_Read | (int)(digitalIn & 0x00ff);
   //Give the command to send out the -I value from the FPGA
  digitalOut = (digitalOut & 0xffc0f0) | 0x000404; WriteDigOutputs();
  digitalOut = digitalOut | 0x080000; WriteDigOutputs();
  FPGA_Cmd_Read = true; get_IQ++; break;
case 21:  //Read the high byte of -I from the FPGA
  ReadDigInputs();
  ImC_Val_Read = (int)(digitalIn & 0x00ff) << 8;
  //Give the command to send out the -I value from the FPGA
  digitalOut = (digitalOut & 0xffc0f0) | 0x000504; WriteDigOutputs();
  digitalOut = digitalOut | 0x080000; WriteDigOutputs();
  FPGA_Cmd_Read = true; get_IQ++; break;
case 22: //Read the low bye of -I from the FPGA
  ReadDigInputs();
  ReadImC = ImC_Val_Read | (int)(digitalIn & 0xff);
   //Give the command to send out the -Q value from the FPGA
  digitalOut = (digitalOut & 0xffc0f0) | 0x000405; WriteDigOutputs();
  digitalOut = digitalOut | 0x080000; WriteDigOutputs();
  FPGA_Cmd_Read = true; get_IQ++; break;
case 23: //Read the high byte of -Q from the FPGA
  ReadDigInputs();
  QmC_Val_Read = (int)(digitalIn & 0xff) << 8;
   //Give the command to send out the -Q value from the FPGA
```

```
  digitalOut = (digitalOut & 0xffc0f0) | 0x000505; WriteDigOutputs();
  digitalOut = digitalOut | 0x080000; WriteDigOutputs();
  FPGA_Cmd_Read = true; get_IQ++; break;
case 24: //Read the low byte of -Q from the FPGA and reset the sequence
  ReadDigInputs();
  ReadQmC = QmC_Val_Read | (int)(digitalIn & 0xff);
   //Give the command to send out the I value from the FPGA
  digitalOut = (digitalOut & 0xffc0f0) | 0x000802; WriteDigOutputs();
  digitalOut = digitalOut | 0x080000; WriteDigOutputs();
  FPGA_Cmd_Read = true; get_IQ++; break;
case 25:  //Read the high byte of I from the FPGA
  ReadDigInputs();
  IRef_Val_Read = (int)(digitalIn & 0x00ff) << 8;
   //Give the command to send out the I value from the FPGA
  digitalOut = (digitalOut & 0xffc0f0) | 0x000902; WriteDigOutputs();
  digitalOut = digitalOut | 0x080000; WriteDigOutputs();
  FPGA_Cmd_Read = true; get_IQ++; break;
case 26:  //Read the low byte of I from the FPGa
  ReadDigInputs();
  ReadIRef = IRef_Val_Read | (int)(digitalIn & 0x00ff);
    //Give the command to send out the Q value from the FPGA
  digitalOut = (digitalOut & 0xffc0f0) | 0x000803; WriteDigOutputs();
  digitalOut = digitalOut | 0x080000; WriteDigOutputs();
  FPGA_Cmd_Read = true; get_IQ++; break;
case 27:  //Read the high byte of Q from the FPGA
  ReadDigInputs();
  QRef_Val_Read = (int)(digitalIn & 0x00ff) << 8;
    //Give the command to send out the Q value from the FPGA
  digitalOut = (digitalOut & 0xffc0f0) | 0x000903; WriteDigOutputs();
  digitalOut = digitalOut | 0x080000; WriteDigOutputs();
  FPGA_Cmd_Read = true; get_IQ++; break;
case 28:  //Read the low byte of Q from the FPGA
  ReadDigInputs();
  ReadQRef = QRef_Val_Read | (int)(digitalIn & 0x00ff);
   //Give the command to send out the -I value from the FPGA
  digitalOut = (digitalOut & 0xffc0f0) | 0x000804; WriteDigOutputs();
  digitalOut = digitalOut | 0x080000; WriteDigOutputs();
  FPGA_Cmd_Read = true; get_IQ++; break;
case 29:  //Read the high byte of -I from the FPGA
  ReadDigInputs();
  ImRef_Val_Read = (int)(digitalIn & 0x00ff) << 8;
   //Give the command to send out the -I value from the FPGA
  digitalOut = (digitalOut & 0xffc0f0) | 0x000904; WriteDigOutputs();
  digitalOut = digitalOut | 0x080000; WriteDigOutputs();
  FPGA_Cmd_Read = true; get_IQ++; break;
case 30: //Read the low bye of -I from the FPGA
  ReadDigInputs();
  ReadImRef = ImRef_Val_Read | (int)(digitalIn & 0xff);
   //Give the command to send out the -Q value from the FPGA
  digitalOut = (digitalOut & 0xffc0f0) | 0x000805; WriteDigOutputs();
  digitalOut = digitalOut | 0x080000; WriteDigOutputs();
  FPGA_Cmd_Read = true; get_IQ++; break;
case 31: //Read the high byte of -Q from the FPGA
  ReadDigInputs();
  QmRef_Val_Read = (int)(digitalIn & 0xff) << 8;
   //Give the command to send out the -Q value from the FPGA
  digitalOut = (digitalOut & 0xffc0f0) | 0x000905; WriteDigOutputs();
  digitalOut = digitalOut | 0x080000; WriteDigOutputs();
  FPGA_Cmd_Read = true; get_IQ++; break;
case 32: //Read the low byte of -Q from the FPGA and reset the sequence
  ReadDigInputs();
  ReadQmRef = QmRef_Val_Read | (int)(digitalIn & 0xff);
```

```
      digitalOut = (digitalOut & 0xffc0f0) | 0x000008; WriteDigOutputs();
      digitalOut = digitalOut | 0x080000; WriteDigOutputs();
      FPGA_Cmd_Read = true; get_IQ = 0; getting_IQ = false; New_IQ = true;
      break;

    default: //Reset to read I value
      get_IQ = 0; break;
   }
  }
}
//wait for the FPGA to respond that it has finished processing a command
else if ( digitalIn & 0x1000 )
{
   FPGA_Cmd_Read = false;
   digitalOut = digitalOut & 0xf7ffff; WriteDigOutputs();
}

//--- blink activity LED @ 4 Hz if calibrating, 8 Hz for normal operation ---
 if (calibrating)  mask = 0x0080;  else  mask = 0x0040;
 if (flashBad) mask = 0x0200;  // reduce to 1 Hz if a hardware problem
 if (TICK_TIMER & mask)
   ClearBits(digitalOut, ACTIVITY_LED);
 else
   SetBits(digitalOut, ACTIVITY_LED);
}
```

APPENDIX C

Digital I/O usage for the ZWorld

Table C.1. Telnet Interface Description.

Pin Name	Description
DO00-DO03	Command Out
DO04-DO6	Extra Output
DO07	Activity LED
DO08-DO09, DO20, DO21	Data Bus Out
DO10-DO13	Extra Output
DO14	FPGA Program Pin Connection
DO15	Fast ADC Enable
DO16	PLL Enable
DO17	PROM Enable
DO18	Extra
DO19	Handshaking Out
DO22	Slow DAC Enable
DO23	Slow ADC Enable
DI00-DI07	Data Bus In
DI08-DI11	Extra Input

APPENDIX D

Phase Meter Schematics

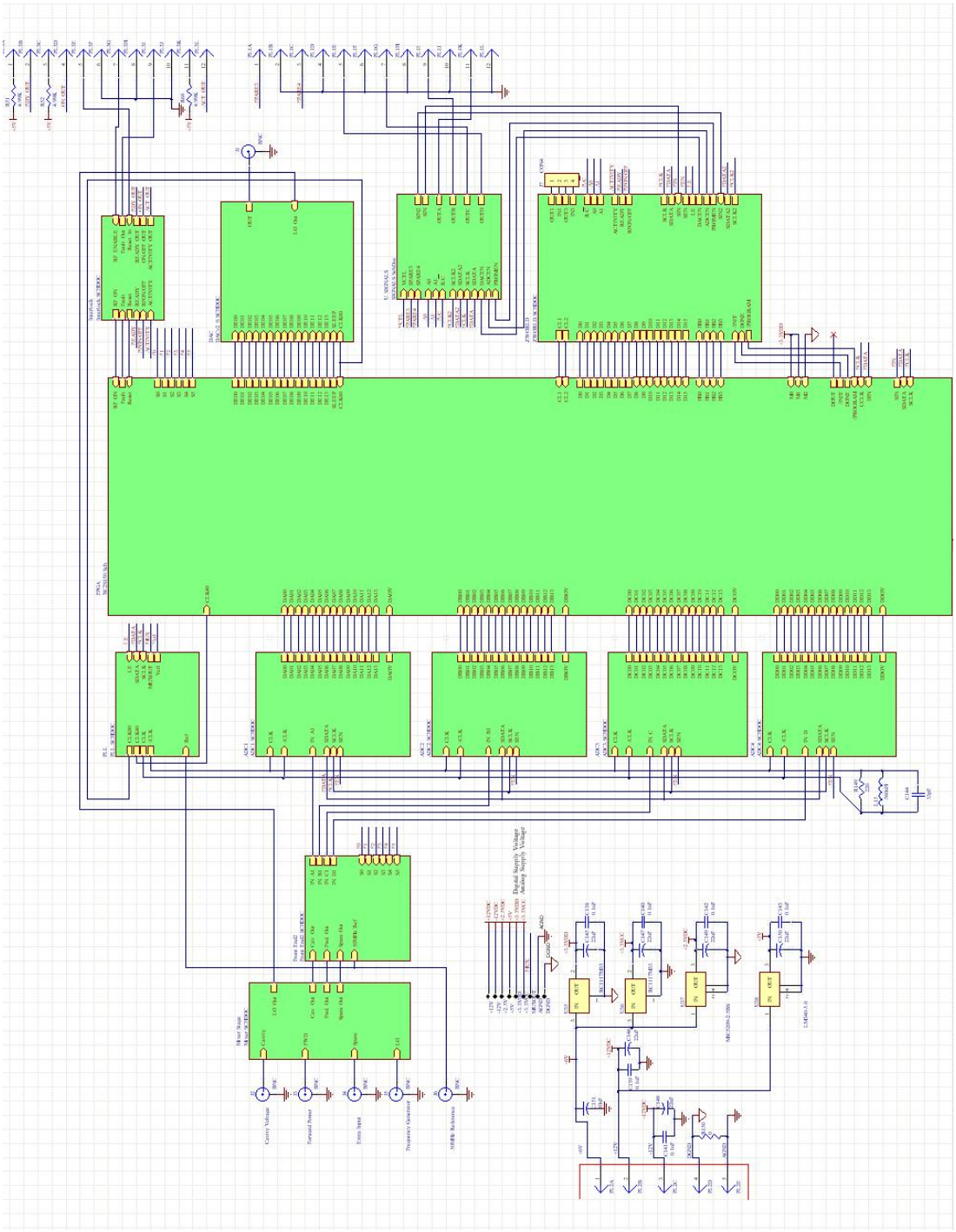

Figure D.1. System Overview

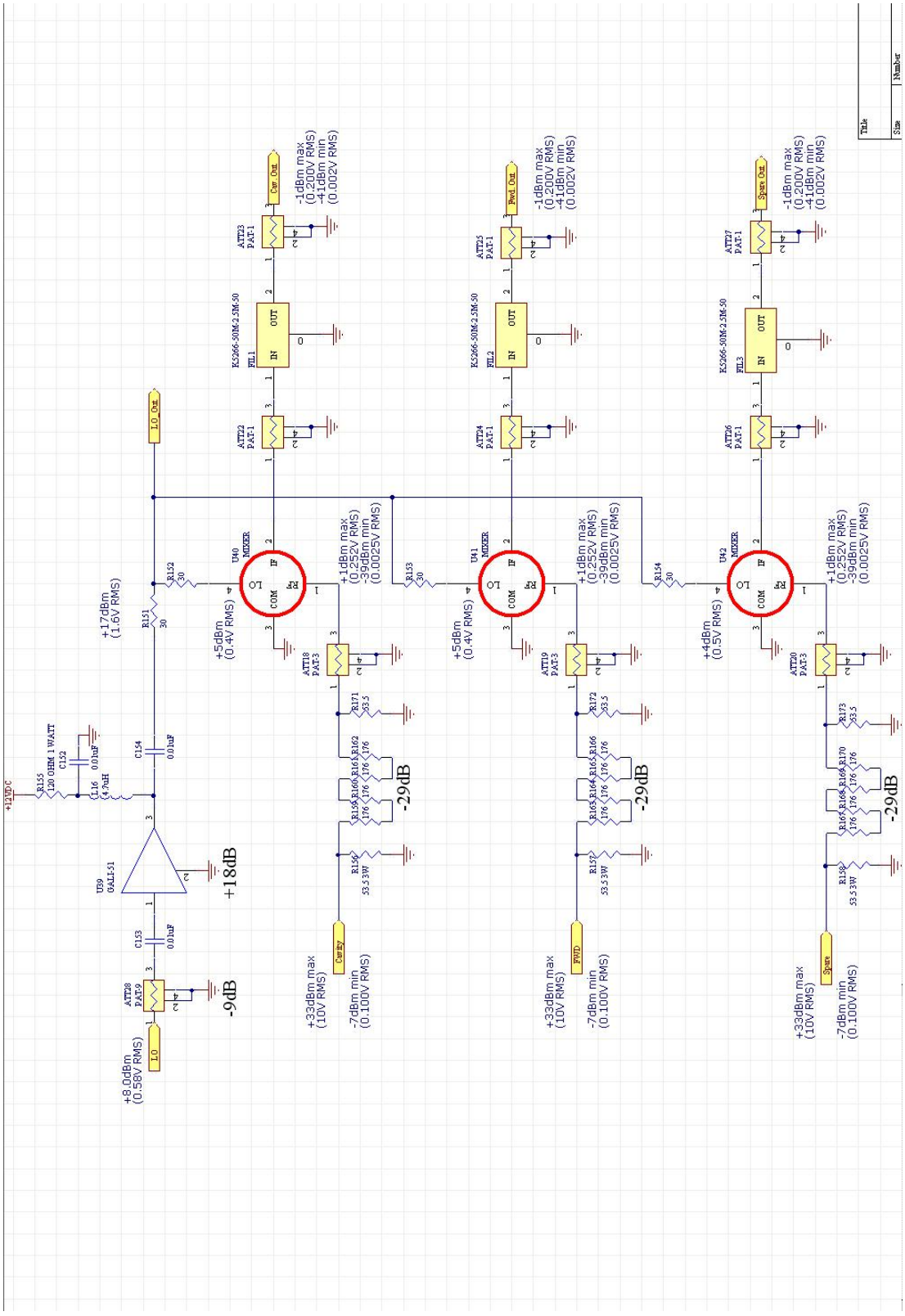

Figure D.2. Mixer Stage

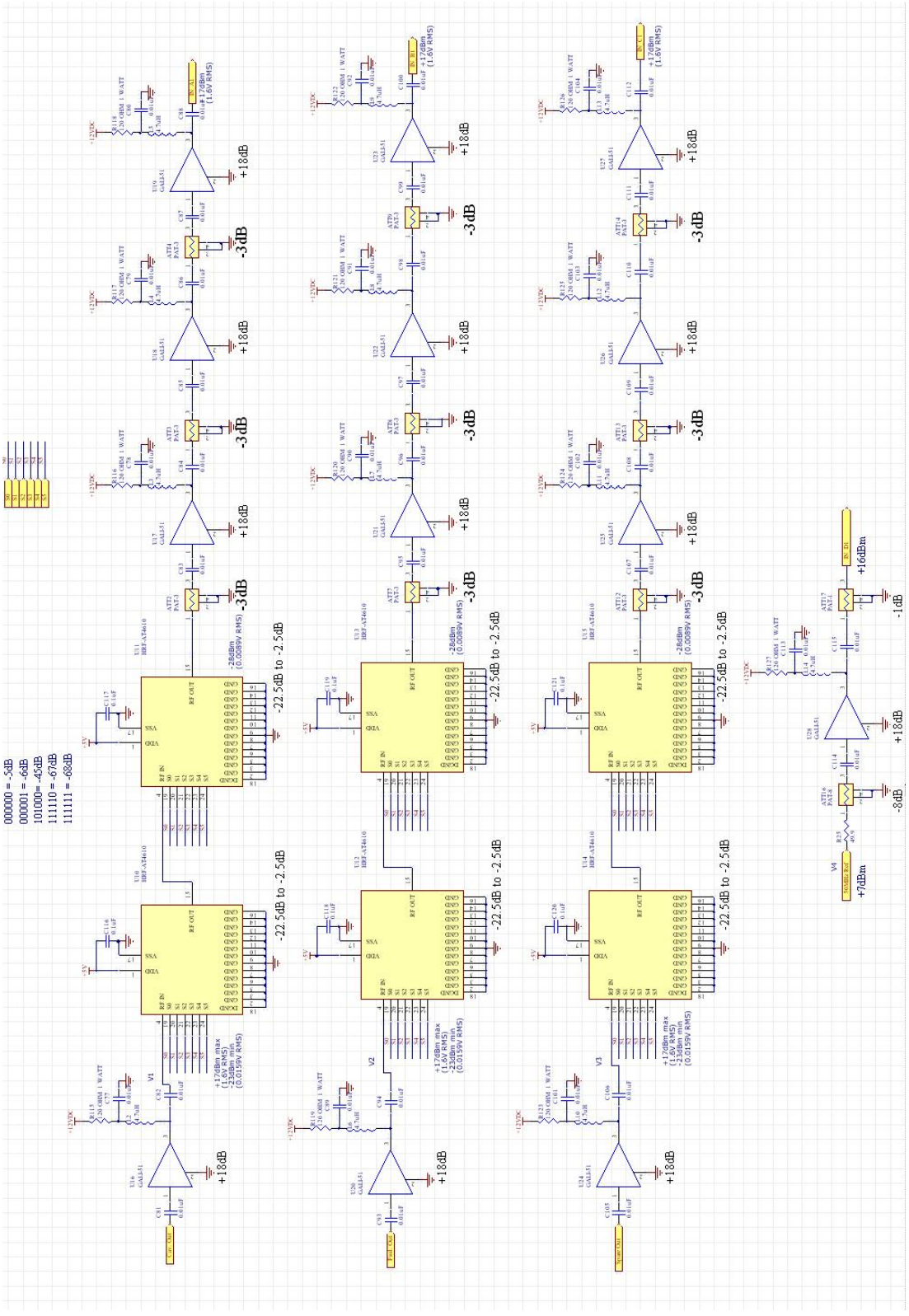

Figure D.3. Signal Conditioning (CH 1 and LO)

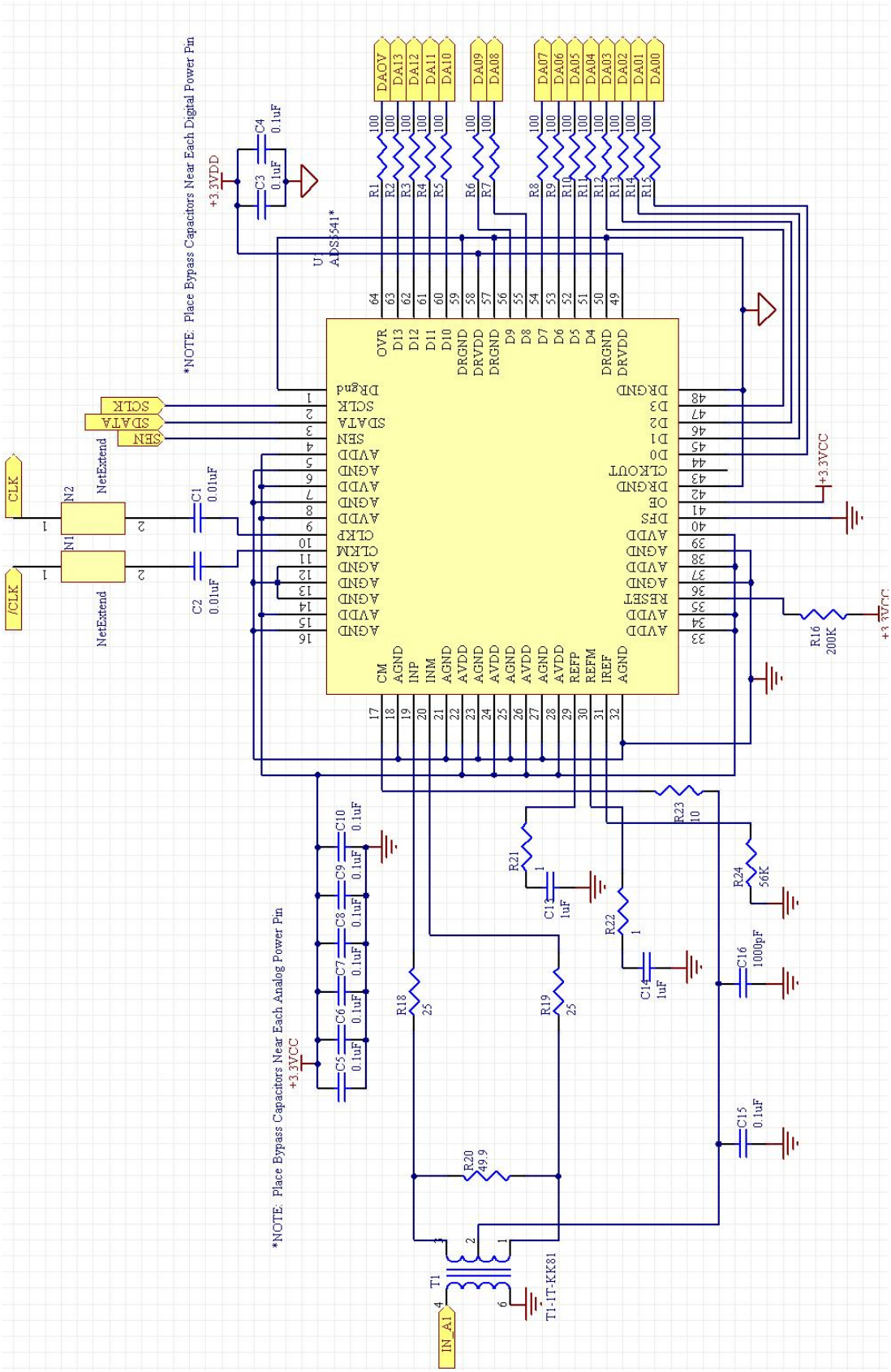

Figure D.4. Analog to Digital Converters (CH 1)

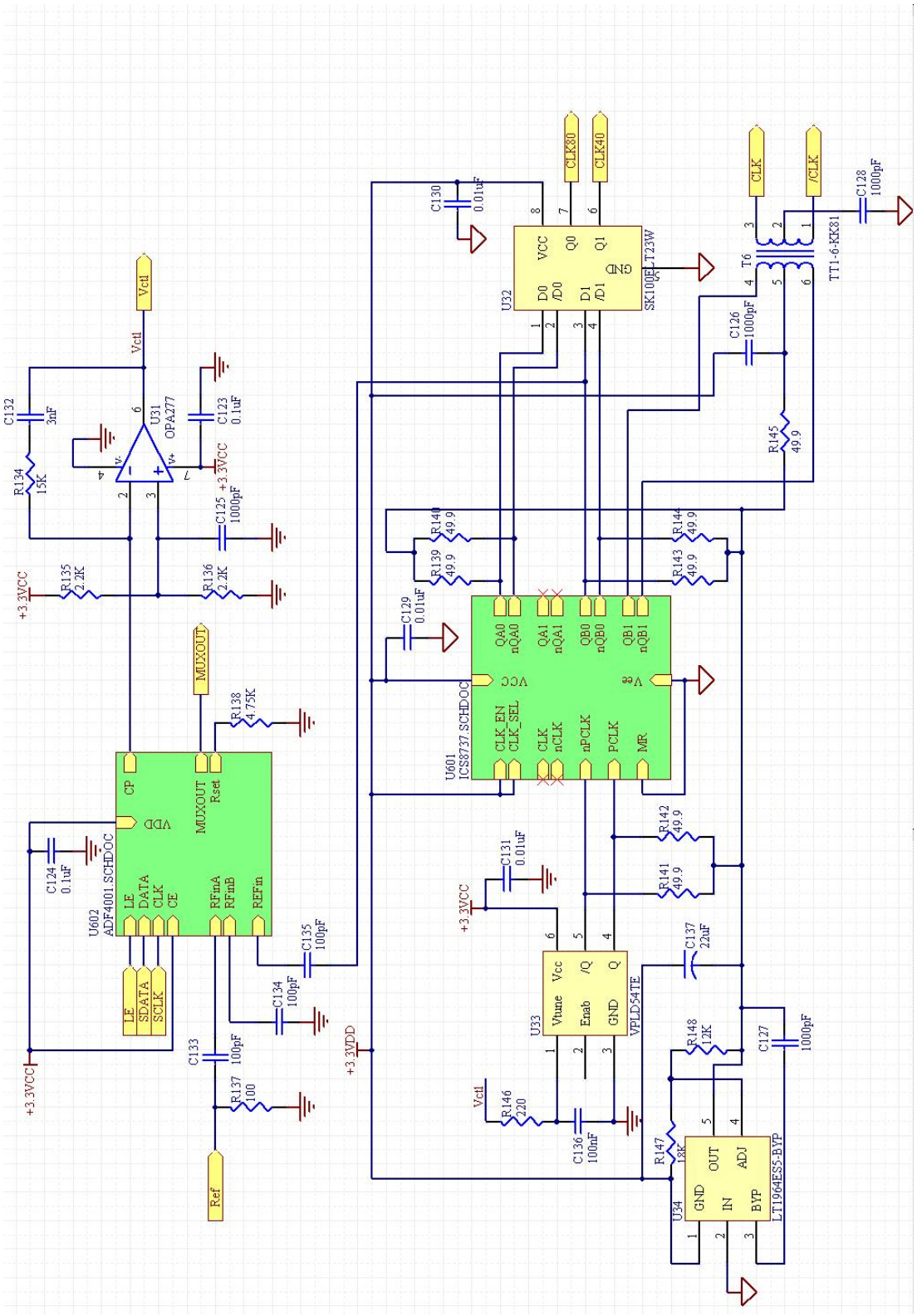

Figure D.5. Phase Lock Loop

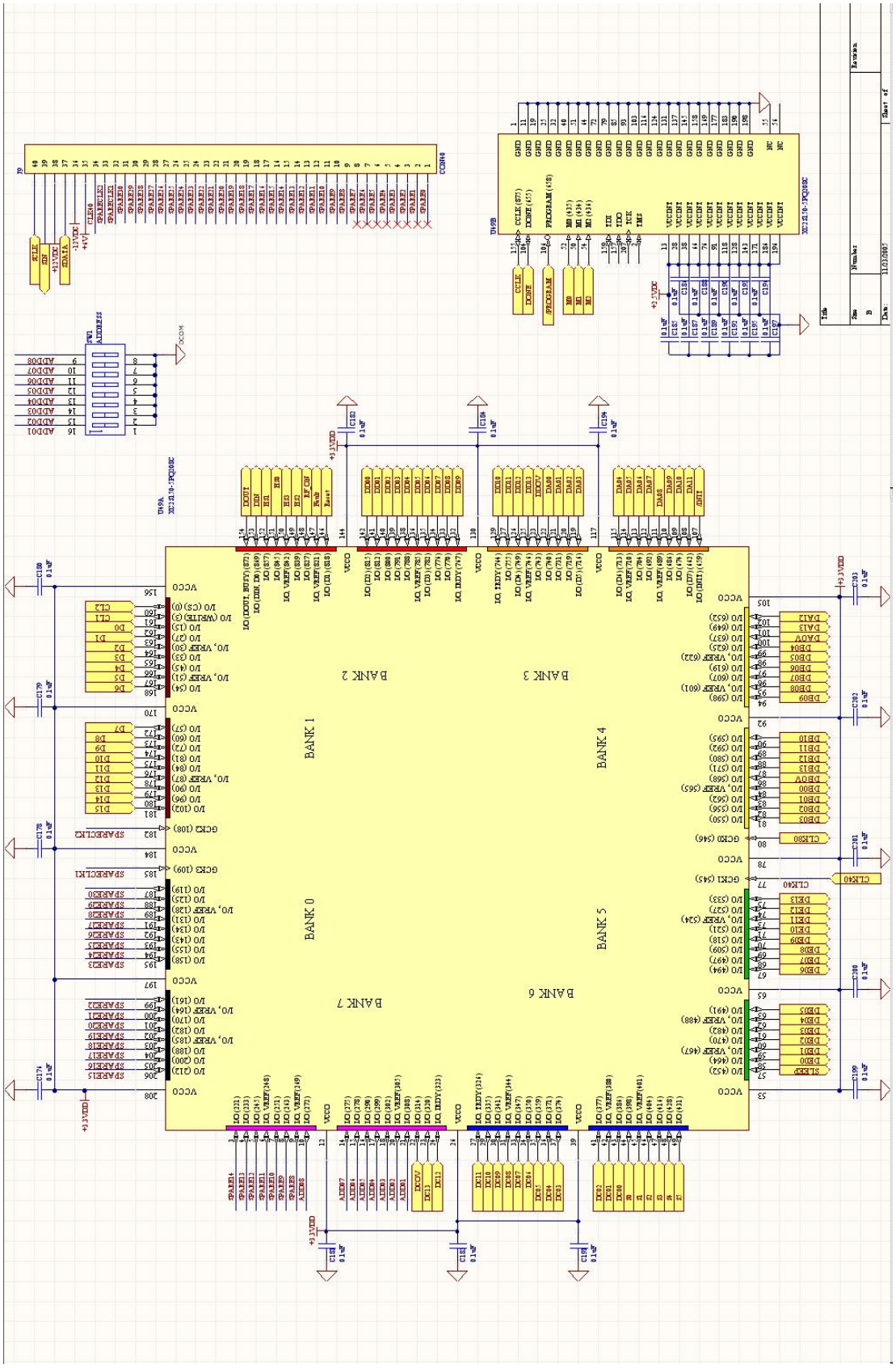

Figure D.6. Xilinx XC2S150 FPGA

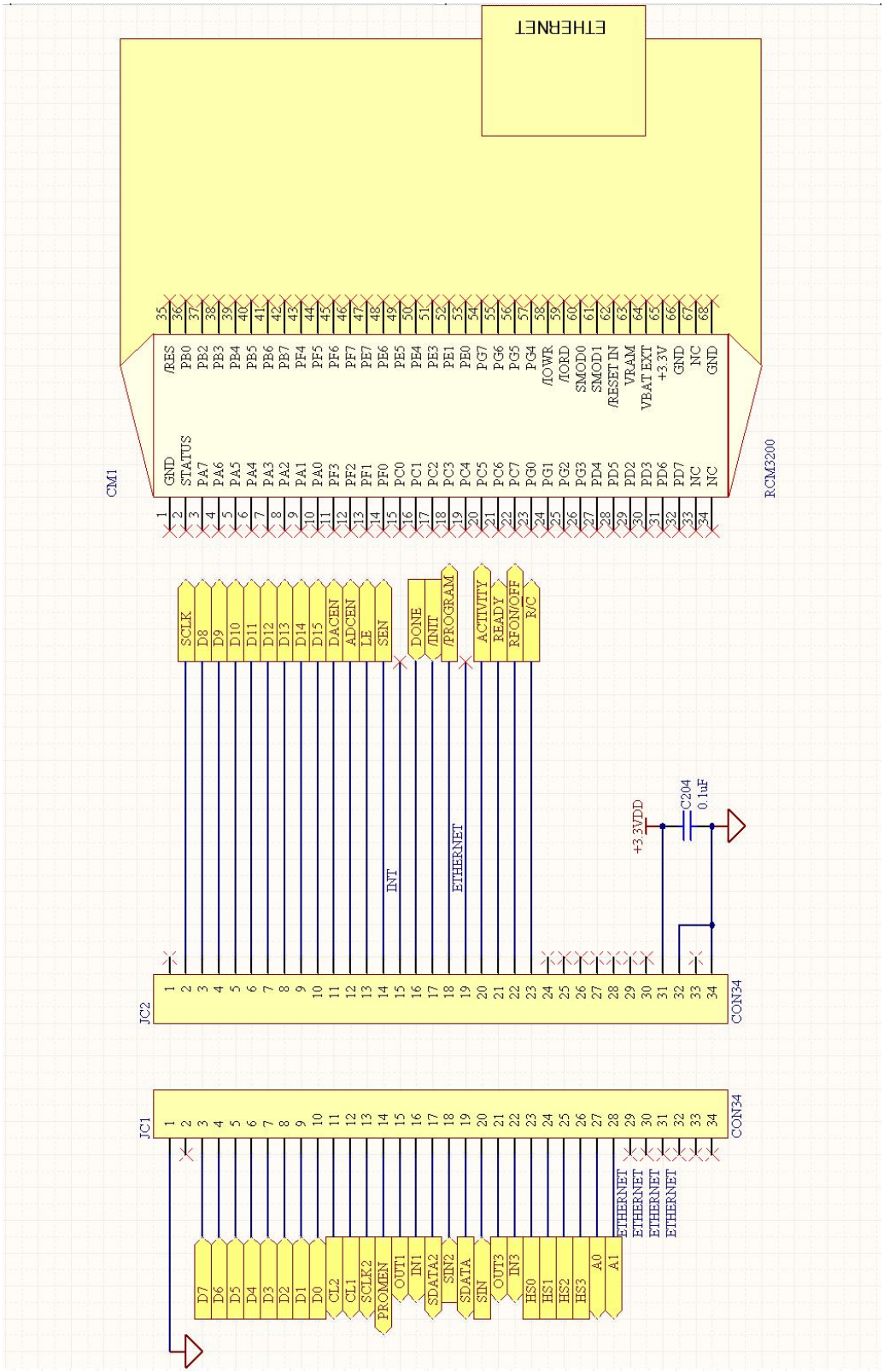

Figure D.7. ZWorld Microcomputer

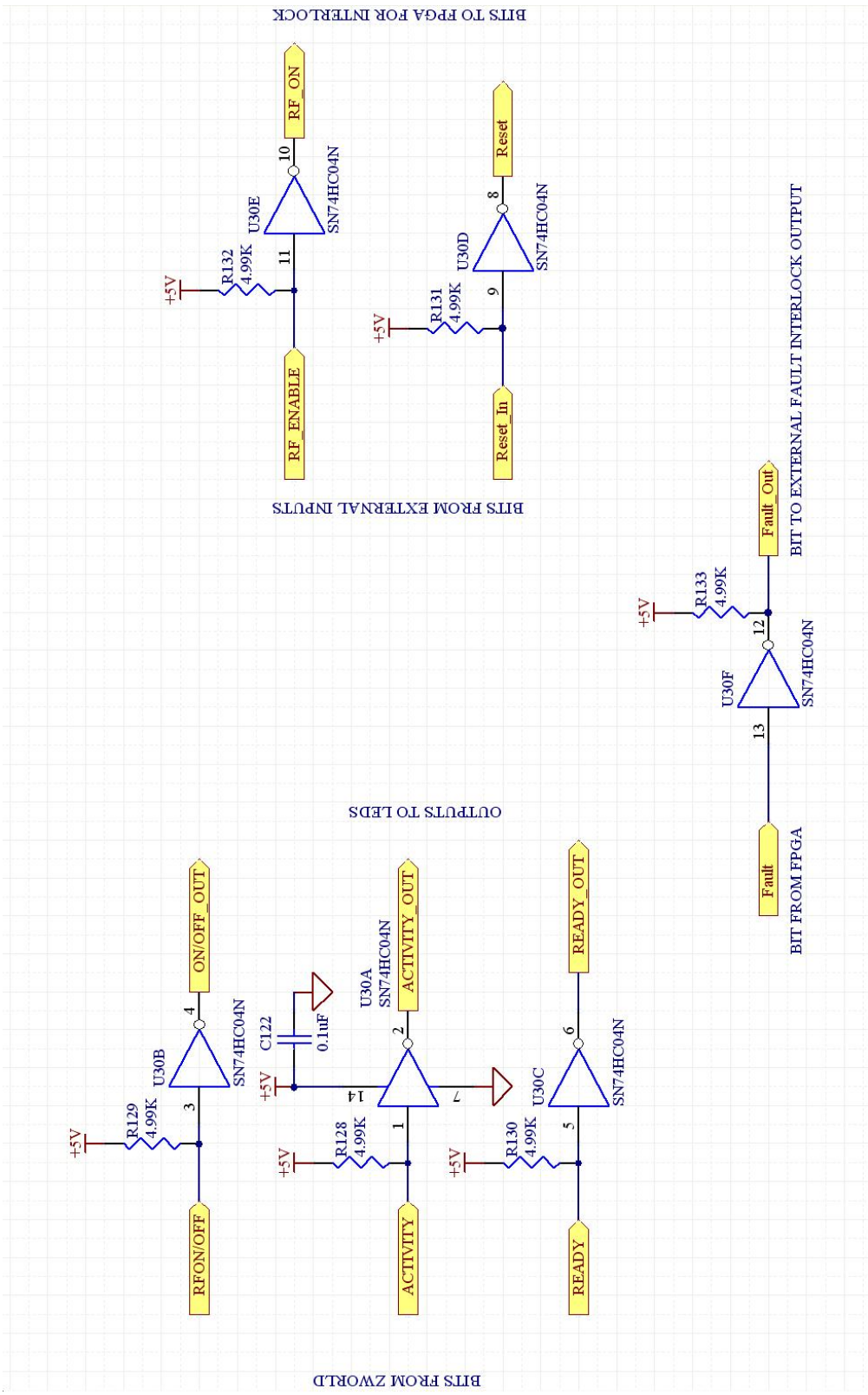

Figure D.8. Interlocks

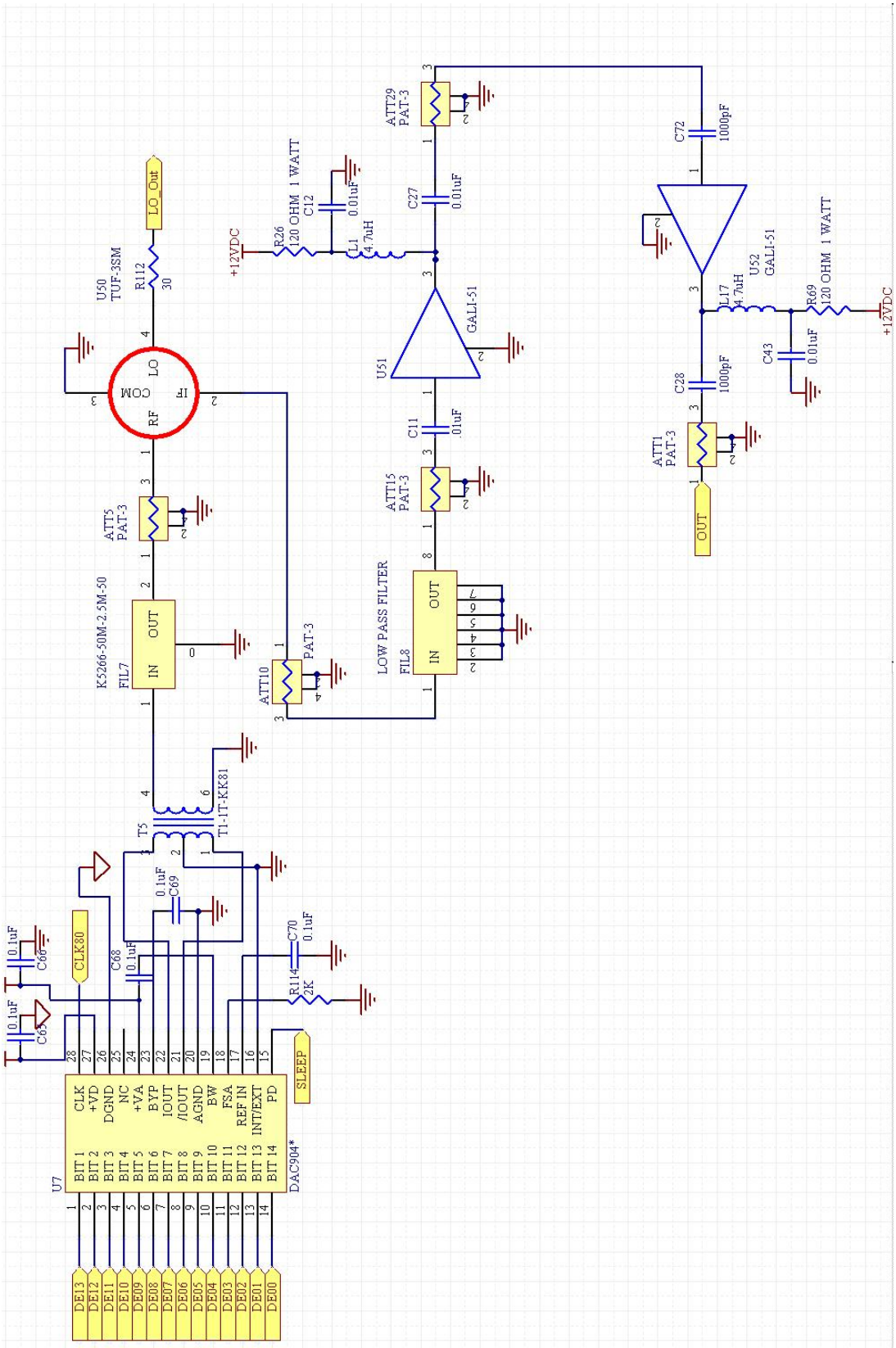

Figure D.9. DAC Output

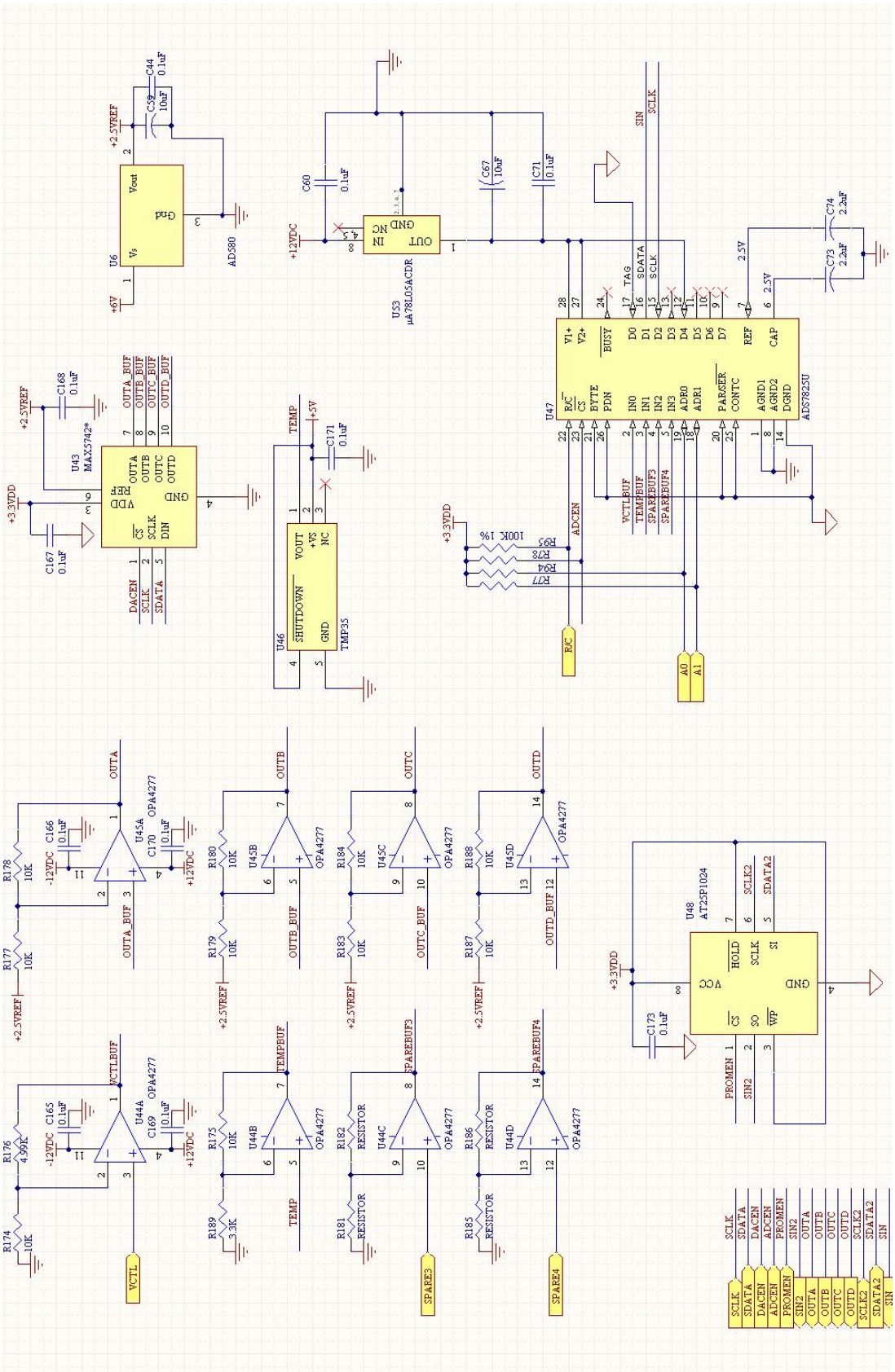

Figure D.10. Housekeeping Circuitry

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] J. Vincent, L. Foth, A. McGilvra, J. Priller. *The NSCL Control System.* Michigan State University, East Lansing, Michigan, 1995.

[2] T. Berenc, J. Brandon, J. Vincent. *CCP Phase Regulation System.* R.F. Note #121, Michigan State University, East Lansing, Michigan, September 1997

[3] A. Fox. *Ask the Applications Engineer-30* Analog Dialogue 36 – 03, www.analog.com/UploadedFiles/Application_Notes/90600605APP_NOTE_FOX.pdf, 2002.

[4] L. R. Doolittle *Plan for a 50MHz Analog Output Channel.* LBNL, Berkeley California. August 2002.

[5] W. Zabolotny, K. Pozniak, R. Romaniuk, T. Czarski, I. Kudla, K. Kierzkowski, T. Jezynski, A. Burghardt, S. Simrock. *Design and Simulation of FPGA Implementation of RF Control System for Tesla Test Facility.* Tesla Report 2003-05, Hamburg, Germany, 2003.

[6] C.Ziomek, P. Corredoura. *Digital I/Q Demodulator.* Stanford Linear Accelerator Center, Stanford, California, 1995.

[7] M. J. O'Farrell. *Low Level Radio Frequency Control of RIA Superconducting Cavities.* Masters Thesis, Michigan State University, East Lansing, Michigan, 2005.

[8] http://www.odyseus.nildram.co.uk/RFMicrowave_Circuits_Files/Attenuator.pdf

[9] B. C. Henderson. *Mixers: Part 1 Characteristics and Performance* WJ Tech Notes vol. 8 #2, Watkins-Johnson Company, 1981.

[10] F. A. Losee. *RF Systems, Components, and Circuits.* Artech House, Inc, Norwood, Massachussets, 2005.

[11] R.K. Feeney, D.R. Hertling. *RF Circuit Design: Active Circuits* Georgiea Institue of Technology Department of Continuing Education, Atlanta, Georgia, January, 1995.

[12] R. E. Best. *Phase-Locked Loops.* The McGraw-Hill Companies, Inc, New York, New York, 2003.

[13] Analog Devices. *200MHz Clock Generator PLL* Device Data Sheet, www.analog.com, 2003.

[14] Maxim Semiconductor. *HFAN-01.0: Introduction to LVDS, PECL, and CML.* Application Note 291, http://www.maxim-ic.com/appnotes.cfm/appnote_number/291, October, 2000.

[15] A. Ambardar. *Analog and Digital Signal Processing.* Brooks/Cole, Pacific Grove, California, 2nd edition, 1999.

[16] Texas Instruments. *14-Bit, 80MSPS Analog-to-Digital Converter.* Device Data Sheet, Burr-Brown Products, March 2005.

[17] Xilinx, Inc. *The Low-Cost, Efficient Serial Configuration of Spartan FPGAs.* XAPP098, Xilinx, Inc, November, 1998.

[18] Xilinx, Inc. *Xilinx In-System Programming Using an Embedded Microcontroller.* XAPP058, Xilinx, Inc, June, 2004.

[19] Xilinx, Inc. *Configuration and Readback of the Spartan-II and Spartan-IIE Families.* XAPP176, Xilinx, Inc, March, 2002.

[20] Hewlett Packard. *HP 8508A Vector Voltmeter Operating and Service Manual.* Service Manual, Hewlett Packard, West Lothian, Scotland, May, 1988.