Q-ENHANCED TUNABLE FILTER DESIGN WITH APPLICATIONS IN RECEIVER ARCHITECTURES

by

CHELSI KOVALA

B.S., Kansas State University, 2009
B.S., Kansas State University, 2009

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Electrical Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2012

Approved by:
Major Professor
William Kuhn

Abstract

Q-enhanced Filters have been researched extensively, but have not been often implemented into receiver architectures due to inherent challenges in the design and stability of these filters. However, recent works have successfully addressed Q-enhanced filter designs which are viable for receiver implementation with tuning algorithms to achieve temperature stability. This work continues these efforts with the redesign of a Two-Pole Q-Enhanced Band-Pass filter tested at narrower fractional bandwidths than previous work of less than one percent and considers potential significant improvements in receiver performance using this filer.

The Q-enhanced filter redesign ports the existing filter to a new integrated circuit technology which performs better at higher frequencies. The redesign in particular addresses problems in the previous design. The frequency divider design is modified, resistance tuning is added, and additional modifications to the overall filter functionality are implemented. General problems in obtaining an ideal passband shape by eliminating unwanted coupling are addressed. The supporting software for the tuning algorithm is modified to use analog controls and shown to achieve further narrowed bandwidths of 5 MHz and 2.5 MHz at center frequencies of 500 MHz, which are demonstrated to be temperature stable. Future software modifications are described to prepare the existing code base for the new filter design.

Potential applications for a Q-enhanced filter include improving the performance of receiver designs. One of the most important performance parameters of a receiver is its spurious response rejection. To explore this behavior, an automated test system is developed to characterize receivers, and four receivers are tested. The test results are presented in a novel graphical display, which is used to evaluate receiver performance and compare receivers. These results motivated the development of a potential modified superheterodyne receiver architecture using the Q-enhanced filter as an image filter and an IF filter. The viability of this receiver design is tested and shown to provide significant improvements to receiver’s spurious rejection response.

Table of Contents

List of Figures

List of Tables

Acknowledgements

My first thanks must go to Dr. Kuhn. I’ve enjoyed many of his classes and benefited greatly from his teaching over several years. Dr. Kuhn truly enjoys his work and cares about teaching that knowledge. I am so grateful to have honor of working under his tutelage.

Thank you also to the rest of my committee, Dr. Rys and Dr. DeVault. I appreciate your time and feedback.

I must also thank Renee Strouts and Joel Schonberger. Their work has been essential to my research. Joel in particular left a beautifully documented project for me and tirelessly answered questions as I began to learn the existing system. I hope to find myself working with engineers of his caliber, and superlative documentation skills, in the future.

A thanks also goes to Steven Melton and to Mathew Clewell. They have both been excellent office mates this past year, providing many fun distractions, much good conversation, intelligent technical advice and many sanity checks.

Thank you also to Joel Carroll for the use of a ‘MyDAQ’ and a willingness to answer a string of questions about LabVIEW.

I am particularly grateful to my close friends Aimee Smith and Josh Zavala for their encouragement and support this past semester. Their help, while dramatically different from each other, has been much appreciated and I am grateful to them both.

I am grateful to so many people not mentioned here. I’m honored to have received the time and effort so of many professors who were doubtlessly quite busy and still found away to answer yet another question. I have studied with many students and learned a great deal from them. I’ve known and learned with many people here at K-State and benefited from their insight. Thank you.

Dedication

This thesis is dedicated to my parents, because they taught me to dream about learning.
This thesis is dedicated to my Grandma Verna, because she taught me how to have faith in my dreams and work to achieve my dreams one pragmatic step at a time.
This thesis is dedicated to the souls that I have known along the path to this dream, because they have taught me how to live and how to love. I hope in the end I have made their lives better as they have made mine. Most of all, I wish them the courage to dream and the faith and passion to achieve their dreams.

Chapter 1 - Introduction

1.1 Objective

This thesis is divided into two parts. First, this thesis documents the redesign of the Two-Pole Q-Enhanced Band-Pass Filter into a new integrated circuit (IC) technology. Problems in the previous filter design are explained and solutions are explored. Additionally, the supporting software is refined with additions and improvements to the pre-existing tuning algorithm and changes to the supporting software and hardware needed for the redesigned IC portion of the filter are explained. Second, this thesis considers the currently used methods for, and historical emphasis placed on, characterizing a receivers spurious rejection responses. The development of an automated test system to explore this behavior is described and the test results are presented in a novel graphical format. Based on the demonstrated capability of the Q-enhanced filter and insight from the spurious rejection response data, this thesis proposes a new receiver architecture. Finally, this architecture is tested to determine if it could significantly improve a receiver's spurious rejection capability.

1.2 Q-Enhanced Filter Development Primer

1.2.1 Integrated Filter Design

1.2.1.1 Previous Work

The existing Q-enhanced filter used as a starting point for this thesis was designed by Renee Strouts based on circuit concepts developed in a class project. The original design is documented in [1]. This active filter uses Q-enhancement to create a tunable variable bandwidth bandpass filter. Q-Enhancement is explained thoroughly in both [1] and [2] and will only be defined here briefly to lend context to this thesis's discussion of the filter.

1.2.1.2 Q-Enhancement

'Q-enhancement' refers to the technique of increasing the quality factor of an inductor, capacitor, or tuned circuit [3]. The 'quality factor' is defined by the ratio of the energy stored to the energy dissipated in these components. In general, Q is defined mathematically by equation (1.1).

$$Q = 2\pi \times \frac{Energy\ Stored}{Energy\ Dissapated\ Per\ Cycle} \quad \textbf{(1.1)}$$

A more detailed explanation of Q and the associated RF models of inductors and capacitors can be found in [2]. The important idea to understand here is that every inductor and capacitor includes a non-ideal resistance which limits the components performance at sufficiently high frequencies. This behavior is crucial to filter design because the Q of a filter fundamentally limits its achievable bandwidth according to (1.2).

$$Bandwidth = \frac{f_0}{Q} \quad \textbf{(1.2)}$$

where f_0 is the center frequency of the filter passband and Q is the composite filter quality factor. Typically, the overall Q of a filter has been limited by the inductors used. In particular, high Q inductors are very difficult to manufacture in integrated circuits [4]. In this work off chip inductors are used, but the previous theses and associated research include efforts to achieve high Q on chip inductors [5]. Q-Enhancement is achieved in the filter in this thesis using cross coupled field effect transistors.

1.2.1.3 Revised Integrated Circuit Block Diagram

The top level block diagram for the revised Two-Pole Q-Enhanced Band-Pass Filter is shown in Figure 1.1.

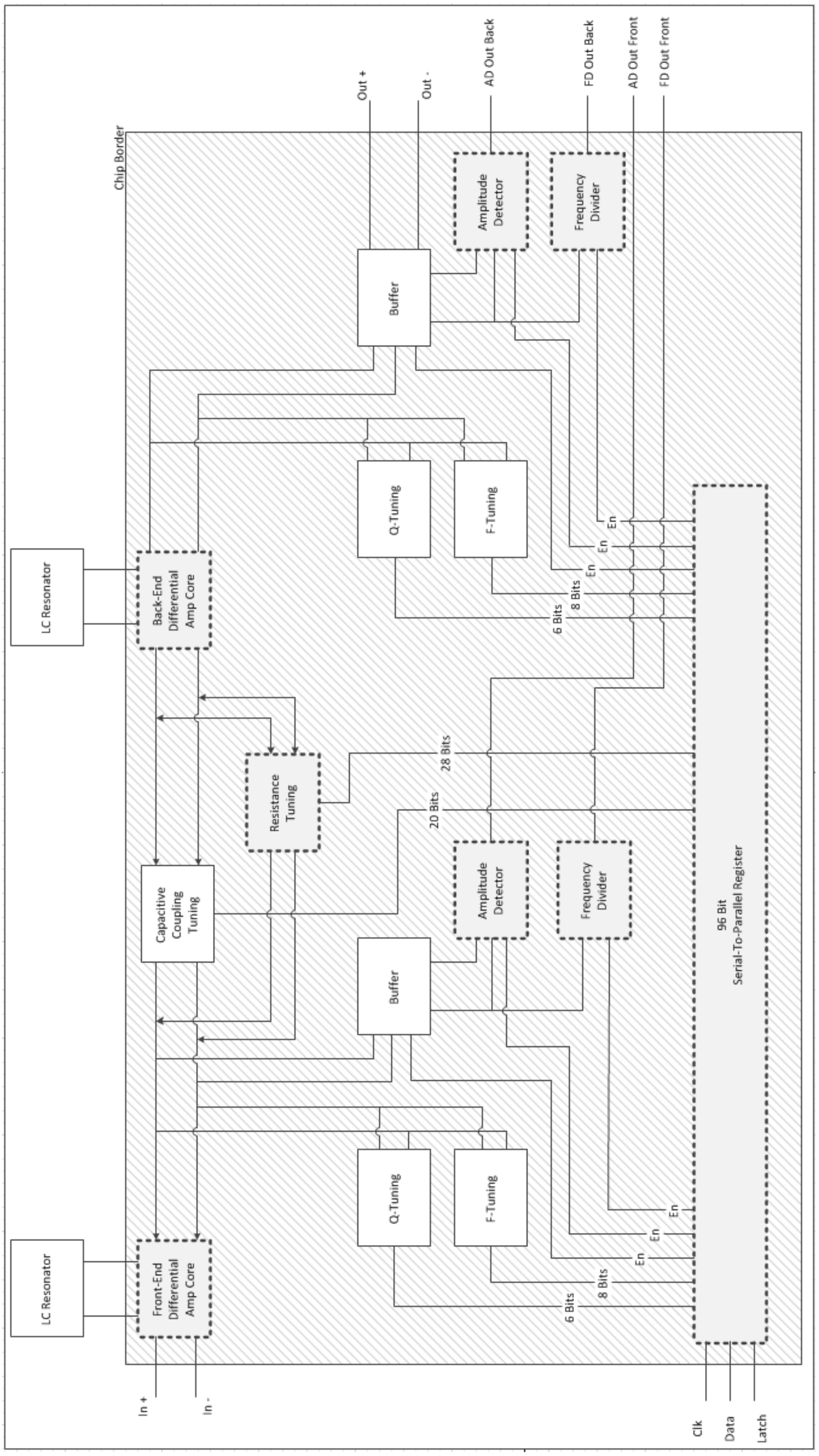


Figure 1.1 – Q-Enhanced Filter Block Diagram

Similar to previous designs, the filter's input signal is driven into a differential cascoded amplifier core, labeled 'Front-End'. The second core, the 'Back-End' is an identical copy of the Front-End with grounded inputs. The amplitude detection and frequency division circuits for the Front-End and Back-end are driven by the cores via identical buffers to protect the filter from too much loading. The filter is programmed via the serial to parallel register from a microcontroller which runs the supporting software introduced in section 1.2.2. This communication is a bit stream which controls the enabling of the buffers, amplitude detectors, and frequency dividers. The rest of these bits control binary weighted cells of Q-enhancement, frequency tuning, capacitive coupling and resistance tuning used to tune the filter center frequency, bandwidth and shape of the passband. The LC tank circuits are off-chip resonators driven by the Front-end and Back-end. The final output of the filter is driven differentially from the Back-End buffer.

The entire IC was ported to a .18µm SOI process technology which is lower power and better performance at higher frequencies than the .5µm and .25µm SOI processes used previously. Portions of the circuit were redesigned to add functionality, improve performance or to fix problems in the existing filter. These modified circuit designs are shown shaded slightly darker with a dashed outline in Figure 1.1. The serial to parallel register was increased from 64 bits to 96 bits to control additional circuitry. The intrinsic gain of the cascoded amplifier cores was lowered to 1 to improve the dynamic range. Resistance tuning circuitry was added to cancel unwanted coupling affects. The frequency dividers were redesigned entirely to eliminate internal oscillation problems in the current design. Also, a pre-existing design flaw was discovered in the amplitude detector circuit. The design changes to the frequency divider and the addition of the resistance tuning are explained in section Chapter 2. The design flaw in the amplitude detector is also explained in Chapter 2. The rest of the circuit design as ported to the new process is documented in appendix A.

1.2.2 Supporting Hardware and Software

1.2.2.1 Previous Work

The filter is implemented on a circuit board using a microcontroller to program and tune the filter for testing and implementation. In the previous thesis work by Joel Schonberger, a test application was written in C# to create a graphical user interface (GUI) which allowed the user to control the filter manually or provide settings to an automated tuning algorithm. The circuit board, the supporting software written for the microcontroller, and GUI are documented thoroughly in [2]. The review here is therefore brief and provided only as a basis for understanding the additions to this supporting software described in this thesis.

1.2.2.2 Software Additions

It has been a long time goal of the work this thesis continues to achieve fractional bandwidths of one percent or less relative to the center frequency. Achieving this narrow bandwidth required the addition of fine tuning in the existing automated tuning algorithm. Those additions are implemented and tested and documented in Chapter 3 and Appendix B. The changes to the filter design also create a need to modify the supporting code for the microcontroller and the GUI. The changes to the code for the microcontroller are explained in Chapter 3. Changes to the GUI are also suggested, but not yet implemented.

1.3 Receiver Architectures and Filter Applications

1.3.1 Motivation

A crucial issue in today’s wireless communication technologies is maximizing throughput in the allocated spectrum. As a result increasing demands are being placed on communication technology. According to Michael Marcus, retired associate chief for Technology with the FCC, “Transmitters don’t use spectrum, receivers do.” [6] Therefore, if receiver performance is improved, the spectrum can be used more efficiently. Given the rapidly increasing popularity of devices using wireless technologies, the demand on the RF spectra is growing. Improving receiver’s performance to meet this demand is an important goal.

Receiver performance is a complex topic with a long history. To improve upon current designs it is necessary to quantify current receiver performance to accurately assess the current state of the technology and gain insight into how it could be improved. The task of a receiver is to detect and translate the signal it’s tuned to receive without being affected by any other signal. One way to measure how well a receiver does this is to measure the receiver’s spurious rejection response. This work attempts to address the need to measure receiver performance by developing a spurious response rejection test system and developing a novel graphical format to display the results.

The limiting factor in improving receiver performance is largely governed by the ability to filter and completely isolate only the desired signal. As a result, much research has been done to design optimal filters. Q-Enhancement has been considered as an option for use integrated receivers in previous work [7] and [8]. However, it's been assumed that the limited dynamic range and high noise figure associated with Q-enhancement would compromise receiver performance [9]. This research in this thesis characterizing receiver's spurious rejection response indicates this conclusion is not fully correct.

1.3.2 Prior Art

There are many criteria used to evaluate various aspects of receiver performance including but not limited to, sensitivity, noise figure, dynamic range, third order intercept, IF rejection, and adjacent channel rejection. This criterion is used both by amateurs [10] and in industry and academia [11]. Another technique often used to look mixing schemes is the so-called 'spur chart' in which a diagram is developed to illustrate potential combinations of incoming signal frequencies and their harmonics and f_{LO} and its harmonics that a receiver may respond to [12]. Despite the useful information this diagram contains, it is difficult to understand quickly. Moreover, no information about the severity of the spurious response is identified.

Literature generally emphasizes the important causes of spurious responses in receivers to be mixing, IF separation, harmonics and coupling with existing signals in the receiver [13]. Other work has explored automated testing spurious rejection responses to apprehend the full complexity of receiver's performance [14], [15].

Unfortunately, spur charts and the many various standards of receiver performances mentioned above fail to yield an intuitive assessment of the receiver's spurious response rejection performance. Even the works on automated spurious response testing, while quite thorough, didn't offer a simple way to view and intuitively evaluate the receiver performance. Chapter 4 in this work addresses a new automated spurious rejection response test system and develops and demonstrates a useful, intuitive graphical display of the test results.

1.3.3 Research Accomplished

1.3.3.1 Spurious Rejection Response Testing System

To understand and characterize receiver spurious rejection response an automated test system was developed. This system allowed four receivers to be tested thoroughly over different amplitude ranges and different frequencies. The system uses a ‘MyDAQ’ and a LabVIEW based test GUI. This system is explained at length in section 4.3.

1.3.3.2 Spurious Rejection Response Results

The data obtained from the Spurious Rejection Test System proved extensive. A graphical display of the results was developed providing insights into the four receivers tested. Explanations for the spurious responses observed were analyzed and evaluated in section 4.5.

1.3.3.3 Filter Application

Finally, a modified superheterodyne receiver using the Q-enhanced filter is proposed in Chapter 5. The potential improvements in spurious rejection are partially tested using the Spurious Rejection Test System. The results strongly indicate this solution might provide an excellent alternative to current receiver architectures.

Chapter 2 - Q-Enhanced Filter Redesign

2.1 Design Overview

This Chapter documents the redesign of a two-pole Q-enhanced band-pass filter IC originally designed by Renee Strouts [1]. The first section will explain briefly the process of porting this design to a new technology. The next section will focus on the problems with an asymmetrical passband in the previous filter design. The origin of this asymmetry is explained theoretically and the solution in the hardware design is documented. Last, this section looks at the circuits which were changed significantly from the original design or had intrinsic issues in the original design.

2.2 Porting Design to Different Integrated Circuit Technology

This section describes porting the previous IC design in a silicon-on-sapphire (SOS) process to a bulk SOI process. The bulk SOI process runs on a lower power voltage and has a smaller minimum length of .18μm than the SOS process. The new process also includes body contacts and a different k_n' value. As a result, porting the circuit design required re-biasing the circuits and choosing new W/L ratios which matched the circuits design specifications.

In general, analog design using metal on oxide semiconductor field effect transistors (MOSFETs or 'FET's) at an IC level is ruled by well-known equation (2.1)

$$I_D = \frac{k_n'}{2}\frac{W}{L}(V_{GS} - v_t)^2 \qquad \textbf{(2.1)}$$

for FETs in the active region neglecting Early effect. Long channel FETs in the triode region are described by (2.2).

$$I_D = k_n'\frac{W}{L}\left((V_{GS} - v_t)v_{DS} - \frac{v_{DS}^2}{2}\right) \qquad \textbf{(2.2)}$$

Also,

$$k_n' = \mu C_{ox} \qquad \textbf{(2.3)}$$

and

$$V_{OV} = (V_{GS} - v_t) \qquad \textbf{(2.4)}$$

The above equations are well known, but the quadratic term is only correct for FETs that are 'long channel' with a sufficiently small overvoltage. If the FET is 'short channel' or the overvoltage is large enough, the equation for I_D versus V_{GS} the active region reduces to (2.5).

$$I_D = \frac{k_n'}{2}\frac{W}{L}(V_{GS} - v_t') \qquad \textbf{(2.5)}$$

Where the new threshold voltage, v_t', is the interpolated V_{GS}-axis intercept of the linear portion of the I_D versus V_{GS} relationship. A particular case of this behavior in a FET is shown in Figure 2.1. This behavior is not typically explained in textbooks, but is consistently exhibited in experimental data [16].

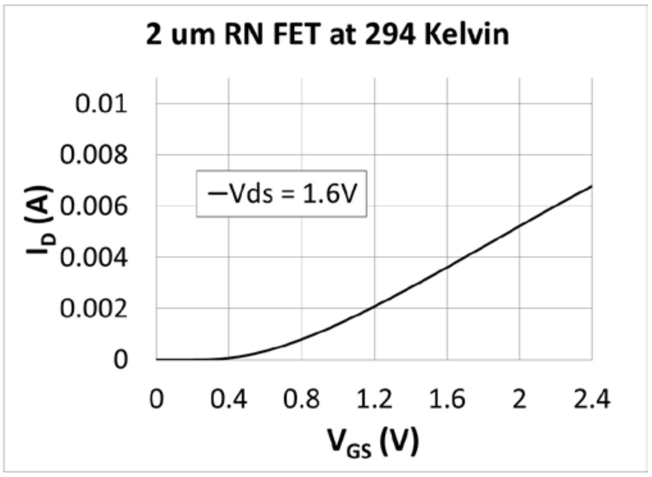


Figure 2.1 – I_D versus V_{GS} Curve Showing Quadratic and Linear Behavior

It's easy to see in the I_D versus V_{GS} above that the quadratic behavior only lasts from about V_{GS} =.35 V to VGS = .8 V. After V_{GS} increases past about .8 V, the current increases linearly. This behavior, as stated above, may start nearly as soon as the FET is in saturation if the length is small enough. The new process lengths used in the Q-enhanced filter redesign were so small that most design work assumes that the FETs are short channel.

Two other equations are important during this design when a FET is used as a switch. First, when the FET is 'on' so that it's conducting current and in the triode region, the resistance that signals see from drain to source, r_{ON}, is given in (2.6).

$$r_{ON} = \frac{1}{\frac{k'_n}{2}\frac{W}{L}(v_{gs}-v_t)} \quad \textbf{(2.6)}$$

Simultaneously, the capacitance of the FET can be calculated using (2.7).

$$C = nWLC_{ox} \quad \textbf{(2.7)}$$

which can then be used to find the impedance of a FET using (2.8)

$$X_c = \frac{1}{2\pi f C} \quad \textbf{(2.8)}$$

The 'n' in (2.7) is a fractional value between zero and one determined by the signal path through the FET and whether the FET is off, in triode or in saturation.

Finally, since the overall design of the previous IC was robust and working well, most circuits could be redesigned by simply assuming the same biasing scheme and altering the W/L ratio to compensate for the change in k'_n. However, this assumed the body effect would be negligible. While this would simplify porting the design, some circuits needed to be addressed in more detail. In general good design practice dictated that all circuits needed to be simulated and evaluated individually to ensure a robust design that matched, or ideally exceeded, its predecessor's performance.

2.3 Asymmetry in the Passband

This section deals with pronounced asymmetry in the passband shape, a major problem in the previous design. This problem is documented extensively in [2] and illustrated in Figure 2.2. The theoretical origin of this asymmetry is investigated and determined to be a result of two issues in the previous design: the non-idealities of inductive coupling with finite Q inductors and an error in the original coupling capacitor circuit design.

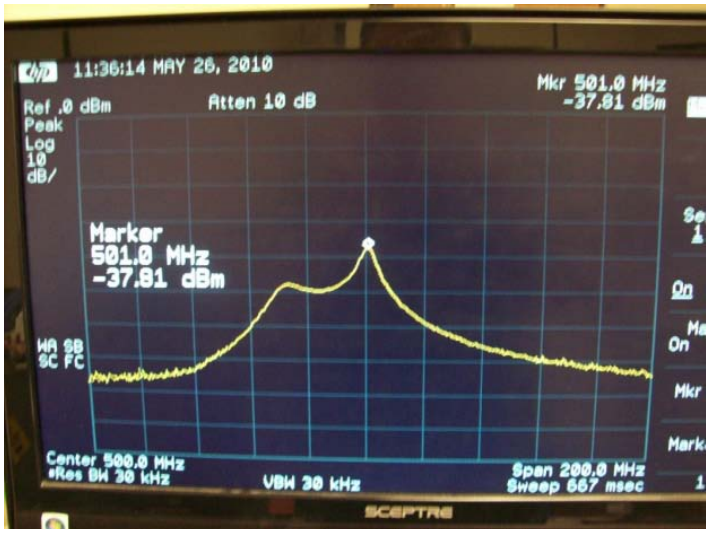


Figure 2.2 – Filter Response Showing Asymmetric Passband (Used with Permission [2])

2.3.1 Sources of Asymmetry

Both sources of asymmetry are explored and characterized mathematically using admittances in [2]. However, there is an algebraic error in the solution describing the inductive coupling, so the corrected solution is explained in section 2.3.1.2 followed by the circuit level solution. To prepare a basis for explaining the hardware design solution to these asymmetries, admittances are reviewed in section 2.3.1.1. Finally, the solution for the coupling capacitors derived in [2] is presented in section 2.3.1.3 along with the circuit level solution.

2.3.1.1 Admittance Review

The basic ideas of admittance are presented here to provide a context for the discussion of characterizing the asymmetries in the passband. The definitions of y-parameters are shown in Figure 2.3 and equations (2.9) – (2.13). Y parameters for inductors, capacitors and resistors are derived in (2.14) – (2.16).

Admittance is defined as the inverse of impedance and can described with the two port network shown in Figure 2.3.

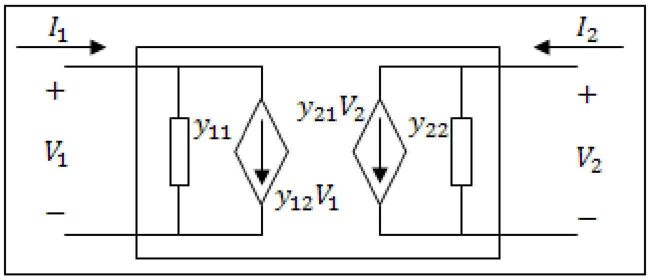


Figure 2.3 – Two Port Network (Used with Permission [2])

This network allows the following definitions to be developed.

$$\begin{bmatrix} i_1 \\ i_2 \end{bmatrix} = \begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \end{bmatrix} \qquad \textbf{(2.9)}$$

Input port admittance

$$y_{11} = \left.\frac{i_1}{V_1}\right|_{V_2=0} \qquad \textbf{(2.10)}$$

Forward Transfer Admittance

$$y_{21} = \left.\frac{i_2}{V_1}\right|_{V_2=0} \qquad \textbf{(2.11)}$$

Reverse Transfer Admittance

$$y_{12} = \left.\frac{i_1}{V_2}\right|_{V_1=0} \qquad \textbf{(2.12)}$$

Output Port Admittance

$$y_{22} = \left.\frac{i_2}{V_2}\right|_{V_1=0} \qquad \textbf{(2.13)}$$

Using the definitions in (2.10) – (2.13) a resistor, capacitor and inductor, connected between ports one and two, are characterized in the next three equations.

$$\text{Resistor: } y_{21} = \left.\frac{i_2}{V_1}\right|_{V_2=0} = \frac{i_2}{Ri_1} = \frac{-i_1}{Ri_1} = -\frac{1}{R} \qquad \textbf{(2.14)}$$

$$\text{Capacitor: } y_{21} = \left.\frac{i_2}{V_1}\right|_{V_2=0} = \frac{i_2}{(-jX_c)i_1} = \frac{-i_1}{(-jX_c)i_1} = \frac{-j}{X_c} \qquad \textbf{(2.15)}$$

$$\text{Inductor: } y_{21} = \left.\frac{i_2}{V_1}\right|_{V_2=0} = \frac{i_2}{(jX_L)i_1} = \frac{-i_1}{(jX_L)i_1} = \frac{j}{X_L} \qquad \textbf{(2.16)}$$

By symmetry for these elements, y_{12} will equal y_{21}. A similar process could be used to find y_{11} and y_{22}, which should also be equal to each other by symmetry.

2.3.1.2 The First Source of Asymmetry

As explained above, all inductors are limited by their Q value and have a small amount of resistance. The previous work determined that the coupling between the inductors in the LC tank circuits was not purely inductive due to the limited Q of the inductors [2]. As a result the inductors introduced an unwanted coupling term 90° out of phase with the desired LC coupling used in realizing the basic 2-pole response. To quantify and understand this unwanted coupling, the impedance of the LC tank circuits are modeled mathematically in section 2.3.1.2.1.

2.3.1.2.1 Origin of Asymmetry: Inductive Coupling

The LC tank circuits for the Q-enhanced filter are off chip and laid out using two discrete inductors and a capacitor as shown in Figure 2.4.

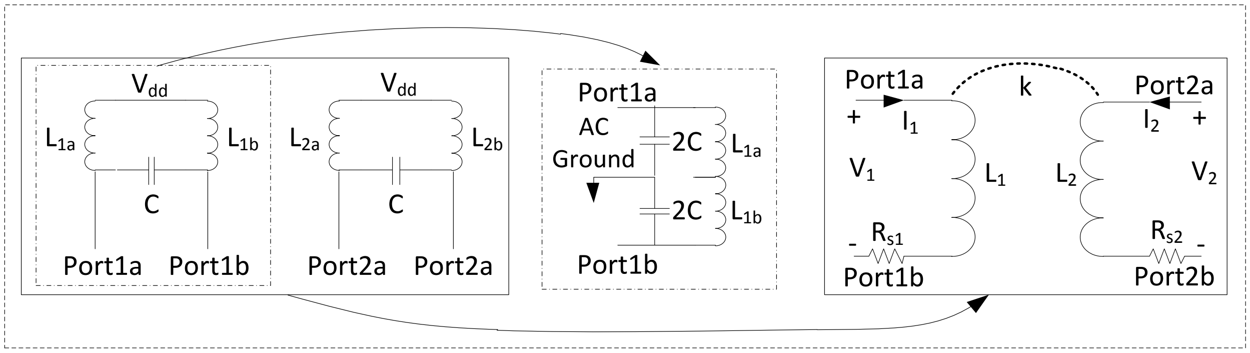


Figure 2.4 – Topological Transformation of LC Tank Circuit

This topology shown on the left of Figure 2.4 was used to allow the desired biasing. It is not immediately obvious how to translate these two circuits into two port network. The topological transformation required is shown in Figure 2.4, beginning with the topology of the tank circuits and ending with the circuit rearranged into a two port network topology for admittance analysis of the inductor coupling issue.

From Figure 2.4 it's defined that $2L_{1a}$=2 L_{1b}= L_1. The capacitors are omitted since we only need to consider the inductive coupling, shown as k in the last box to the right. Using these definitions the forward transfer admittance can be derived as follows

$$L_1 = L_2,\ R_{s1} = R_{s2} = R_s, M = k\sqrt{L_1 L_2} \qquad \textbf{(2.17, 2.18, 2.19)}$$

$$V_1 = j\omega L I_1 + j\omega L I_2 k + I_1 R_s \quad \textbf{(2.20)}$$

$$0 = j\omega L I_2 + j\omega L I_1 k + I_2 R_s \quad \textbf{(2.21)}$$

Solving (2.21) for I_1 and letting $X_L = \omega L$

$$I_1 = -\frac{I_2(jX_L + R_s)}{jX_L k} \quad \textbf{(2.22)}$$

Substitute (2.22) into (2.20) where $X_L = \omega L$

$$V_1 = -\frac{I_2(jX_L + R_s)}{k} + jX_L I_2 k - \frac{I_2(jX_L + R_s)R_s}{jX_L k} \quad \textbf{(2.23)}$$

$$y_{21} = \frac{I_2}{V_1}\bigg|_{V_2=0} = \frac{1}{-\frac{(jX_L+R_s)}{k} + jX_L k - \frac{(jX_L+R_s)R_s}{jX_L k}} \quad \textbf{(2.24)}$$

Combining fractions, inverting and multiplying out yields

$$y_{21} = \frac{jX_L k}{-(jX_L+R_s)jX_L - (jX_L+R_s)R_s} \quad \textbf{(2.25)}$$

$$y_{21} = \frac{jX_L k}{-(-1){X_L}^2 - jR_s X_L - jX_L R_s + R_s^2} \quad \textbf{(2.26)}$$

$$y_{21} = \frac{jX_L k}{R^2 + {X_L}^2 - j2X_L R_s} \quad \textbf{(2.27)}$$

$$y_{21} = \frac{jX_L k}{(R_s^2 + {X_L}^2) - j2X_L R_s}\left(\frac{(R_s^2+{X_L}^2) + j2X_L R_s}{(R_s^2+{X_L}^2) + j2X_L R_s}\right) \quad \textbf{(2.28)}$$

$$y_{21} = \frac{-2{X_L}^2 R_s k}{(R_s^2+{X_L}^2)^2 + 4{X_L}^2 R_s^2} + \frac{jX_L k(R_s^2+{X_L}^2)}{(R_s^2+{X_L}^2)^2 + 4{X_L}^2 R_s^2} \quad \textbf{(2.29)}$$

Now letting $Q \gg 1$, so $R_s \ll X_L$ due to $R_s = {}^{X_L}\!/_{Q}$, all R_s^2 go to zero.

$$y_{21} = \frac{-2R_s k}{{X_L}^2} + \frac{jk}{X_L} = \frac{k}{X_L}\left(-\frac{2}{Q} + j\right) \quad \textbf{(2.30)}$$

Last, we cancel the unwanted real component of y_{21} using the admittance of a resistor derived in (2.10). The necessary resistance to eliminate the unwanted y_{21} admittance can be found by summing that admittance with the admittance of a resistor and solving for the resistor. The resulting resistance value is shown calculated in (2.27).

$$\frac{(R_s^2+{X_L}^2)^2 + 4{X_L}^2 R_s^2}{2{X_L}^2 R_s k} \approx \frac{{X_L}^2}{2R_s k} \approx \frac{QX_L}{2k} = R \quad \textbf{(2.31)}$$

where the real parts in (2.29) and (2.30) are shown having solved for the wanted resistance value. From (2.31) we observe that in the ideal case y_{21} is equal to $\frac{jk}{X_L}$ due to the quadrature relationship between V and I in an inductor. This is the term needed in a coupled resonator filter to from the desired passband response [17]. However, (2.31) shows an additional undesired "in-phase" term as well. This term gives rise to the asymmetric shape seen in Figure 2.2 previously. This result in (2.29) was confirmed using ADS. That simulation and its output are shown in Figure (2.5) – (2.6).

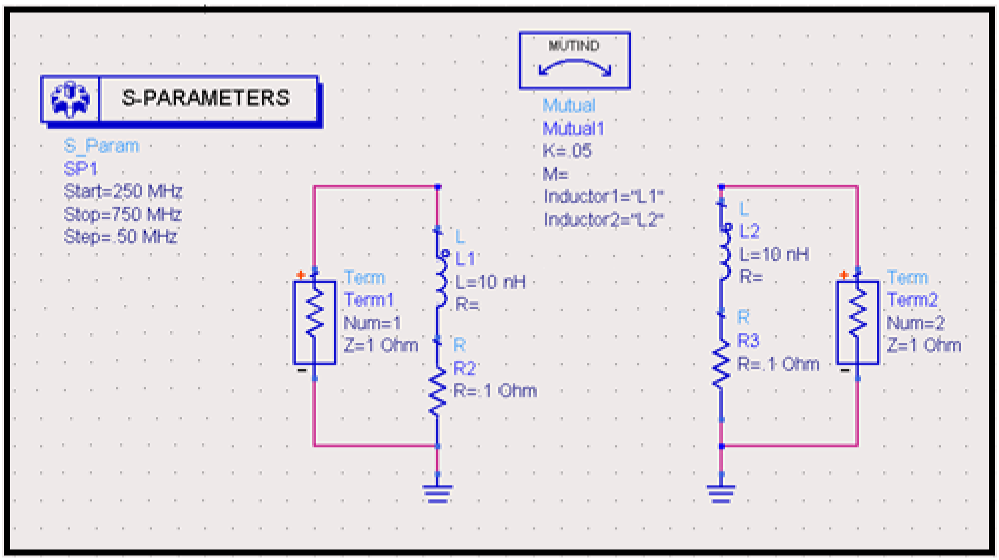


Figure 2.5 – ADS Circuit Simulating Admittance Parameters

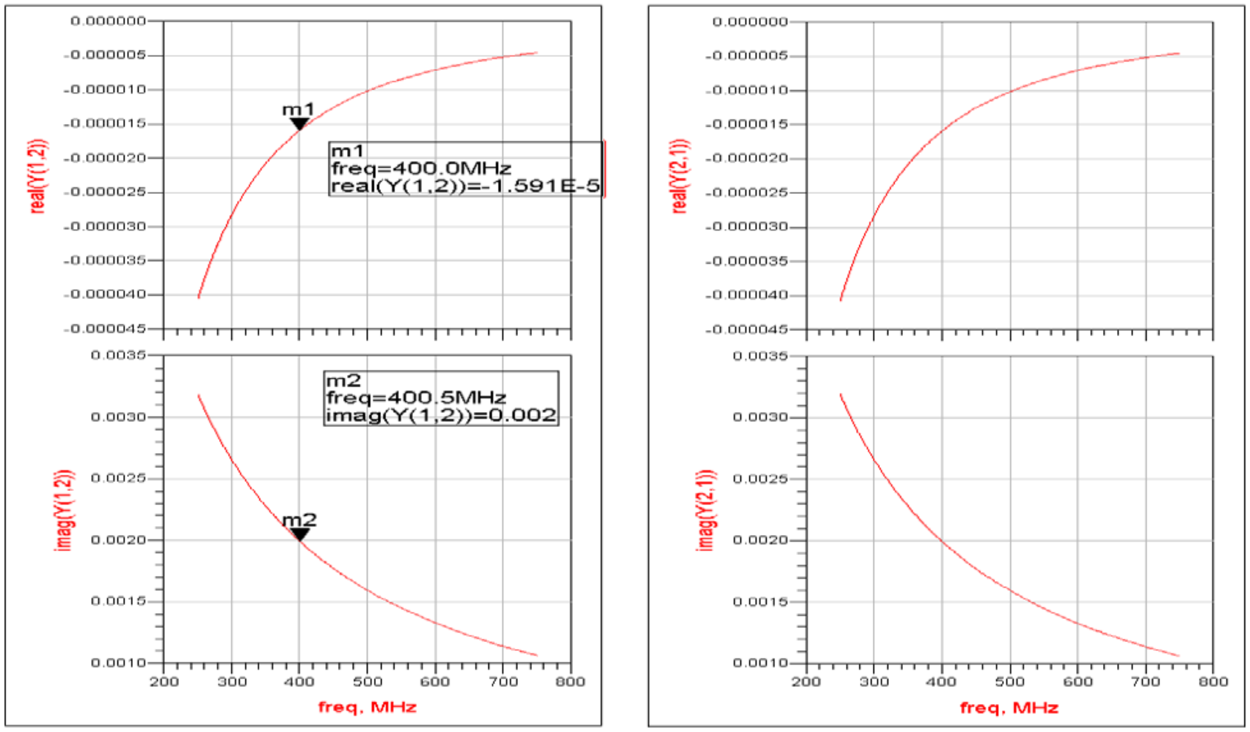
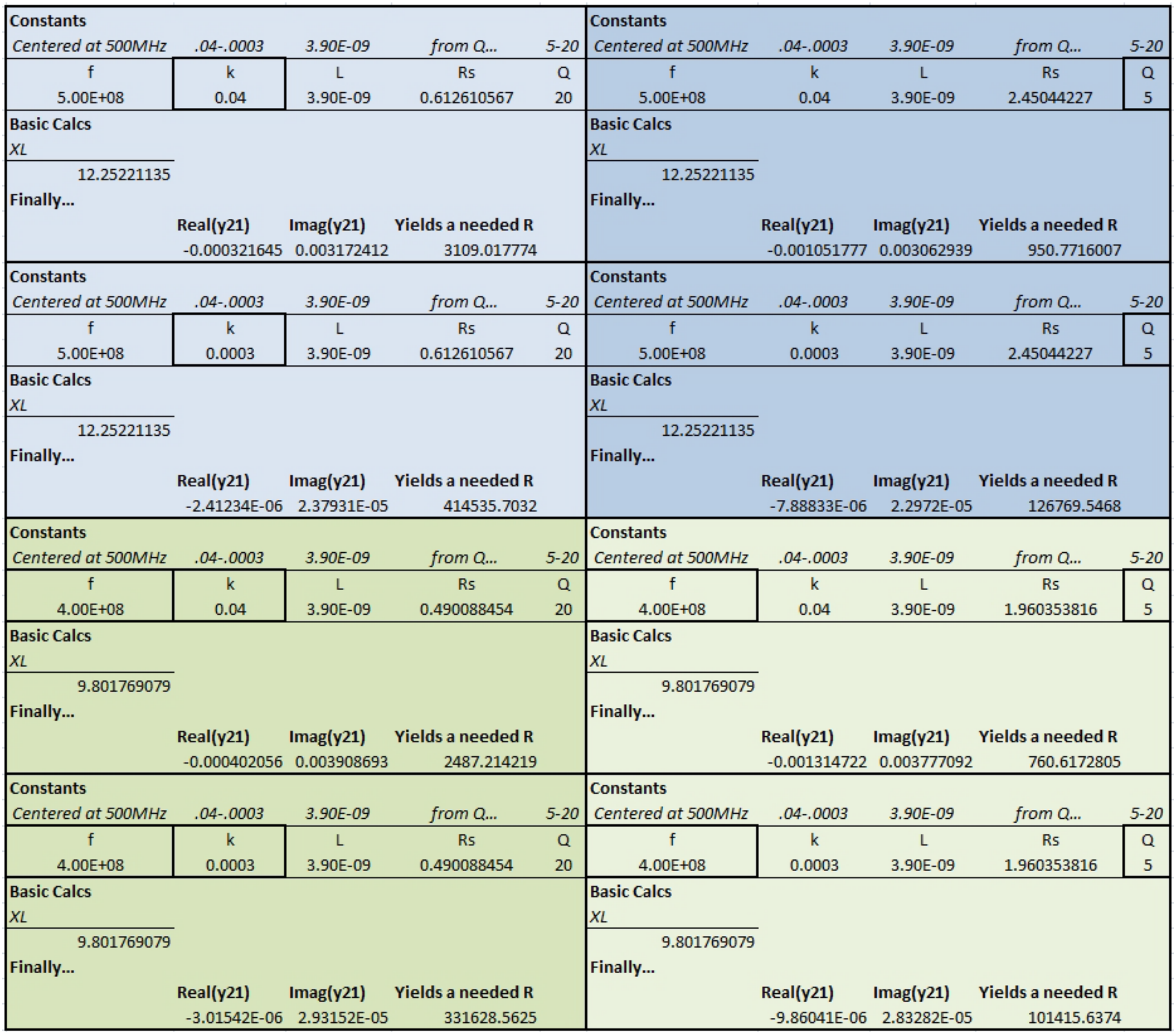


Figure 2.6 – ADS Simulation Output of Admittance Parameters

The simulation results in Figure 2.6 show agreement with the full form of the solution. The next section describes the circuit level solution to the unwanted portion of the inductive coupling in the next Q-enhanced filter design.

2.3.1.2.2 Circuit Design Solution: Resistance Tuning

The work above shows that resistors interconnecting the LC tank circuits can be used to cancel the asymmetry in the passband. According to the result in the previous section the resistance value required will depend on frequency of operation, the strength of the coupling between the inductors and the Q of the inductors. The k value, or the amount of coupling between inductors, is documented in [2]. The range of frequencies considered is 400 MHz - 500 MHz. A typical range of Q values for on board inductors range from 5-20. Using this equation and these ranges, a potential range of resistance values was calculated using the full form of the solution and varying these parameters from the expected minimum to the expected maximum. A table summarizing these results is shown in Figure 2.7.



Constants					**Constants**				
Centered at 500MHz	*.04-.0003*	*3.90E-09*	*from Q...*	*5-20*	*Centered at 500MHz*	*.04-.0003*	*3.90E-09*	*from Q...*	*5-20*
f	k	L	Rs	Q	f	k	L	Rs	Q
5.00E+08	0.04	3.90E-09	0.612610567	20	5.00E+08	0.04	3.90E-09	2.45044227	5
Basic Calcs					**Basic Calcs**				
XL					*XL*				
12.25221135					12.25221135				
Finally...					**Finally...**				
	Real(y21)	**Imag(y21)**	**Yields a needed R**			**Real(y21)**	**Imag(y21)**	**Yields a needed R**	
	-0.000321645	0.003172412	3109.017774			-0.001051777	0.003062939	950.7716007	
Constants					**Constants**				
Centered at 500MHz	*.04-.0003*	*3.90E-09*	*from Q...*	*5-20*	*Centered at 500MHz*	*.04-.0003*	*3.90E-09*	*from Q...*	*5-20*
f	k	L	Rs	Q	f	k	L	Rs	Q
5.00E+08	0.0003	3.90E-09	0.612610567	20	5.00E+08	0.0003	3.90E-09	2.45044227	5
Basic Calcs					**Basic Calcs**				
XL					*XL*				
12.25221135					12.25221135				
Finally...					**Finally...**				
	Real(y21)	**Imag(y21)**	**Yields a needed R**			**Real(y21)**	**Imag(y21)**	**Yields a needed R**	
	-2.41234E-06	2.37931E-05	414535.7032			-7.88833E-06	2.2972E-05	126769.5468	
Constants					**Constants**				
Centered at 500MHz	*.04-.0003*	*3.90E-09*	*from Q...*	*5-20*	*Centered at 500MHz*	*.04-.0003*	*3.90E-09*	*from Q...*	*5-20*
f	k	L	Rs	Q	f	k	L	Rs	Q
4.00E+08	0.04	3.90E-09	0.490088454	20	4.00E+08	0.04	3.90E-09	1.960353816	5
Basic Calcs					**Basic Calcs**				
XL					*XL*				
9.801769079					9.801769079				
Finally...					**Finally...**				
	Real(y21)	**Imag(y21)**	**Yields a needed R**			**Real(y21)**	**Imag(y21)**	**Yields a needed R**	
	-0.000402056	0.003908693	2487.214219			-0.001314722	0.003777092	760.6172805	
Constants					**Constants**				
Centered at 500MHz	*.04-.0003*	*3.90E-09*	*from Q...*	*5-20*	*Centered at 500MHz*	*.04-.0003*	*3.90E-09*	*from Q...*	*5-20*
f	k	L	Rs	Q	f	k	L	Rs	Q
4.00E+08	0.0003	3.90E-09	0.490088454	20	4.00E+08	0.0003	3.90E-09	1.960353816	5
Basic Calcs					**Basic Calcs**				
XL					*XL*				
9.801769079					9.801769079				
Finally...					**Finally...**				
	Real(y21)	**Imag(y21)**	**Yields a needed R**			**Real(y21)**	**Imag(y21)**	**Yields a needed R**	
	-3.01542E-06	2.93152E-05	331628.5625			-9.86041E-06	2.83282E-05	101415.6374	

Figure 2.7 – Range of Resistance Values

The range of resistance values predicted in Figure 2.7 range from .75 kΩ to 400 kΩ. However, the 400 kΩ case uses a coupling of k=.0003. In [2] this coupling value was shown to be so small that no significant asymmetry was produced in the passband. Therefore here, when choosing the range of values to be implemented, values on the order of 400kΩ were treated as open circuits . Conversely, the lower range of resistance values result from much higher coupling values which were also shown in [2] to cause significant asymmetry in the passband and were given more emphasis when choosing resistance values for the circuit design.

A bank of binary weighted resistors were designed to interconnect the LC tank circuits in the topology shown in Figure 2.8. Values were chosen to meet the range indicated in Figure 2.7, but more importance was given to resistance values resulting from higher coupling coefficients based on the results in [2], the available range of resistance values chosen is 750 Ω to 40 kΩ. The circuit is shown at the top level in Figure 2.8 and with a closer in view in Figure 2.9.

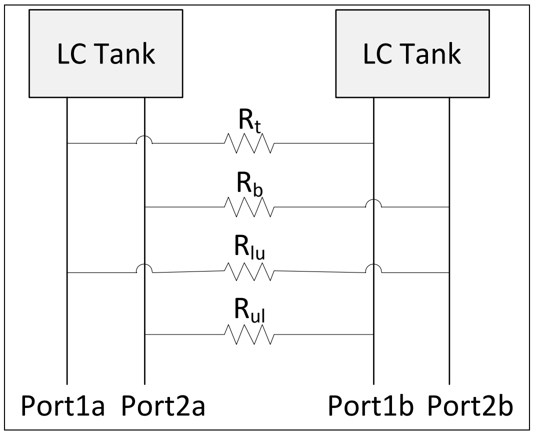


Figure 2.8 – Resistance Tuning Circuitry Topology

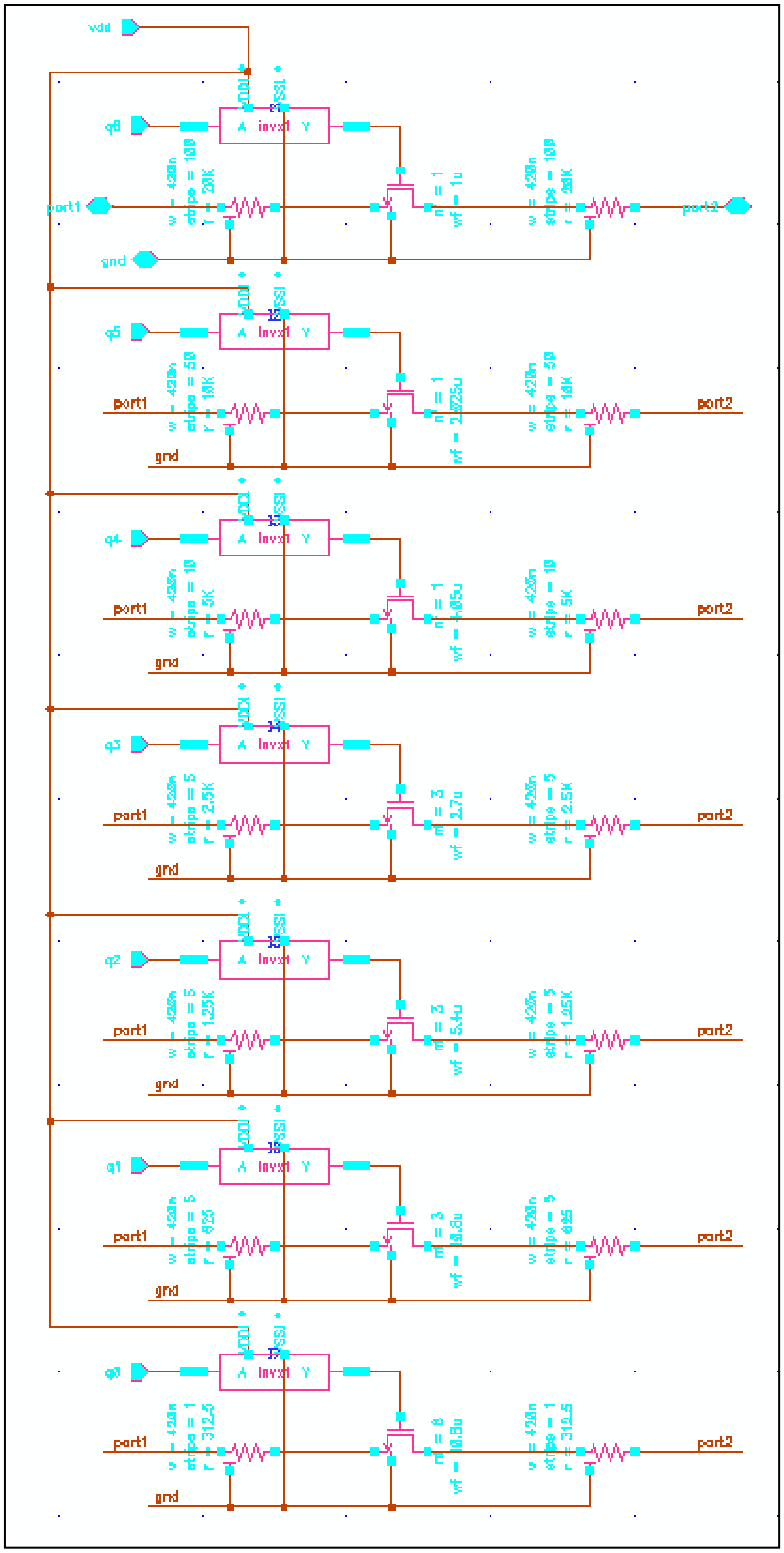


Figure 2.9 – Top View Resistance Tuning Circuitry

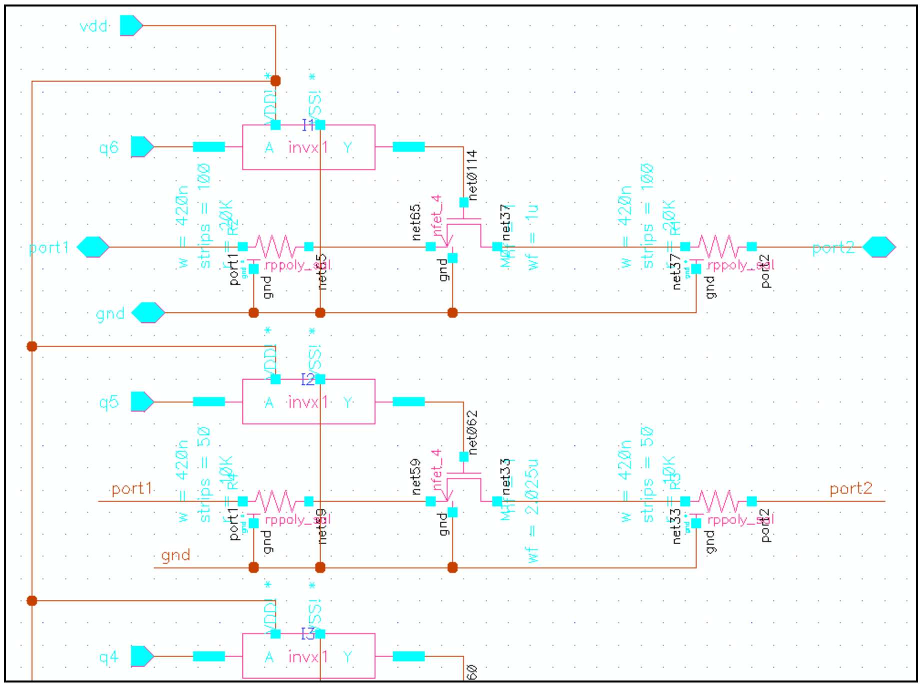


Figure 2.10 – Narrowed View Resistance Tuning Circuitry

The circuits in Figures 2.9 and 2.10 were implemented for each of the resistors shown in Figure 2.8 so these tuning circuits could cancel all possible geometries of inductive coupling.

These resistor banks were designed to be controlled by digital signals from the S/P register driven through inverters to the gate of a FET. When the FET's are on, they are driven in the triode region. When the FETs are off they are seen in the circuit as small capacitors. The first design consideration here was keeping the FET large enough that it's on resistance was negligible relative to the resistors it activated when on. The W/L ratio was calculated by targeting a low on resistance relative to the resistors being driven using equation (2.32).

$$\frac{W}{L} = \frac{1}{r_{ON}\frac{k_n'}{2}(v_{gs}-v_t)} \qquad \textbf{(2.32)}$$

Calculation shows a W/L ratio of 350 was adequate to produce R_{ON} value on the order of 2.5% of the total resistance from port1 to port2 for the 750 Ω resistor when the FET was on.

The second design consideration was to ensure the impedance effects of capacitance due to the FET when it's off are sufficiently small. This restriction dictates that the W/L of the FET be small enough that its impedance was large relative to the resistors it was driving, using equations (2.7) and (2.8). The FETs used to produce the small R_{ON} were shown to produce a complex impedance 14 times larger than the resistors, which should be sufficient to avoid loading ports one and two when the FET is off. Figure 2.11shows the capacitances modeled for the design in the on and off states.

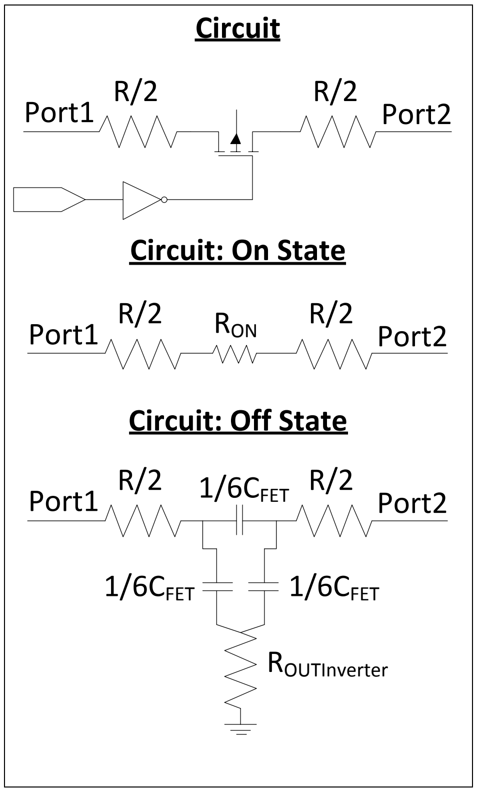


Figure 2.11 – Resistance Tuning Circuitry Models

2.3.1.3 The Second Source of Asymmetry

2.3.1.3.1 Origin of Asymmetry: Incorrectly Driven Coupling Capacitors

The second source of asymmetry in the passband shape was a result of errors in the driving circuitry of the coupling capacitors used to offset the inductive coupling. The assumption originally was that the capacitors, which were driven through a resistor via an inverter saw an AC ground as shown in Figure 2.12.

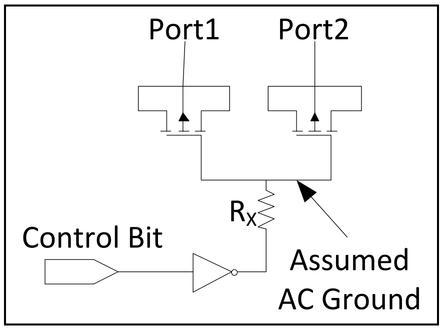


Figure 2.12 – Resistance Tuning Circuitry Models

However, these capacitors were connected between the front-end and band-end LC tank circuits in the same topology used for the resistors in Figure 2.8. Unfortunately the two LC tank circuits contain different signals, so the signals at port1 and port2 didn't cancel each other. As a result the gates of the FETs were not an AC ground.

Similar to section 2.3.1.2 the incorrect loading was modeled using Y-parameters and the admittance needed to cancel the unwanted loading was found to be described by (2.33).

$$y_{21} = \frac{R_X}{{X_C}^2+4{R_X}^2} - \frac{j2{R_X}^2}{X_C({X_C}^2+4{R_X}^2)} \qquad \textbf{(2.33)}$$

(2.33) shows for the unwanted real part of y_{21} to be negligible, X_C must be much greater than R_X. This derivation and confirmation in simulation and tests with the Q-enhanced circuit is documented in [2]. Section 2.3.1.3.2 describes the hardware solution to resolve this loading problem.

2.3.1.3.2 Circuit Design Solution: Corrected Loading

As (2.33) shows, the solution to mitigate the loading is to make the R_X small relative to the impedance of the capacitive impedance of the FET when it's on. The X_C of the FET is calculated by equations (2.7) and (2.8) and can be used to estimate the necessary R_X. The range of coupling capacitor values were achieved as by implementing binary weighted banks of capacitors, similar to the resistor scaling. The R_X values for each bank of capacitors were scaled down as the coupling capacitor values are increased. The schematic for this circuitry is shown below in three views to explain the overall layout of the whole circuitry and provide views of the lower level topology and smaller elements.

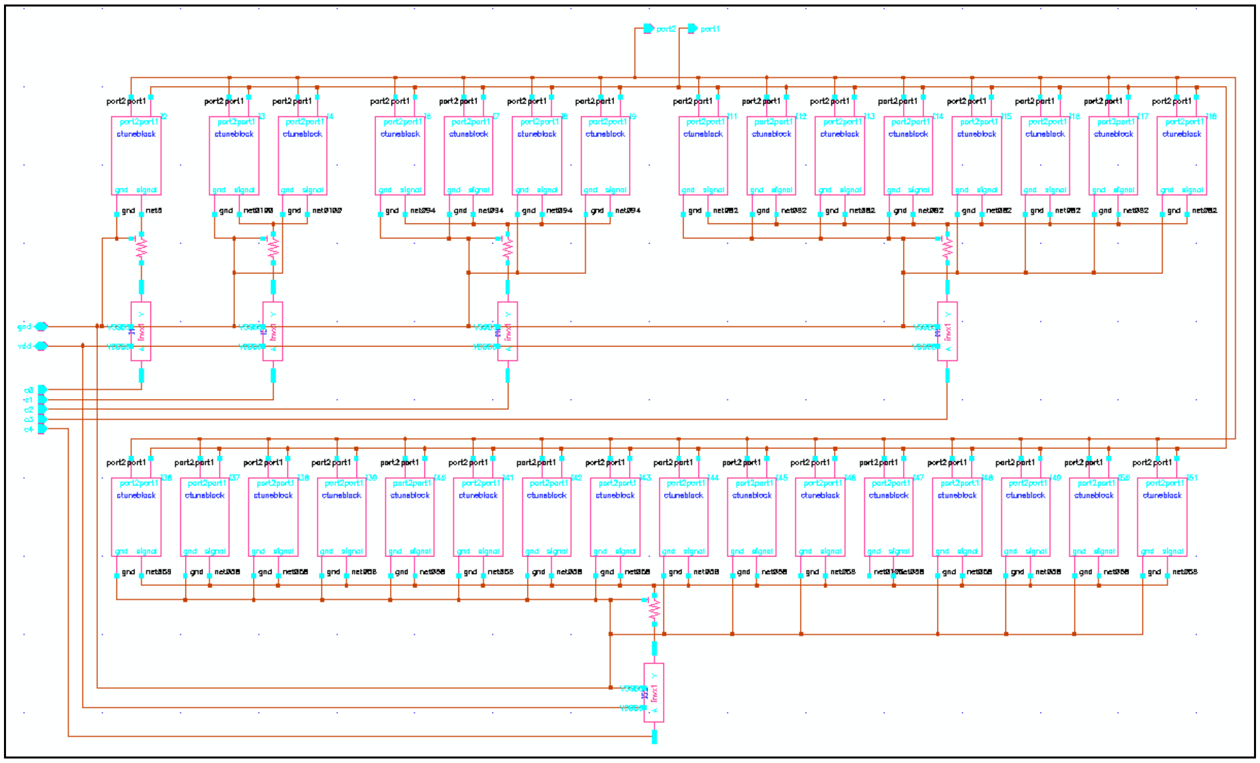


Figure 2.13 – Top View Capacitive Coupling Circuit

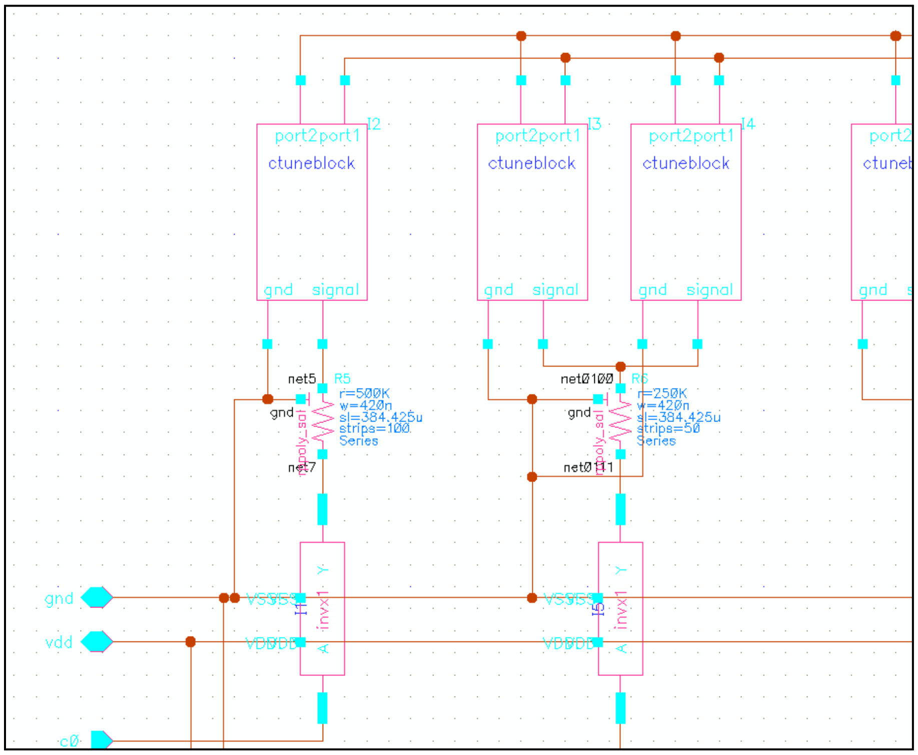


Figure 2.14 – Narrowed View Ctuneblock from Fig. 2.13

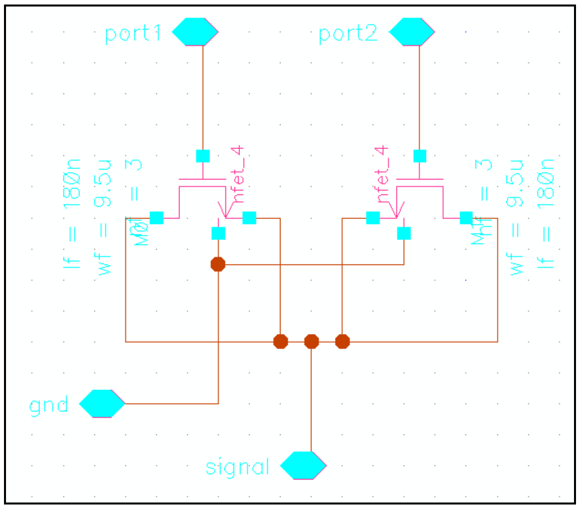


Figure 2.15 – Cell View Ctuneblock

2.4 Additional Circuit Redesigns

2.4.1 Frequency Divider

In the previous design the frequency divider circuitry lacked the desired sensitivity due to an internal oscillation frequency documented in [2]. Many designs were considered as potential solutions because the SOI process the filter was designed in didn't have the digital circuitry that could function reliably at the frequency ranges needed to divide down 500 MHz. However, the new process D-Flip-Flop and inverter circuits performed well according to simulation and the new frequency divider was designed as shown below in Figure 2.16.

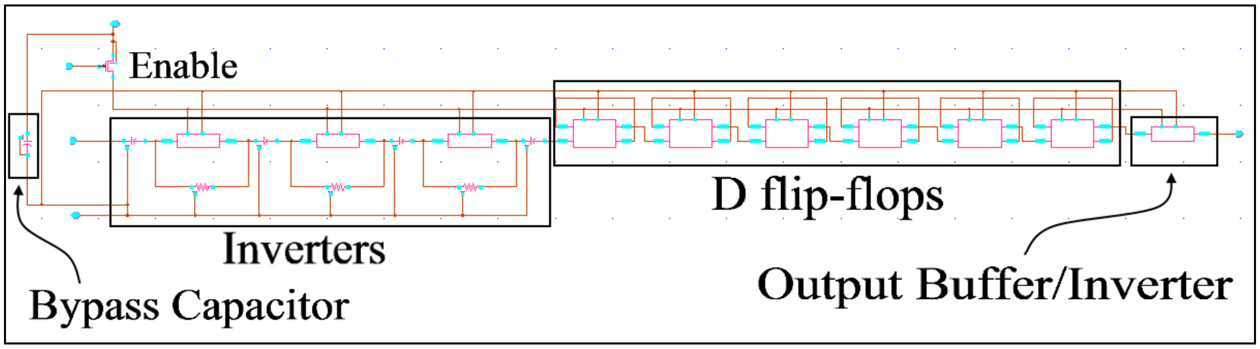


Figure 2.16 – Top View Frequency Divider

This circuit works by using the inverters, self-biased using the resistors shown, and AC coupled to amplify and shift DC offset of the signal to clock the first DFF. The series of 6 DFF provides the desired division of 64 and an inverter is used as a buffer at the output to avoid loading issues.

According to simulation, this circuit takes a 10 mV_{PP} differential sinewave at 500 MHz and outputs a single ended digital square wave from 0 V to V_{dd} at 7.8125 MHz, 1/64 the original frequency. The testbench and simulation output are shown in Figures (2.17) and (2.18).

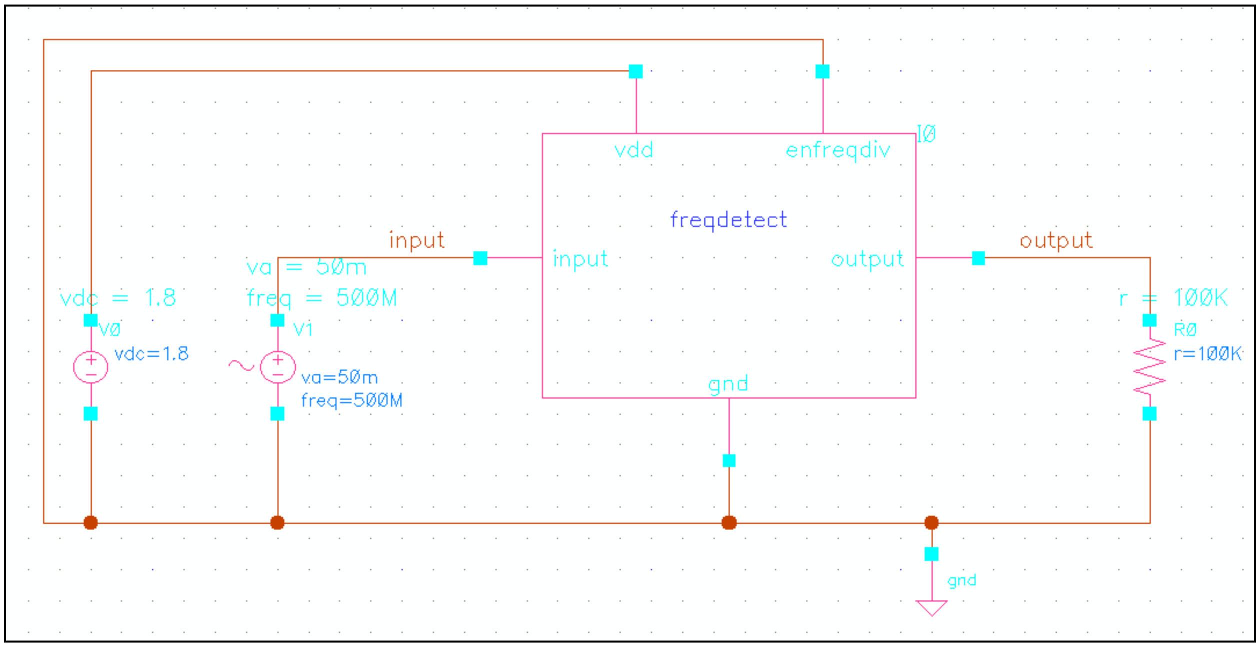


Figure 2.17 –Frequency Divider Testbench

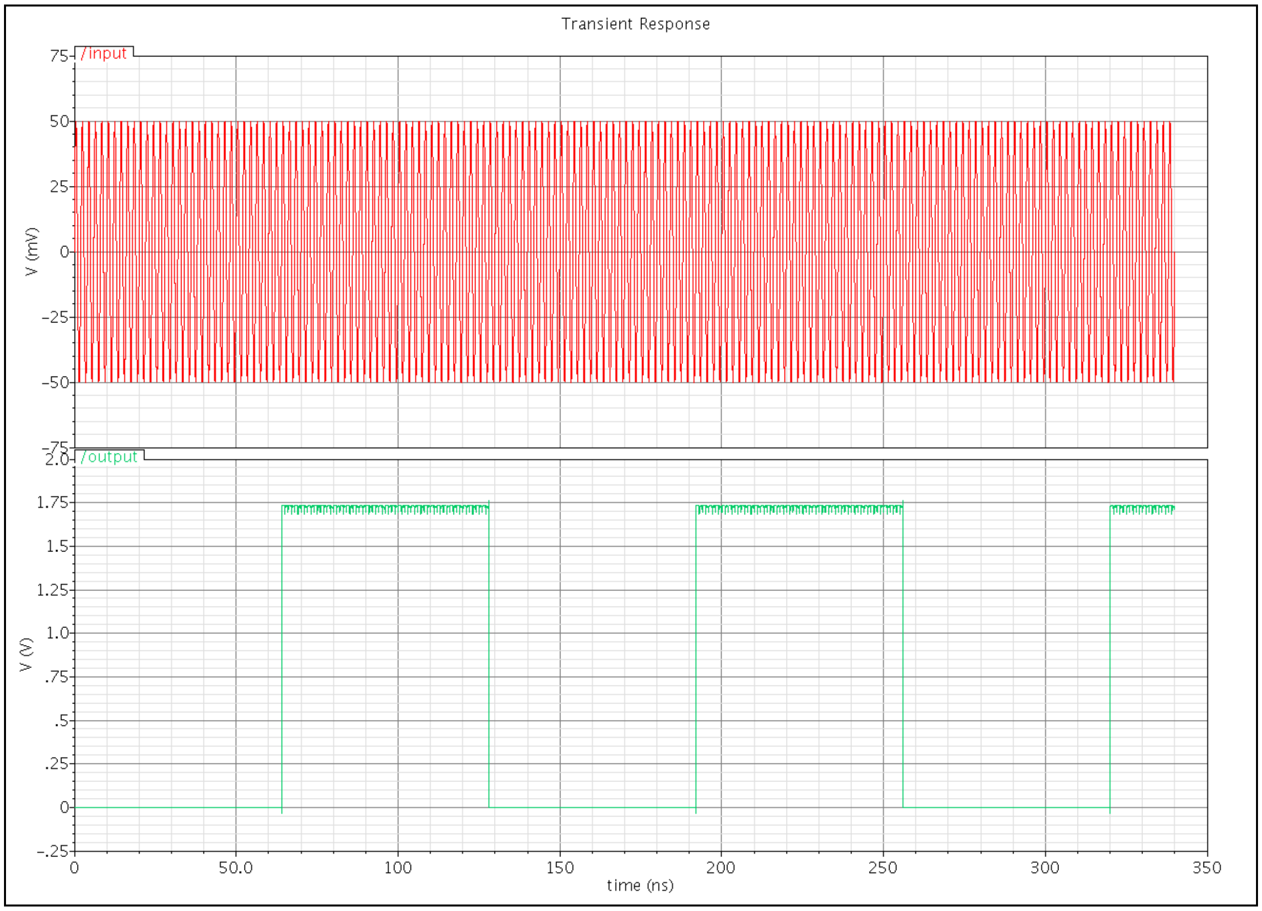


Figure 2.18 –Frequency Divider Simulation Output

The digital circuitry used in the new frequency divider design has the potential couple into the low-level circuits within the die and needed a bypass capacitor, shown in the above design, to smooth out any spikes in the power line. To ensure the cap was large enough to protect the circuitry and keep the power input constant the circuit test bench was set up with a 1H inductor in the power line to so that the circuit couldn't draw current from the supply immediately. Then the simulation was run to see if the circuit functioned correctly. The size of the bypass cap was increased until the circuit maintained normal operation even with the inductor in the power line. The testbench and the first part of the circuit in Figure 2.16 are shown in Figures 2.19 and 2.20 followed by two output simulations. The first simulation output in Figure 2.21 shows the circuitry performance with the bypass capacitor too small and the second simulation output in Figure 2.22 shows the circuit performance with the bypass capacitor sufficiently large.

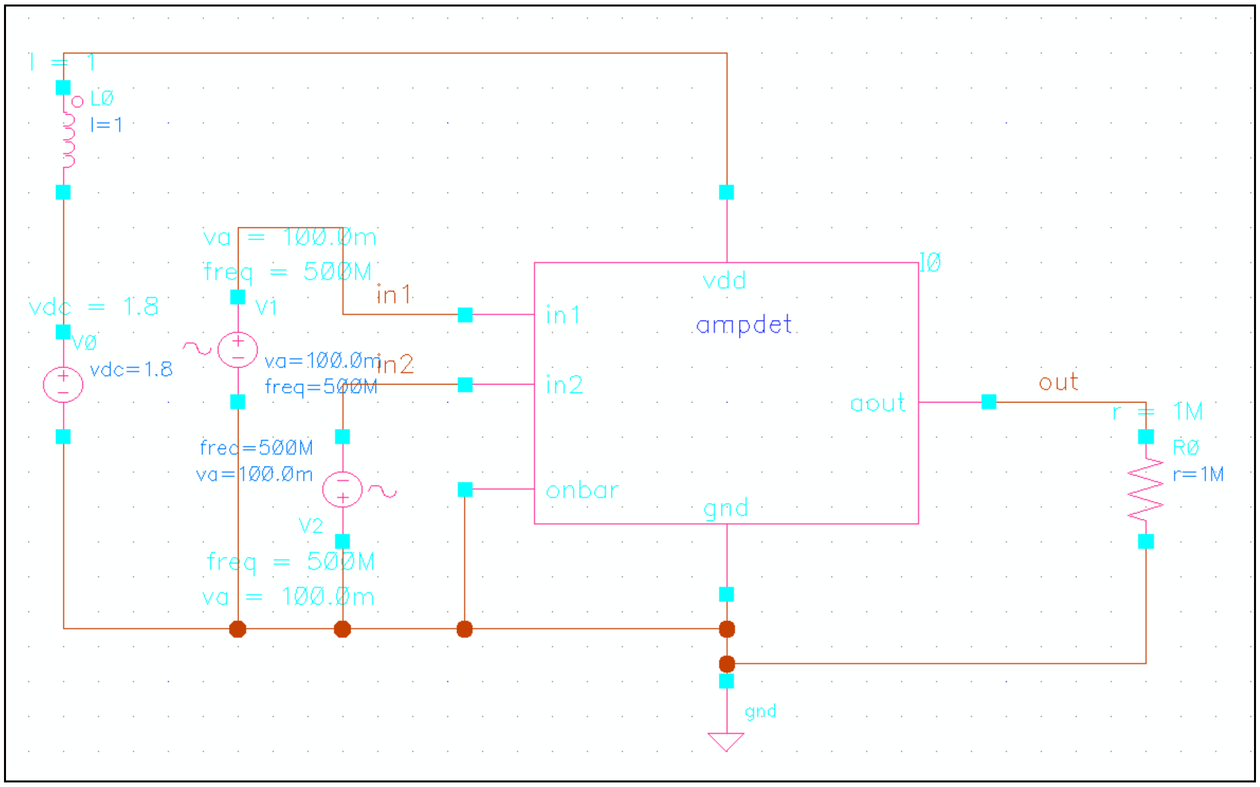


Figure 2.19 –Frequency Divider Testbench to Test Bypass Capacitor

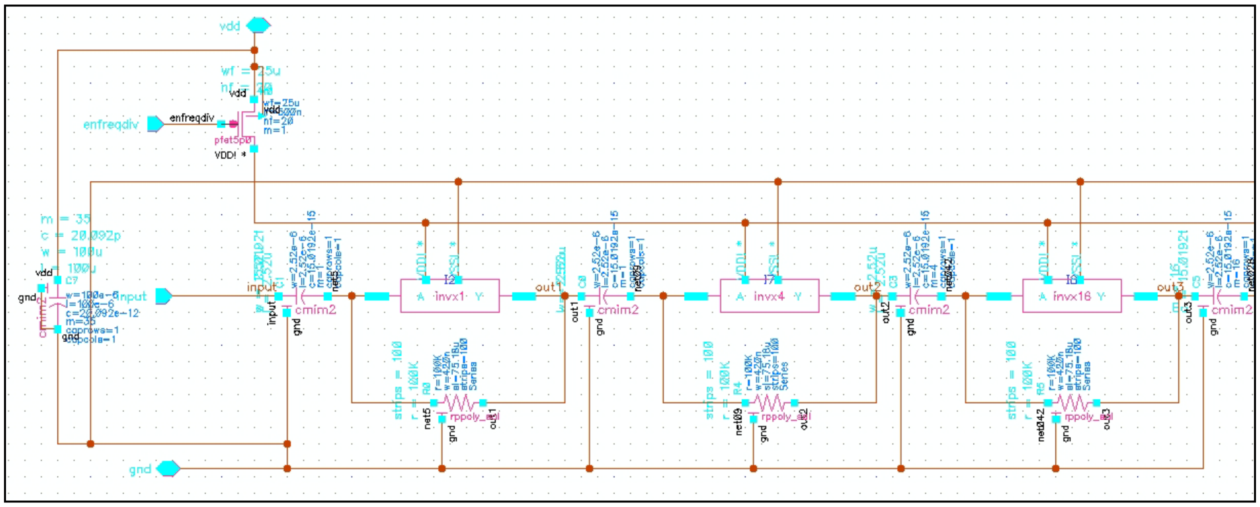


Figure 2.20 –Frequency Divider Circuit Narrowed View

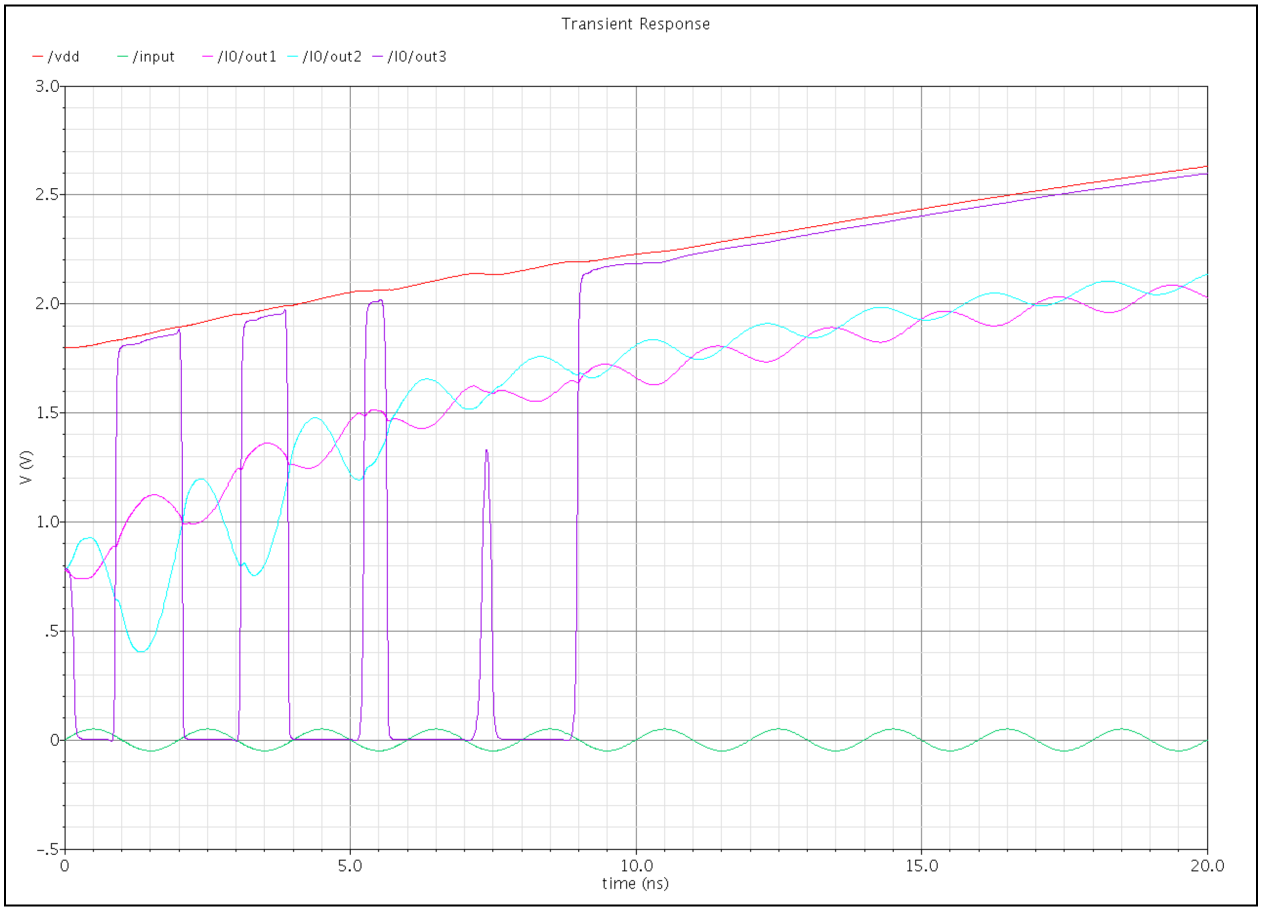


Figure 2.21 –Frequency Divider Circuit to Test Bypass Capacitor Simulation Output with Insufficiently Large Bypass Capacitor

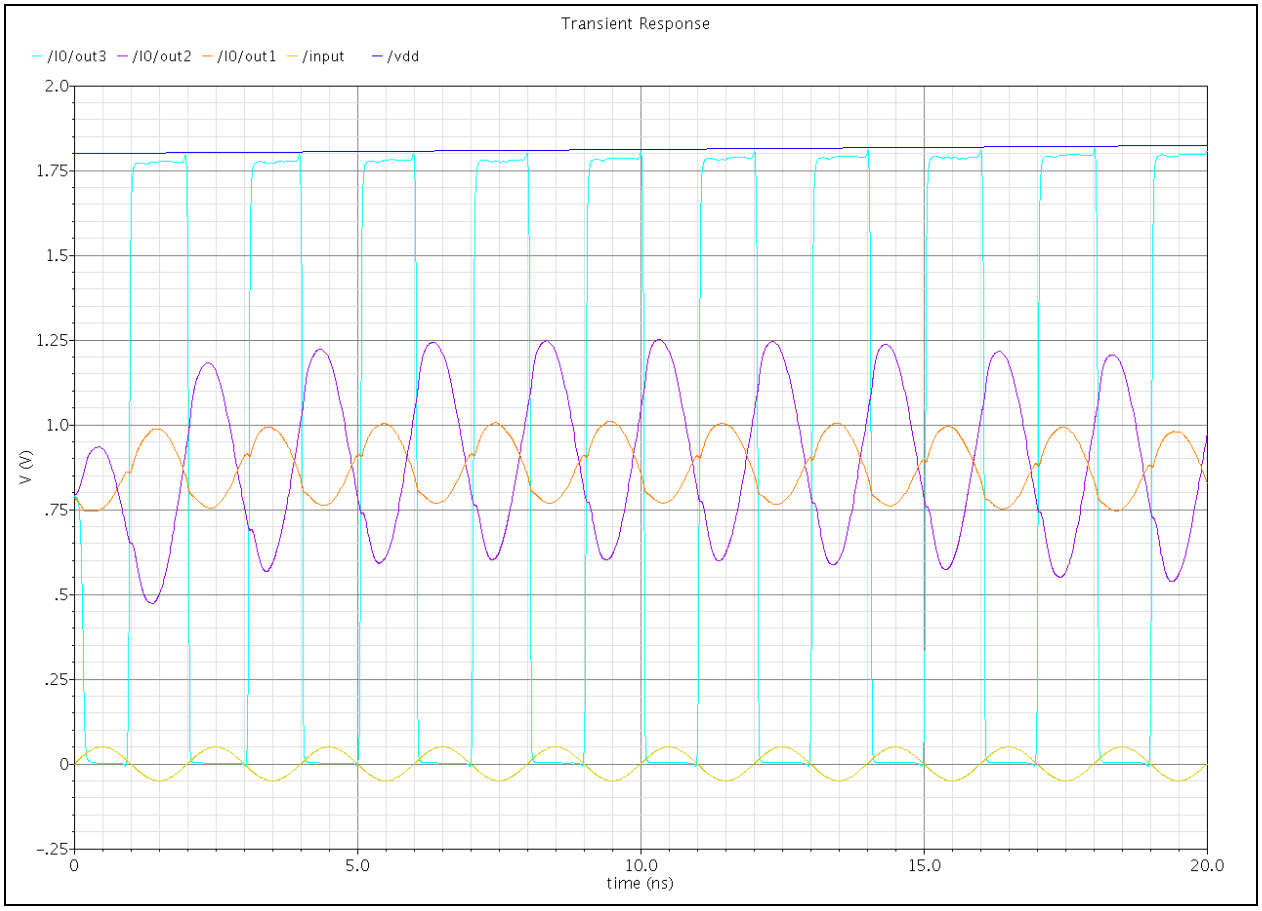


Figure 2.22 –Frequency Divider Circuit to Test Bypass Capacitor Simulation Output with Correctly Sized Bypass Capacitor

As shown by the simulation outputs the final bypass cap of about .7nF was large enough to assure smooth and stable operation.

2.4.2 Amplitude Detector

The last circuit discussed in detail is the amplitude detection circuitry. During simulation a problem with the basic design was discovered. In the previous filter small signals were never input to the amplitude detector due to problems with the frequency detector. As a result the problem in the circuit design was never noticed. At this time a full solution to this problem is not known, but the design flaw is detailed briefly here to document the problem to provide future designers a basis to begin designing upon. The amplitude detection circuit as currently designed is shown in Figure 2.23.

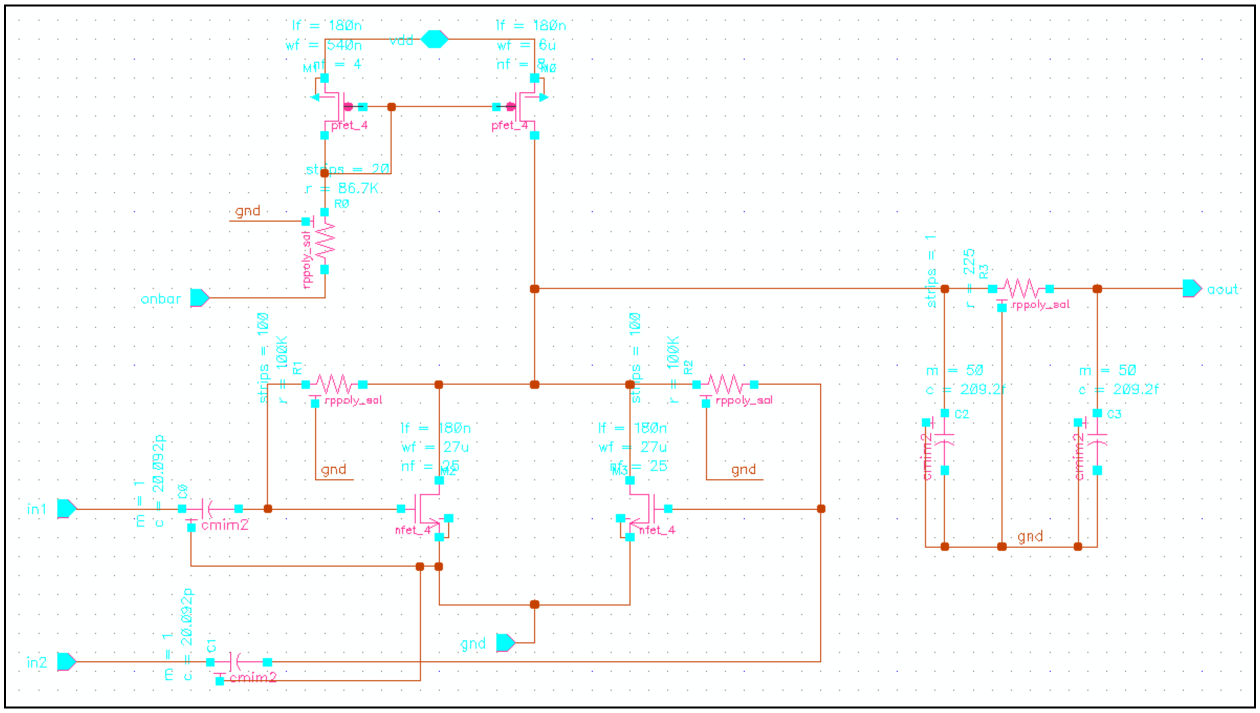


Figure 2.23 –Amplitude Detector Circuit

This circuit sets up a bias current using the current mirror with the PFETs at the top of the circuit. The input signal is driven differentially through AC coupling capacitors to the input. These capacitors have a small impedance relative to the FETs and biasing resistors connected to the signal at the input. When there is a sufficiently large AC signal applied at the input, the core FETs turn on shorting the drain voltage to ground and the voltage at the drain of both FETs drops. Good sensistivity is ensured by biasing the core FETs close to their thresholds so that only a small voltage is needed to turn them on. The output stage is a resistor and two capacitors in a low pass configuration to filter the output signal. Ideally, a stable DC voltage is output which drops quickly and significantly with the application of an input signal. During the design and simulaiton of this circuit a problem was discovered with the design as illustrated in Figures 2.24 and 2.25.

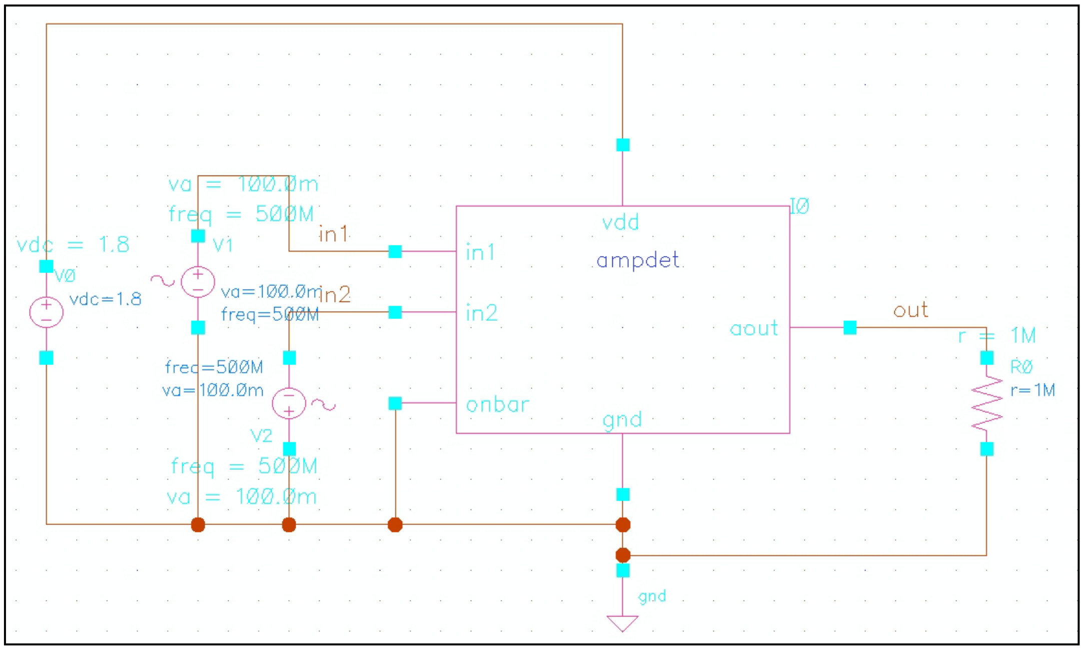


Figure 2.24 –Amplitude Detector Circuit Testbench

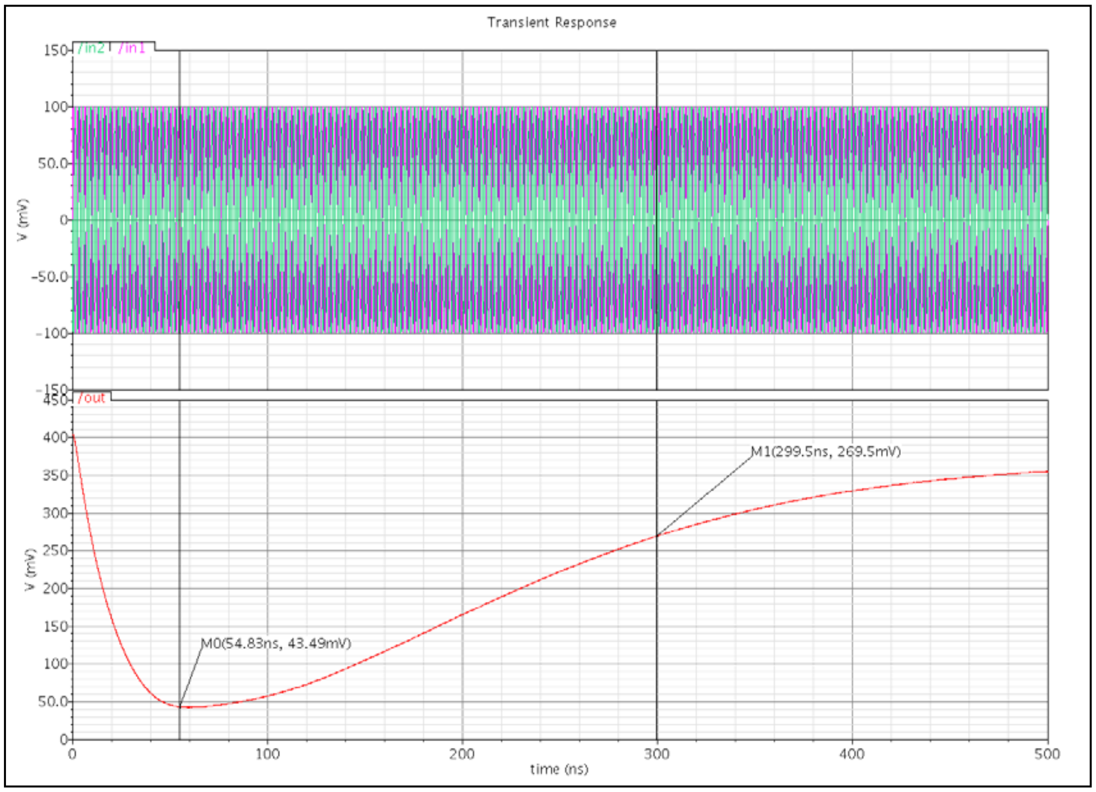


Figure 2.25 –Amplitude Detector Simulation Output

As desired, this circuit produces a sharp decrease in the output voltage level in direct proportion to the amplitude of the input signal. However the output simulation shows that after the voltage drops sharply it slowly increases. This behavior was determined to be caused by the time constant created by the resistors biasing the core FETs and the input capacitors. The problem occurs because when one of the FETs is turned on, the voltage drops as the input signal is still being applied, but the currents charging the input capacitor are not symmetrical. With the FET on, a larger C_{GS} is created, and then when it is off during the subsequent half of the cycle of the input, the C_{GS} is no longer the same value. As a result the input coupling capacitor stores up a charge and slowly pushes the output voltage up again. This issue will need to be resolved before the design is fabricated.

The rest of the circuits in the Q-enhanced filter were very similar to the original design and are omitted in the body of this text. However for clarity and documentation, these circuits are included in the appendix A where they are explained briefly and shown with their simulations.

Chapter 3 - Supporting Hardware and Software

3.1 Previous Work

In the thesis preceding this work, C based support code was written to for a dsPIC64FJ802 microcontroller to program and control the Q-enhanced filter. The microcontroller sent a control word, 64 bits in length, to the filter's serial to parallel register. Additionally, C# code was used to create a GUI that allowed the user to interface with the filter. Two versions of the C code on the microcontroller were developed: an automated, temperature stable algorithm that took user settings and could achieve bandwidths of 20MHz-5MHz, and a manual algorithm that allowed the user to manually set all the bits of the control word. As a basis to understand the code modifications implemented and recommended in the future, the next section briefly reviews the automated algorithm with the inclusion of resistance tuning.

3.2 Top-level Code Implementation

The automated algorithm worked as shown in Figure 3.1. The first step in the algorithm is setting all controls based on user inputs and initializing all additional settings. Next, the algorithm tunes the front-end frequency, then the back-end frequency, sets the coupling controls, sets the resistance tuning, waits the designated time interval and iterates. The only change at this top level of the algorithm is the addition of the resistance tuning. The information for modifying the code to implement this new hardware is detailed in section 3.3.2. The other additions to the algorithm occur within the front-end tuning and back-end tuning blocks. These changes are explained in section 3.3.1.

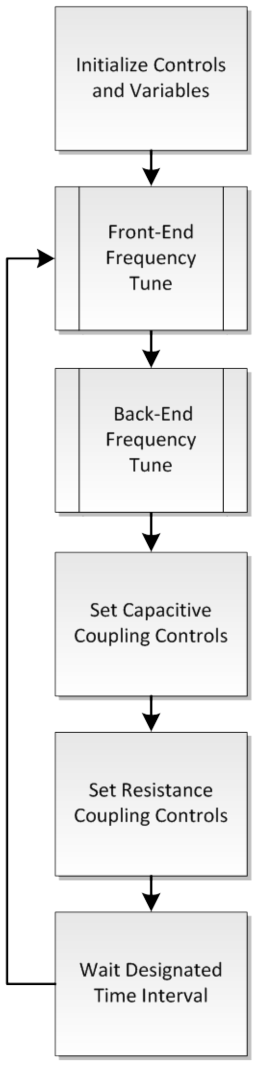

Initialize Controls and Variables
Front-End Frequency Tune
Back-End Frequency Tune
Set Capacitive Coupling Controls
Set Resistance Coupling Controls
Wait Designated Time Interval

Figure 3.1 – Top Level Flowchart of Tuning Algorithm

3.3 Software Additions

The first set of changes to the C code are additions implemented to improve the tuning algorithm by using the fine resolution frequency and q-enhancement controls in the current filter design. The second set of additions are needed to prepare the current code for the addition of the resistance tuning capability. The next section on fine tuning explains the changes to the code and documents the resulting improvements in the automated algorithm. The following section on resistance tuning will describe the suggested implementation and hardware specifics needed to implement passband symmetry control. Additionally, each of these sections enumerates the additions to the C code and C# code that are needed.

3.3.1 Fine Tuning

3.3.1.1 Implementation

It was a long time goal of this work to achieve finer frequency tuning accuracy and smaller bandwidth capability. In preparation for this goal external DACs were already implemented on the circuit board with the microcontroller. These DACs were intended to generate the necessary analog voltages to the pins on the filter which controlled the analog frequency tuning and Q-enhancement controls, but were not tested and exercised in previous work [2]. The analog circuits the fine-tuning explained in this section uses are essentially identical to those in Figures A.17 and A.19. Figures 3.2 through 3.5 show the flowcharts for the modified automated algorithm and are explained below.

The modified frequency tuning algorithm is shown in Figure 3.2 is identical to the flowchart in [2], except for the addition of fine tuning which uses the analog controls, implemented immediately after the course tuning which uses the digital controls. The ‘get-frequency’ flowchart in Figure 3.3 for the front and back ends is unchanged. However the ‘find-critical-oscillation’ flowchart in Figure 3.4 now includes the fine Q-enhancement tuning. The last flowchart in Figure 3.5 is entirely new and documents the logical flow of the fine tuning algorithm.

The fine tuning portion of the algorithm functions very similarly to the course tuning. First, the fine tuning determines whether the current frequency is higher or lower than desired. Second, the algorithm iterates to find the first analog setting to set the current frequency equal to the desired frequency. Those analog settings are stored and then further incremented until the frequency no longer equals the desired frequency. Those analog settings are compared to the stored settings and the analog controls are set centered between them to achieve the closest frequency to the desired frequency as possible.

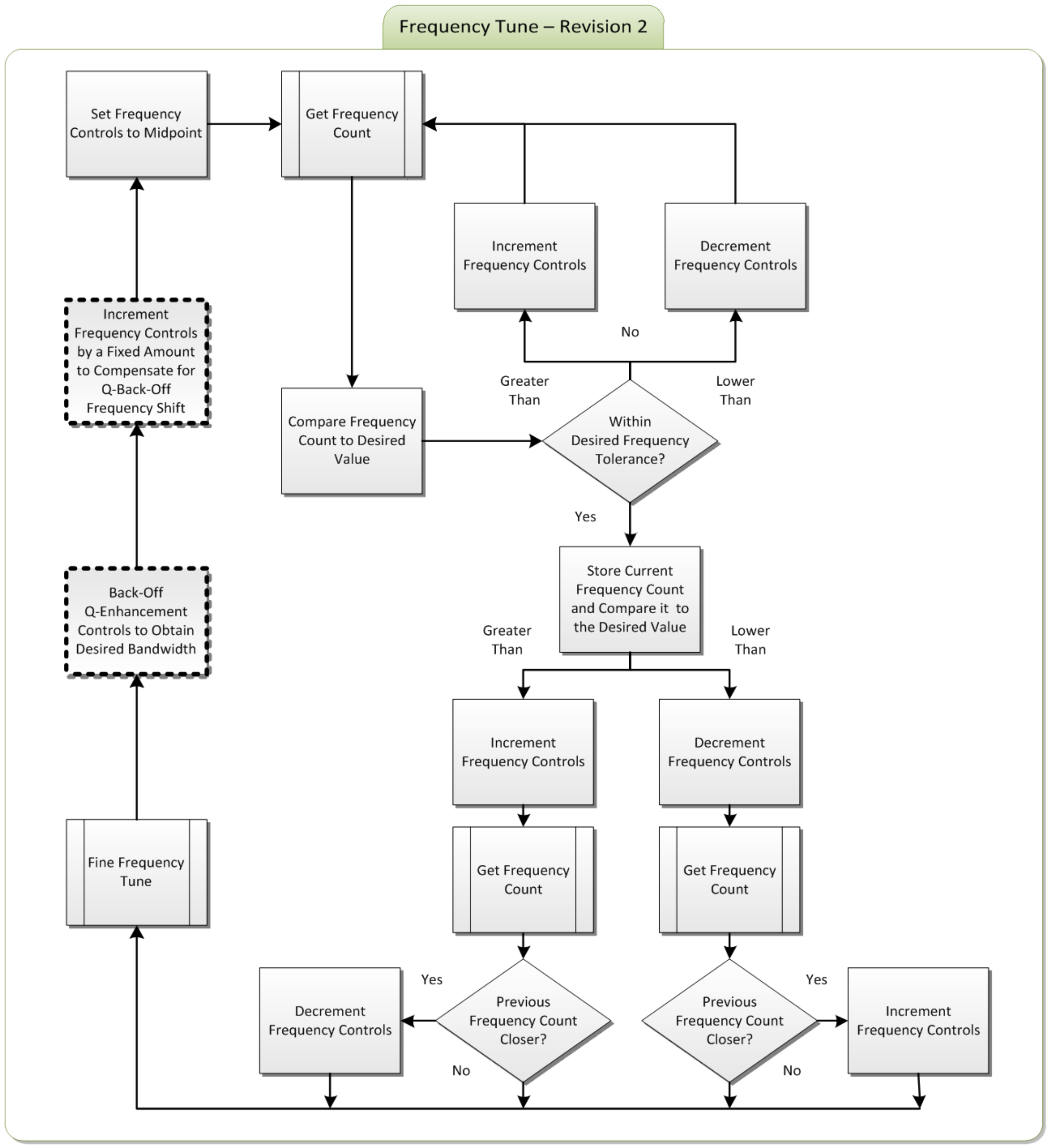

Frequency Tune – Revision 2
Set Frequency Controls to Midpoint
Get Frequency Count
Increment Frequency Controls
Decrement Frequency Controls
No
Greater Than
Lower Than
Increment Frequency Controls by a Fixed Amount to Compensate for Q-Back-Off Frequency Shift
Compare Frequency Count to Desired Value
Within Desired Frequency Tolerance?
Yes
Back-Off Q-Enhancement Controls to Obtain Desired Bandwidth
Store Current Frequency Count and Compare it to the Desired Value
Greater Than
Lower Than
Increment Frequency Controls
Decrement Frequency Controls
Fine Frequency Tune
Get Frequency Count
Get Frequency Count
Decrement Frequency Controls
Yes
Previous Frequency Count Closer?
No
Previous Frequency Count Closer?
Yes
No
Increment Frequency Controls

Figure 3.2 – Frequency Tuning Algorithm Flowchart: Revision 2

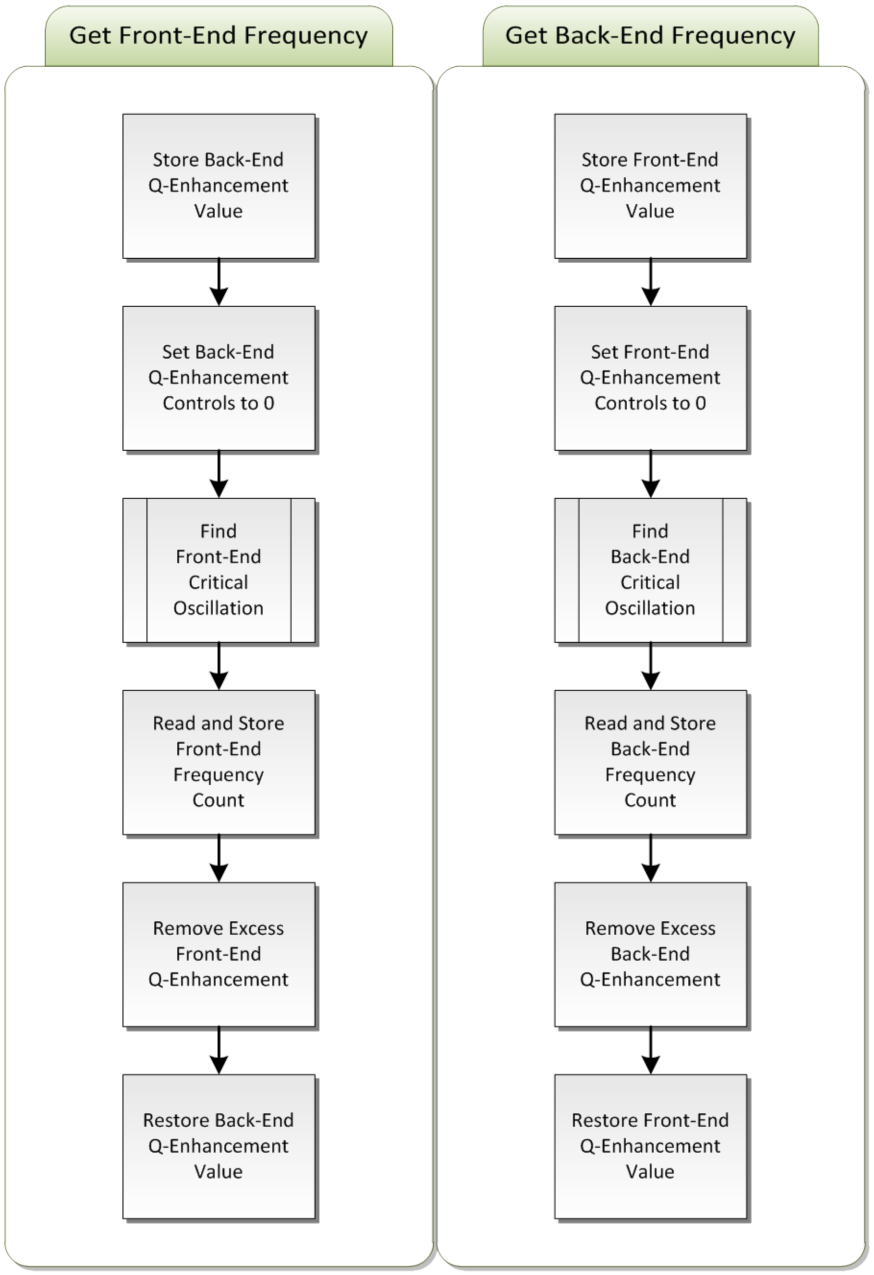

Get Front-End Frequency
Store Back-End Q-Enhancement Value
Set Back-End Q-Enhancement Controls to 0
Find Front-End Critical Oscillation
Read and Store Front-End Frequency Count
Remove Excess Front-End Q-Enhancement
Restore Back-End Q-Enhancement Value
Get Back-End Frequency
Store Front-End Q-Enhancement Value
Set Front-End Q-Enhancement Controls to 0
Find Back-End Critical Oscillation
Read and Store Back-End Frequency Count
Remove Excess Back-End Q-Enhancement
Restore Front-End Q-Enhancement Value

Figure 3.3 – Get Frequency Algorithm Flowchart (Used with Permission [2])

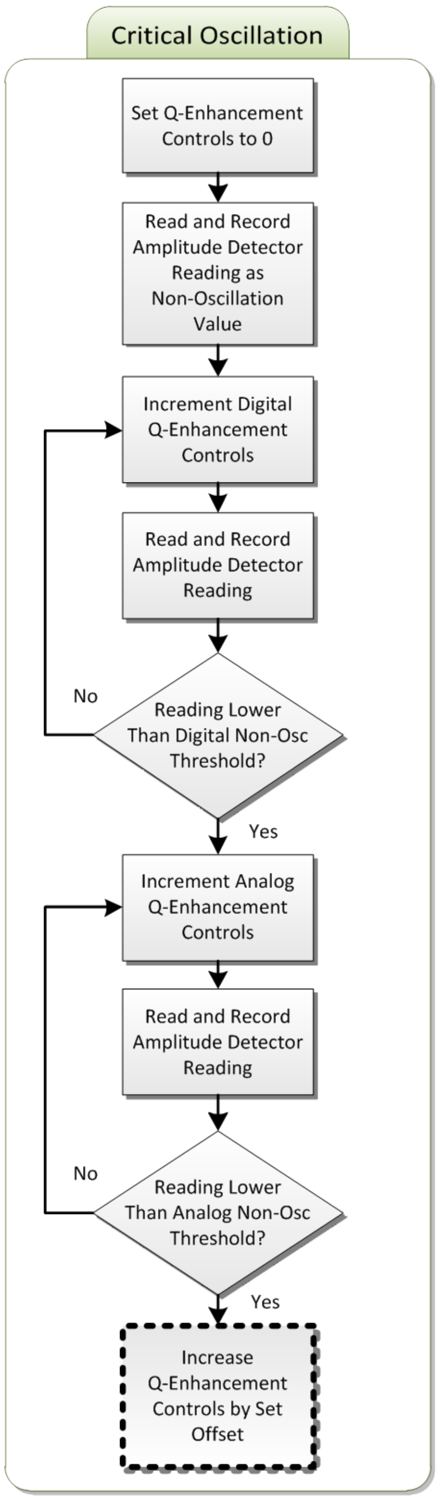

Critical Oscillation
Set Q-Enhancement Controls to 0
Read and Record Amplitude Detector Reading as Non-Oscillation Value
Increment Digital Q-Enhancement Controls
Read and Record Amplitude Detector Reading
No
Reading Lower Than Digital Non-Osc Threshold?
Yes
Increment Analog Q-Enhancement Controls
Read and Record Amplitude Detector Reading
No
Reading Lower Than Analog Non-Osc Threshold?
Yes
Increase Q-Enhancement Controls by Set Offset

Figure 3.4 – Find Critical Oscillation Flowchart

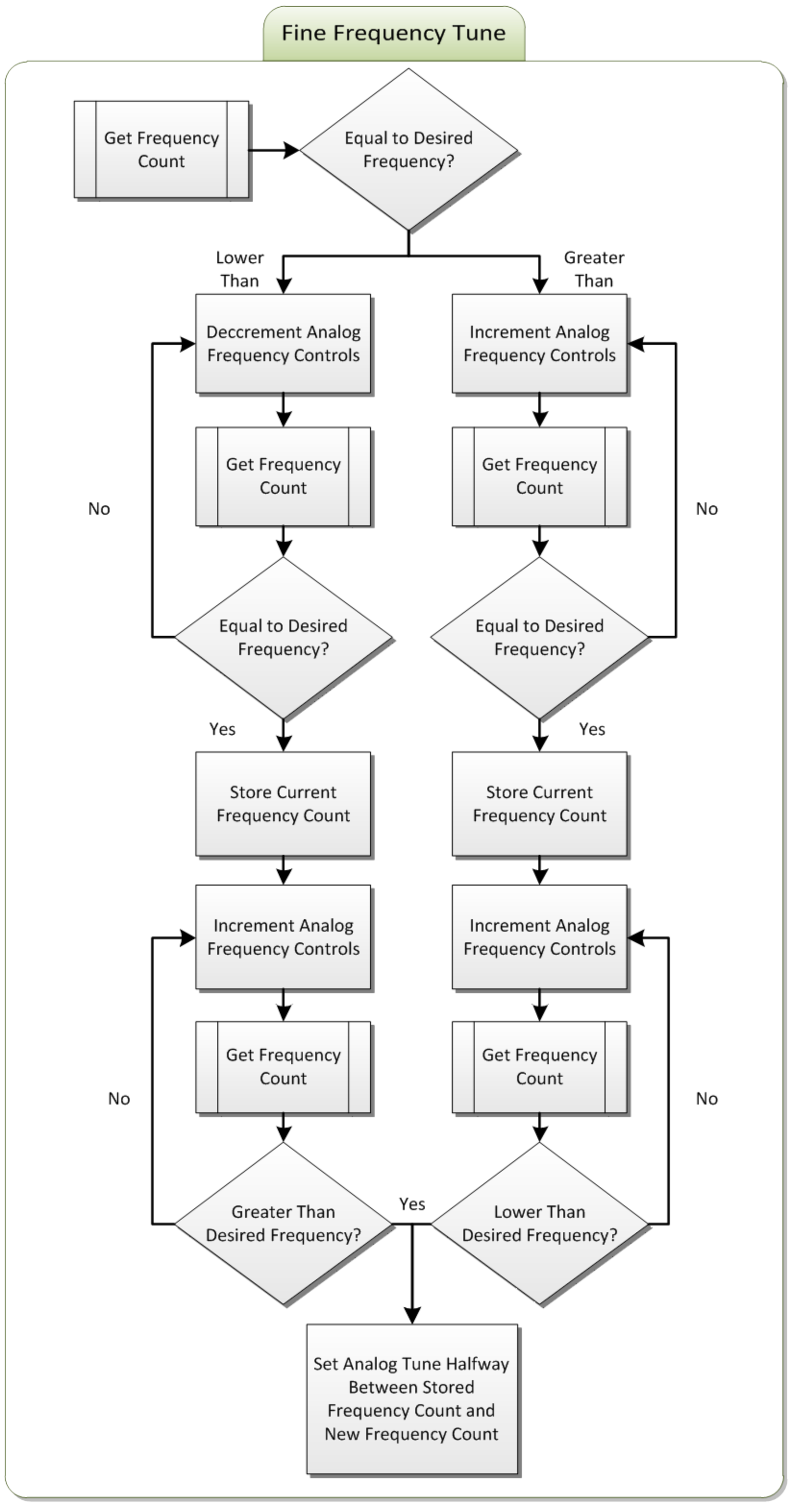

Fine Frequency Tune
Get Frequency Count
Equal to Desired Frequency?
Lower Than
Greater Than
Deccrement Analog Frequency Controls
Increment Analog Frequency Controls
Get Frequency Count
Get Frequency Count
No
No
Equal to Desired Frequency?
Equal to Desired Frequency?
Yes
Yes
Store Current Frequency Count
Store Current Frequency Count
Increment Analog Frequency Controls
Increment Analog Frequency Controls
Get Frequency Count
Get Frequency Count
No
No
Greater Than Desired Frequency?
Yes
Lower Than Desired Frequency?
Set Analog Tune Halfway Between Stored Frequency Count and New Frequency Count

Figure 3.5 – Fine Tune Flowchart

__3.3.1.2 Results__

The fine-tuning code addition improved the algorithm's performance. Without fine tuning some settings at bandwidths on the order of 5MHz exhibited variation in gain and bandwidth in the passband from one tuning iteration to the next. When the filter was programmed and the automatic tuning algorithm run, the settings that were chosen by the tuning algorithm were output to the GUI. These outputs showed a change ±1 in the digital q-enhancement settings when the gain in the passband varied. This indicated that the tuning algorithm needed additional precision to avoid fluctuation between two values in the algorithm. Bandwidths below 5MHz were not achievable without producing unstable outputs from the filter.

Figure 3.6 shows the passband variation caused by this quantization error in the tuning algorithm. This behavior was captured by setting the first trace of the spectrum analyzer to capture the maximum value of the passband and the second trace to capture the minimum value of the passband. This variation between tuning iterations was eliminated when the fine tuning code was implemented.

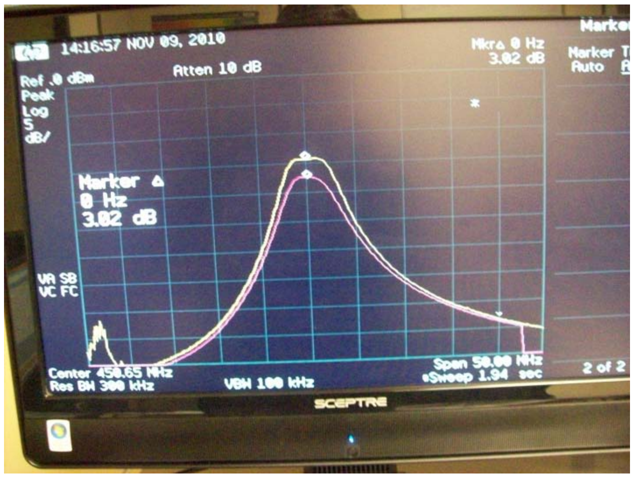


Figure 3.6 - Passband Variation without Fine Tuning

After the fine tuning was implemented, a temperature stable bandwidth of 2.5MHz with a center frequency of 450 MHz, a fractional bandwidth of about .6%, was achieved. Figure 3.7 shows a screen capture of the filter tuned to this narrow bandwidth and the settings which achieved this passband shape are shown in Table 3.2. This bandwidth and passband shape was maintained by the algorithm which included the fine tuning when heated from 20ºC to 75ºC.

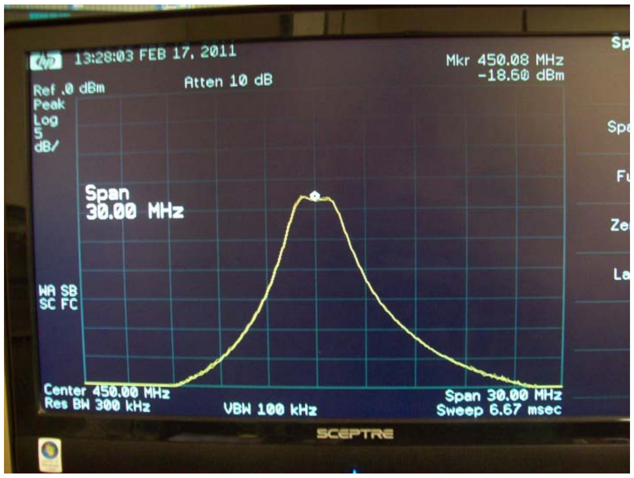


Figure 3.7 – 2.5 MHz Bandwidth Passband with Fine Tuning

Algorithm Settings		
Bandwidth	5 MHz	2.5 MHz
Center Frequency	450 MHz	
Frequency Tolerance	.3 MHz	
AD Threshold 1	3	2
AD Threshold 2	3	3
Q-Offset	2	2
Q-Back-Off	2	1
F-Offset	2	2
Capacitive Upper Tuning	8	5
Capacitive Lower Tuning	6	6
Capacitive UFLB Tuning	0	0
Capacitive LFUB Tuning	0	0

Table 3.1 – Fine Tune Settings for 2.5 MHz Bandwidth

These settings are dependent on the resistors used to cancel passband asymmetries and the Q of the on chip inductors. These settings may need to be varied ±1 if one of these variables has been changed. The addition of the more precise controls suggested in the future work should help to mitigate uncertainty in these settings.

3.3.2 Resistance Tuning

3.3.2.1 Implementation

The resistance tuning code needs to be developed to utilize resistance tuning controls described in section 2.3.1.2.2. This code can't be fully tested without the new filter design, but many of the necessary additions to the code-base are clear. Some of the modifications needed are detailed below.

3.3.2.2 Control Word and Passband Controls

The filter is programmed by the microcontroller via SPI communication. The current code sends a 64 bit word to the filter. The new filter will have an additional 28 bits of data to set the resistance tuning circuitry documented in section 2.3.1.2.2. The full 96 bit control word that the serial to parallel register will need from the microcontroller is shown in table 3.3. 'FENDCON' and 'BENDCON' are the control bits for the front-end and back-end frequency tuning, the amplitude detector and frequency divider. 'CAPCON1' and 'CAPCON2' are the control bits for the capacitive coupling circuits. 'RESCON1' and 'RESCON2' are the control bits for the passband asymmetry neutralization. RESCON1 and RESCON2 are unimplemented at the time of this thesis's publication.

FENDCON	BENDCON	CAPCON1	CAPCON2	RESCON1	RESCON2
MSB					LSB

Table 3.2 – Word Sent from Microcontroller to Filter

The resistance value needed to cancel asymmetries due to unwanted inductive coupling in the LC tanks, calculated in section 2.3.1.2.1, is shown again in (3.1) and then expressed in terms of coupling, k, and the Q of the filter.

$$\frac{-1}{R} = \frac{-2{X_L}^2 R_s k}{\left(R_s^2 + {X_L}^2\right)^2 + 4{X_L}^2 R_s^2} = \frac{-2Qk}{R_s((1+Q^2)^2 + 4Q^2)} = \frac{-2Q^2 k}{2\pi f((1+Q^2)^2 + 4Q^2)} \quad \textbf{(3.1)}$$

since $R_s = {}^{X_L}/_{Q}$. Letting $Q \gg 1$, then $R \ll X_L$ and all R^2 go to zero and this equation can be

expressed as (3.2) where we assume the Q of the filter is dominated by the Q of the inductors.

$$\frac{1}{R} = \frac{k}{X_L}\frac{2}{Q} \xrightarrow{yields} R = \frac{QX_L}{2k} \quad \textbf{(3.2)}$$

Note this resistance value assumes that the non-symmetry in the capacitive coupling discussed in section 2.3.2 has been successfully mitigated by raising the resistor values between the inverter and the FET's they drive and the ***only*** resistance being canceled out by this tuning is due to the inductive coupling. However, these resistors can also compensate for a capacitor coupling non-idealities if needed; only the value will need to be modified.

The capacitive coupling is currently implemented as a manual input by the user. It's recommended that the resistance tuning be implemented the same way for testing. Eventually, when the filter is fully tested a bandwidth and center frequency could be chosen and the capacitive coupling and resistance tuning could be chosen based off of a lookup table. This would require knowing the coupling value between the inductors either by user input or hard coding.

Chapter 4 - Spurious Responses in Receivers

This chapter addresses the impact of filtering on receiver performance at an architectural level. As a basis for cogent discussion, receiver architectures are briefly enumerated and a superheterodyne architecture is explained in some detail as a foundation of further analysis. Next, receiver performance is discussed focusing on the intrinsic limitations imposed by a receiver's ability to avoid spurious responses. An overview of spurious responses is presented with a brief explanation of their origins in superheterodyne receivers. To observe these behaviors in receivers a novel method of measurement and receiver characterization is described. Using this evaluation method some commercial handheld radios, a software defined radio (SDR) and an integrated radio are characterized. Finally in Chapter 5, a modified receiver architecture using a tunable variable bandwidth Q-enhanced bandpass filter is presented and our testing procedure is used to verify the validity of this solution.

4.1 Introduction to Receiver Architectures

The two primary types of receiver architectures generally used in modern communication systems are direct conversion and superheterodyne. Direct conversion receivers offer some excellent performance advantages for wideband signals and have been almost universally adopted by the cell phone industry. However, superheterodyne architectures still dominate narrowband receiver technologies due to innate limitations in current technology and remain the architecture of choice for wide coverage designs such as spectrum analyzers and software defined radio front-ends. Superheterodyne receivers therefore may be favored in applications such as sensor networks and cognitive radio. As explained in the introduction receiver performance is a crucial issue in modern communication technology which limits spectra usage and defines the demands put on the communication infrastructure. Four architectures are addressed below and explained briefly. The various types of spurs and behaviors mentioned are explained in 4.2.

4.1.1 Direct Conversion

Direct conversion receivers use a 0 Hz IF frequency, converting the received signal directly to baseband. These architectures typically include a preselect filter and a LNA before the mixer. After the conversion to baseband lowpass filters are applied and amplification and demodulation is accomplished. A basic block diagram of this receiver architecture is shown in Figure 4.1.

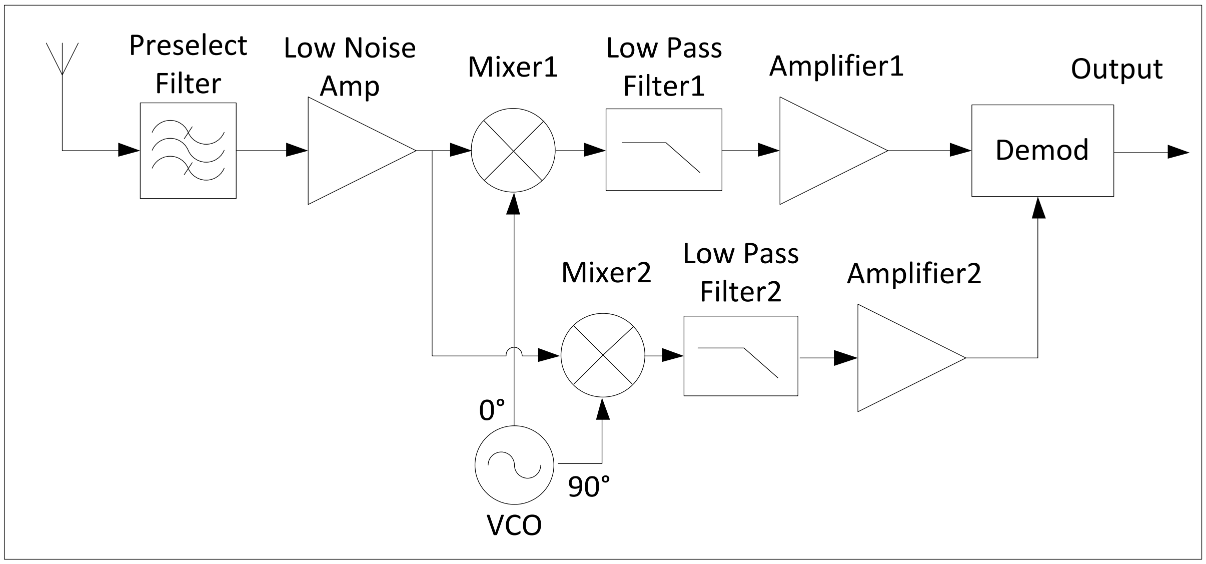


Figure 4.1 – Direct Conversion Receiver Block Diagram

This architecture offers some nice potential performance capabilities. The most obvious advantage of a 0Hz IF frequency is that there is no image frequency. Additionally the filter design complexity is reduced somewhat because only a good lowpass filter is required. However, this also means that the VCO is equal to the received frequency posing potential problems of f_{LO} feedthrough and crosstalk between nearby receivers. Another major problem is the DC offsets innate in any 0Hz IF design. Resolving this using an AC coupled system is made more difficult by the potential for very low frequency variations in the DC offset from variation in path or coupling. Finally, when implementing in a CMOS process especially, 1/f noise becomes a problem. For a narrowband radio, high 1/f noise requires excessive gain ahead of mixer in order meet receiver noise figure goals. Such broad-band high gain creates strong spurious responses when operating in a dense signal environment. So, while this architecture works very well for wideband reception which doesn't have demanding filter needs, narrow band applications are a different matter because they put highly demanding restrictions on the lowpass filter roll off.

4.1.2 Superheterodyne

Superheterodyne architectures are built around the central idea of converting the received RF frequency to a lower intermediate frequency (IF) for filtering, amplification and demodulation. These receivers typically include a preselect filter, a LNA, an image filter and a mixer. Sometimes the image filter is omitted and an image rejection mixer is used instead. After the signal is mixed to the IF, it is further filtered, amplifier and demodulated. A basic superheterodyne receiver is shown in Figure 4.2.

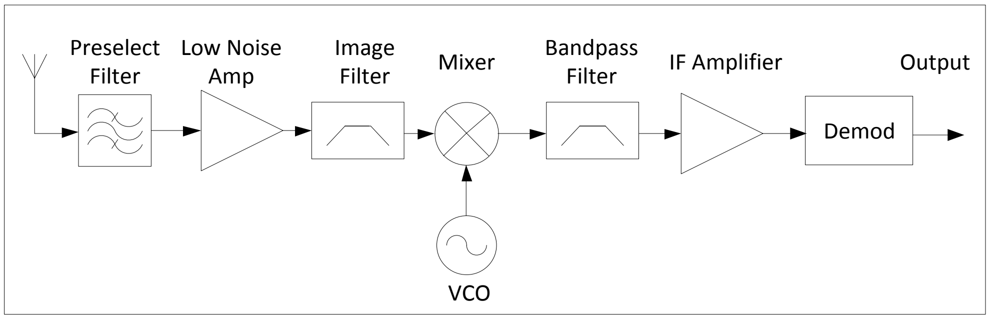


Figure 4.2 – Superheterodyne Receiver Block Diagram

There are many advantages to heterodyning the received signal. The narrowband filtering can be done at a lower fixed frequency reducing the design demands on the filters quality factor. The amplification can be split between two different frequencies reducing the potential for positive feedback. Additionally, the bulk of the gain can be provided at the lower IF frequency where amplification is easier to achieve. A significant drawback to this architecture is the image frequency inherent in the downconverting operation, even though image rejection has been well studied and is often successfully mitigated by image reject filters and/or image reject mixers. Another drawback is that the LO frequency used to downconvert the received signal will have some phase noise which adds into the mixed spectrum, but this is inherent in any frequency generation and conversion, including direct conversion applications. Also, if a phase locked loop is used to control the VCO, f_{LO} will have harmonics produced by the pulse train generated to vary the control voltage. However, despite these design challenges this architecture still provides the best performance for narrowband receivers since no other architecture sufficiently addresses the filtering, coupling and noise challenges innate in narrowband reception.

4.1.3 Multiple Conversion Superheterodyne

The superheterodyne receiver is often extended to double or triple conversion architectures to help minimize the image response and increase the receiver's spurious response avoidance. A basic dual conversion receiver is shown in Figure 4.3.

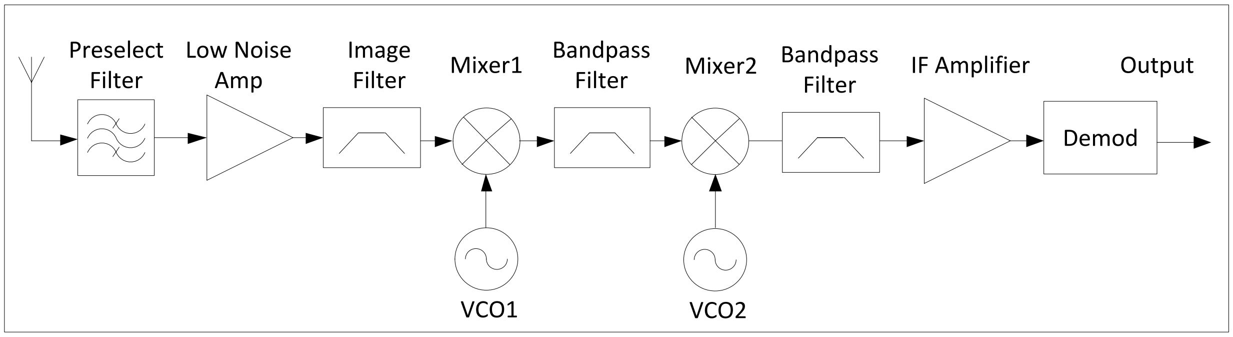


Figure 4.3 – Multiple Conversion Superheterodyne Receiver Block Diagram

This architecture might upconvert a signal to a first IF and then downconvert to the final IF, or down convert through two or more conversions to the final IF where the signal is demodulated. In any case, the use of additional IF through multiple conversions significantly reduces the problematic image frequency and theoretically helps reduce IF feedthrough as well.

4.1.4 Software Defined Radio

Software defined radios (SDR) are a completely different architecture. The potential flexibility available in a digital implementation is very attractive and many SDRs have been developed. However, the need for an analog front end is inescapable due to the demands of high frequency narrowband reception. A typical block diagram for a SDR is shown in Figure 4.4.

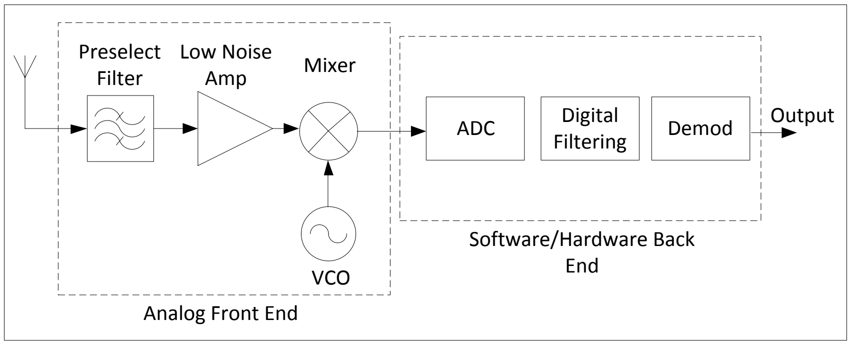


Figure 4.4 – Superheterodyne Receiver Block Diagram

As shown some type of preselect filter is typically included with an LNA and downconversion. Then the signal is sampled by the ADC, filtered and demodulated digitally. These receivers are very complex and a full analysis of their inner workings is beyond the scope of this work. They are introduced here as context for later discussions.

4.2 Spurious Responses in Receivers

Spurious responses, or ‘spurs’, are defined as frequencies the receiver responds to which are different than the frequency the receiver is tuned to receive. In general, the receiver will respond to any signal which mixes to or distorts the IF or its modulation. These spurs can occur through several mechanisms resulting from different aspects of the receiver architecture. These spurious responses often occur when 2nd or 3rd harmonics of a received signal (produced in the mixer or amplifiers preceding it when an input is sufficiently strong) mix to the IF with harmonics of the VCO or other pre-existing frequencies in the receiver. Spurious responses also occur when signals “feedthrough” due to inadequate filtering. Additionally, intermodulation may occur when two strong off-channel signals mix directly to the RF channel or the IF. It’s also noteworthy that some receivers may have a spurious response when tuned to receive specific frequencies when no external signal is applied, due to an internally generated signal at that frequency. These responses are referred to here as “birdies”.

To discuss spurs graphically Figure 4.5 is provided detailing the expected spectrums at each major node of a typical superheterodyne architecture. The architecture includes a frequency synthesizer with a TCXO and ADC and demodulation (DMOD) circuitry as these are quite common in these designs. This figure is meant to show how real world spectrums should behave in this architecture and assumes no spurious responses are occurring. The only non-idealities shown are noise, other frequencies in the received signal’s environment, LO feedthrough, close in synthesizer spurs around f_{LO} and the f_{LO} and f_{RF} products from the mixer.

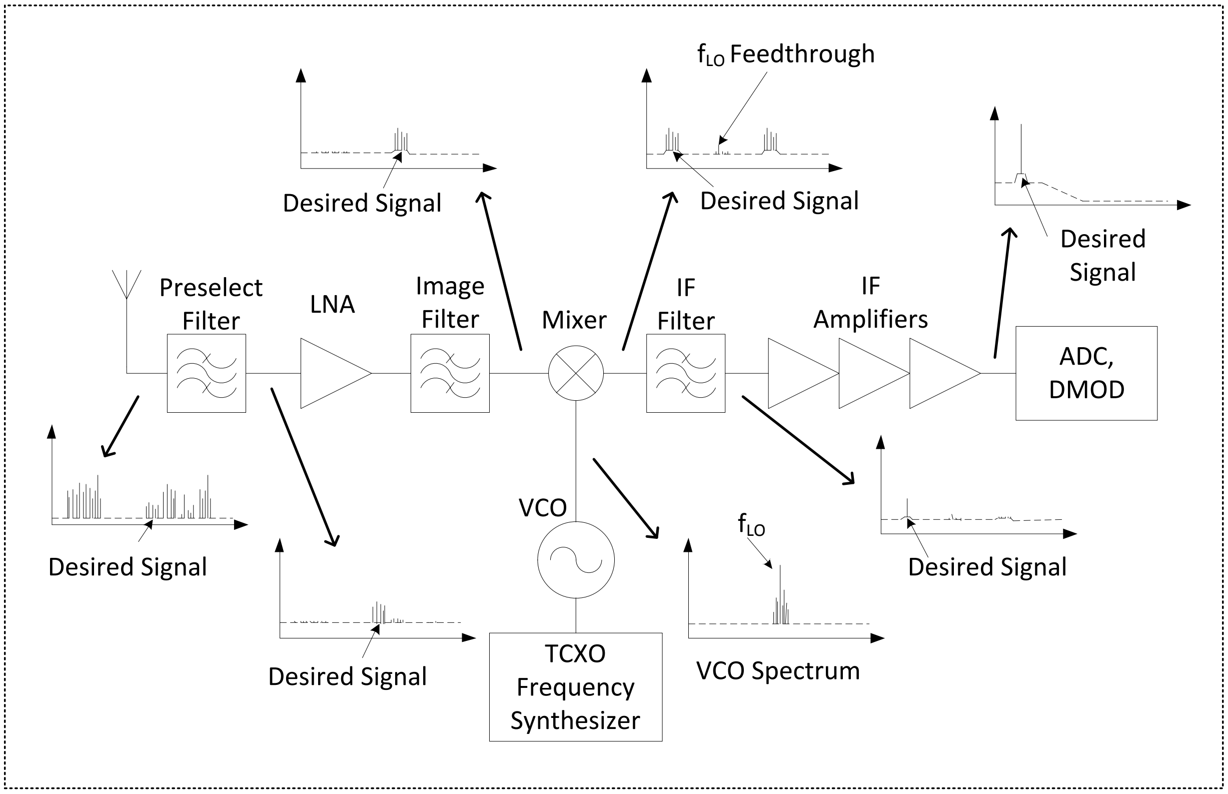


Figure 4.5 – Superheterodyne Receiver with Spectra

The first step in a superheterodyne receiver is the preselect filter. This filter ideally eliminates all frequencies outside of the bandwidth the receiver is designed to receive. The next step is low noise amplification to improve the noise figure of the receiver. The image reject filter shown here isn't always implemented in superheterodyne architectures, but it is shown here to emphasize the need to address the potential spurious response caused by the image frequency. Most superheterodyne receivers will implement an image reject filter or image reject mixer to mitigate the image frequency. Also, the noise itself becomes amplified so that the signals of interest are on a 'noise pedestal' following the image filtering. Next, the mixer uses f_{LO} from the VCO to downconvert the received signal. The VCO is driven by a frequency synthesizer using a PLL which causes some phase noise and close in spurs with f_{LO}. The spectrum after the mixer will contain the upconverted spectrum, some amount of 'LO feedthrough' and the desired downconverted spectrum. As long as these signals are not too strong and the IF is high enough, the IF filtering will ideally eliminate all but the signal at the IF, as shown in the spectrum after the IF filter. Last, the IF signal is amplified to ensure the demodulation circuitry has a strong signal to interpret.

4.2.1 Nonlinear Behaviors

4.2.1.1 Amplification

One of the most important problems in receivers is the nonlinear nature of active devices. Any active device which has a perfect sine wave as an input, such as an amplifier or mixer, will output a signal which has harmonics besides the fundamental with magnitudes depending on the design of the device and the power of the input signal. The linear region of the device is defined as the range of operation over which the output signal contains harmonics which are so small they are negligible. In the case that the device isn't operating in the linear region the nonlinear output will contain harmonics with significant amplitudes. The harmonics can be modeled mathematically by expressing the output signal as a summation of the input signal using a Maclaurin expansion as follows:

$$v_O = A_1 v_i + A_2 {v_i}^2 + A_3 {v_i}^3 + \ \dots \qquad \textbf{(4.1)}$$

Using trigonometric identities it's easy to see the frequency harmonics coming from the sinusoidal input evolving from this equation as follows:

$$v_O = A_1 V cos(w_O t) + \frac{A_2}{2} V^2 [1 + cos(2w_O t)] + \frac{A_3}{3} V^3 [3cos(w_O t) + cos(3w_O t)] + \ \dots \qquad \textbf{(4.2)}$$

when

$$v_i = V cos(w_O t) \qquad \textbf{(4.3)}$$

Any simple sinusoidal signal passing through a non-linear device such as mixers or amplifiers can be described this way. Looking at the above expression it's clear that an input signal of significant power will produce significant harmonics. So any strong signal which is in the bandwidth of operation, or a signal outside this bandwidth which the preselect filter fails to mitigate, can produce these harmonics. Additionally, the synthesizer f_{LO} signal will contain harmonics, or even if the LO is reasonably pure, the switching mixer used in nearly all designs implicitly introduces LO harmonics in the mixing process itself.

4.2.1.2 Mixer Spurs

All of these amplified signals and their harmonics have the potential to mix to the IF. In particular, the potential combinations of RF and LO signals which can mix to the IF band are predicted by the well-known equation:

$$f_{IF} = M f_{RF} + N f_{LO} \qquad \textbf{(4.4)}$$

Obviously, good filtering is crucial because any unwanted received or internally generated signal in the receiver is subject to this nonlinear behavior and could produce spurious responses if the signal is strong enough. Given this nonlinear behavior, good gain control is also important to minimize unnecessary creation of harmonics due to over amplifying a received signal. Figure 4.6 graphically depicts a potential development of this nonlinear behavior in a section of the superheterodyne architecture.

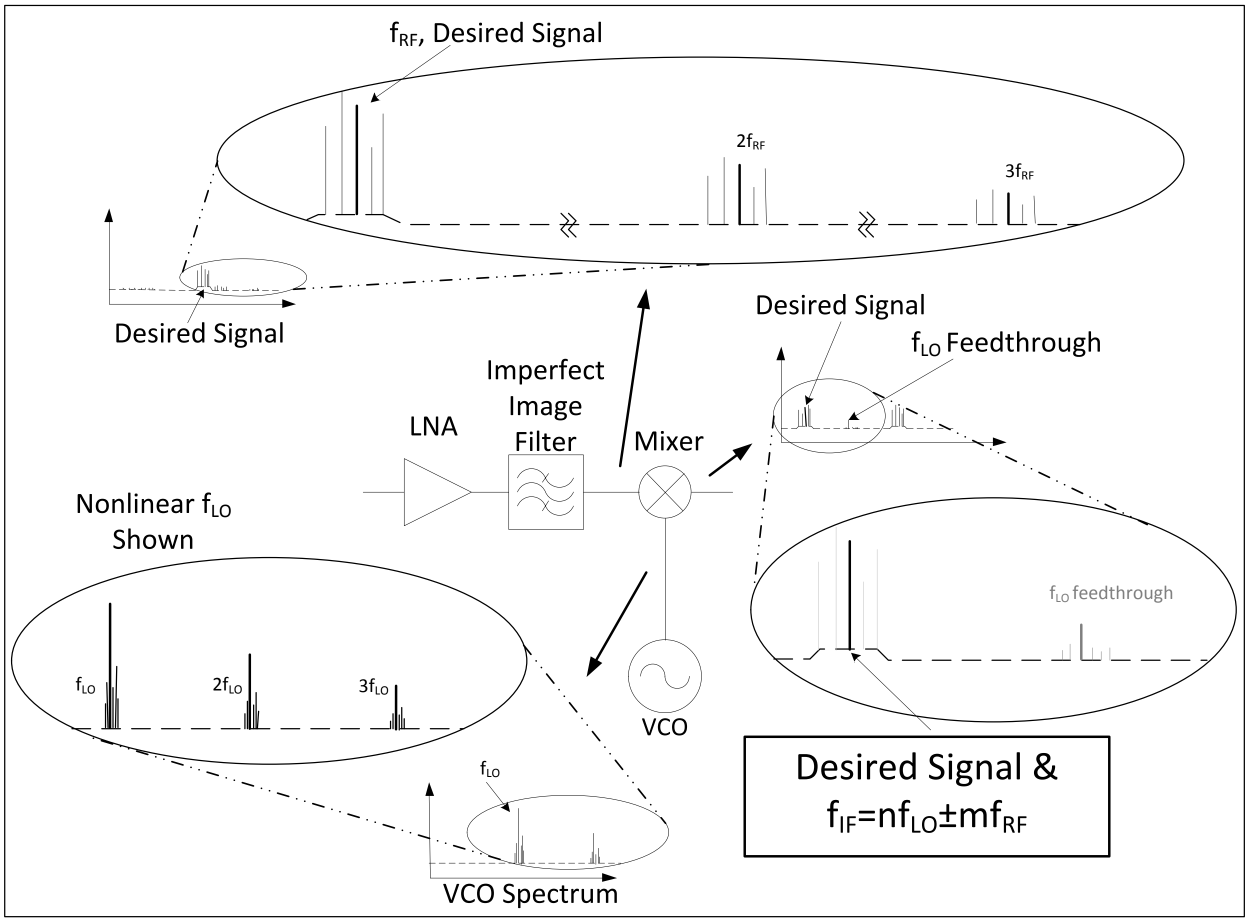


Figure 4.6 – Nonlinear Spurs

As shown in Figure 4.6 strong signals produce harmonics at regular intervals with decreasing amplitudes. The desired signal at f_{RF} mixes with f_{LO} as expected. However it is also possible that a signal near f_{RF} will have a second harmonic that differs from the second harmonic of f_{LO} by f_{IF} producing a signal at f_{IF} which will distort the desired signal. Also, any time a signal is a fractional value of f_{RF} and is strong enough to produce harmonics, the harmonic may reach the demodulation circuitry by mixing to the IF. We will refer to this as a sub-harmonic spurious response.

4.2.1.3 Intermodulation Distortion

Intermodulation distortion is another type of spurious response. However, this behavior differs from other spurious responses described above because this is a result of two or more signals at the input of the receiver interfering rather than one signal from the input mixing with a signal internally generated in the receiver. This is a particularly difficult problem because the spectrums in most environments on earth are full of strong signals at many frequencies. Signals which are strong enough to produce harmonics could self-mix or mix to the IF through the mixer. Intermodulation is an important issue in receiver performance, however due to time constraints this issue is not addressed in this work.

4.2.2 Spurs from Digital Synthesizers

Another major source of spurious responses in many receiver architectures today is digital synthesizers. Digital synthesizers are used because they provide precise fine resolution tuning of the VCO. A typical synthesizer includes a VCO, a temperature compensated crystal oscillator (TCXO), division and filtering circuitry and a phase-frequency detector (PFD). There are two major causes of spurious responses resulting from this circuitry: the TCXO and the pulsed output from the PFD. The TCXO causes spurs because it provides a strong signal as a reference which is often divided down to be used in the N synthesizer. The fundamental frequency of the TCXO, and any frequency it is divided down to, could be a strong signal within the receiver. These signals have the potential to mix with each other or unwanted incoming signals to the IF.

The other major source of spurs, the PFD, compares the divided TCXO and VCO outputs. Based on this comparison the PFD generates a varying voltage which is input to the VCO to keep the VCO locked on the desired frequency. This modulation of the VCO frequency creates spurs close in around the fundamental frequency that the VCO is producing. This combined spectrum is the f_{LO} input to the mixer. When a signal besides the desired RF enters the mixer and is strong enough it will combine with a TCXO spur or a close in spur of the f_{LO} and may mix to IF band as shown in Figure 4.7.

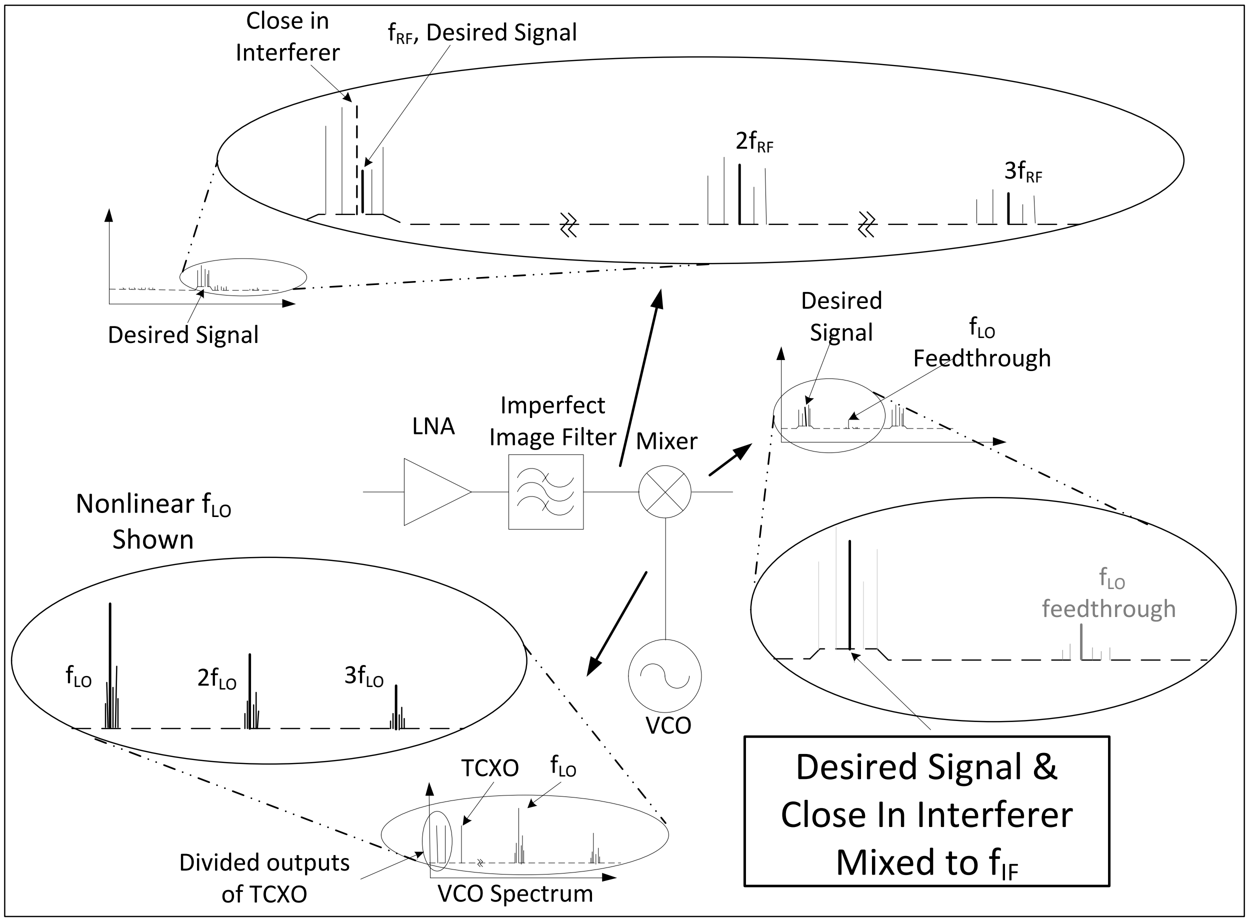


Figure 4.7 – Digital Spurs

In these two cases the root problem is the additional frequency components inherent in the frequency synthesizer design. The potential for either the TCXO or its subharmonics or the close in spurs of f_{LO} to mix with incoming signals is compounded by the nonlinear behavior or amplifiers and mixers. Each of these undesired signals could produce additional harmonics when amplified which also have the potential to mix to the IF.

4.2.3 Image Frequency

A well known major spur problem specific to superheterodyne architectures, results from a signal at the “image frequency”. For high side injection, the image frequency is above the f_{LO} signal by a frequency difference equal to the IF frequency placing it $2f_{IF}$ above the frequency like the receiver is tuned to. Conversely, if the receiver uses low side injection, the f_{LO} frequency is less than the desired RF frequency by the IF frequency and the image frequency is below the f_{LO} frequency by the IF frequency. In either case the difference between f_{LO} and f_{RF} and the image frequency is the IF and results in an unwanted signal at the IF if there is a signal present at this image frequency, f_{image}. Using symmetry the image frequency can be calculated using (4.5).

$$f_{Image} = f_{RF} \pm 2f_{IF} \tag{4.5}$$

where the sign of the addition is determined by the use of high or low-side injection respectively. Figure 4.8 shows the image frequency mixed to the IF when the receiver uses high side injection.

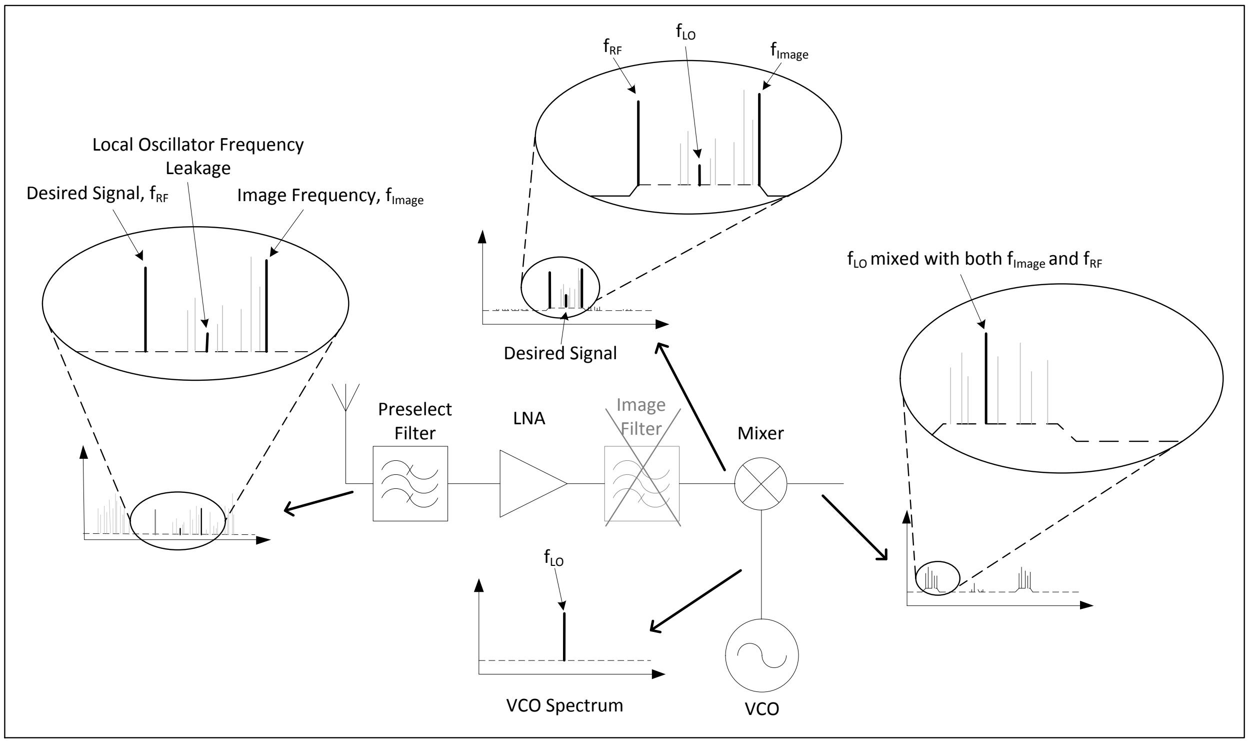


Figure 4.8 – Image Frequency

As shown the image frequency is located symmetrically about f_{LO} with respect to f_{RF}. Without the image filter the image frequency will mix to the IF if neither the preselect filter or the LNA have a narrow enough bandpass shape that the image frequency is suppressed.

4.2.4 1/2IF & 1/3IF Spur Frequencies

The ½ IF spur problem is a less known but also significant issue in superheterodyne receivers. In this case the 2nd harmonic of the LO frequency and the 2nd harmonic of the 'half IF frequency', a frequency differing from f_{LO} by one half the IF frequency, will have a difference equal to the IF. As a result these frequencies will mix to f_{IF} distorting the desired information at the demodulation input. The ½ IF frequency can be calculated as follows:

$$f_{\frac{1}{2}IF} = f_{LO} \pm \frac{1}{2} f_{IF} \tag{4.5}$$

Figure 4.9 shows the 1/2IF problem graphically.

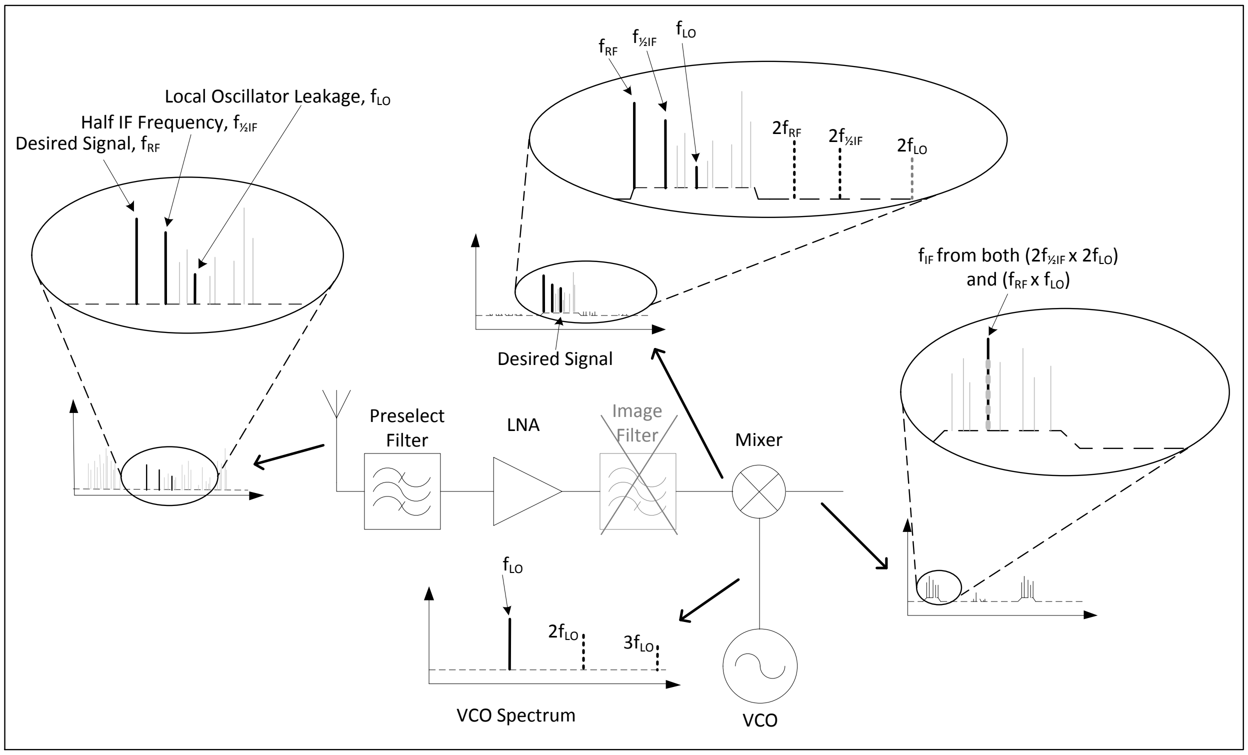


Figure 4.9 – 1/2IF & 1/3IF Spur Frequencies

It’s noteworthy to consider the same concept works for the third harmonics of the fLO frequency and the ‘one third IF frequency’, a signal located 1/3 of the IF frequency away from the fLO. The third harmonics of these two signals will also differ by the IF as well. Beyond this however, higher order spurs are typically not an issue as they are reduced enough in amplitude to be negligible if they are mixed to the IF frequency. Figure 4.9 shows the ½ IF spur problem graphically.

4.3 Spurious Response Test System

4.3.1 Manual Measurement Process

To investigate and characterize the spurious response of a “real world” radio receiver, it is necessary to observe how a receiver tuned to a desired frequency responds to a wide range of frequencies. To test receiver’s behavior, a signal was transmitted to the receiver using a signal generator, and the output of the receiver was observed to see if the signal was detected. If a signal was detected, the amplitude of the transmitted signal was decreased. This was repeated until the receiver no longer received the transmitted signal. Then the frequency and the minimum amplitude that the transmitted frequency was received at were recorded.

When performed manually, this process proved very time consuming as the following calculation is shows

$$T_{Measure} = T_{\Delta Frequency} + T_{Evaluate} = .5s + 1s = 1.5seconds$$

where $T_{\Delta Frequency}$= the time needed to change the frequency of the signal generator and $T_{Evaluate}$= the time needed to evaluate whether a signal is received. Taking the frequency range divided by a step size of 10 kHz, chosen based on the bandwidth of the last IF filter, the number of measurements becomes

$$\frac{f_{final} - f_{initial}}{\Delta f} = \frac{1GHz}{10kHz} = 100000$$

and the time to evaluate multiplied by the number of measurements yields

$$100000 * 1.5s \approx 41hours$$

This omits the time it takes to decrement the amplitude when a quieting event occurs and is quite optimistic about the time to manually tune and measure each frequency. Additionally, due to the sheer number of measurements needed to scan an adequate frequency range human error is increasingly likely. As a result it was determined an automated test setup was needed to obtain this amount of data.

4.3.2 Automated Measurement Development

4.3.2.1 – Quieting Detection Method

The first step in automating this test was determining a reliable method of event detection. When this test was done by hand the audio output of the receiver indicated when a signal was reaching the IF by “quieting” which is a natural result when a CW signal is received by a receiver in an FM mode. If no signal is reaching the IF the audible output of the receiver is just static. If an FM signal reaches the IF, it will be demodulated and output through the audio circuitry. In the case an unmodulated signal is received a fixed frequency is demodulated and the audio output of a receiver will be quiet.

The static output of the receiver is just white noise at audible frequencies, so an FFT of the digitized audio output from the receiver can be used to detect this ‘quieting’ event. First we establish an average noise magnitude when no signal is received. When signal is receiver it can be detected by noticing a decrease in the average noise magnitude as shown in Figure 4.10. Using a MyDAQ the audio output from the receiver was digitized and stored in the computer where LabVIEW code performed an FFT to analyze whether a signal was being received. The threshold to determine whether an event had occurred was set as a relative change in dB, typically 2dB. This made the system sensitive enough that it detected events with slightly greater sensitivity than by listening to the static by ear. However, comparisons in the lab showed the system agreed with manual event detection ± 2dB.

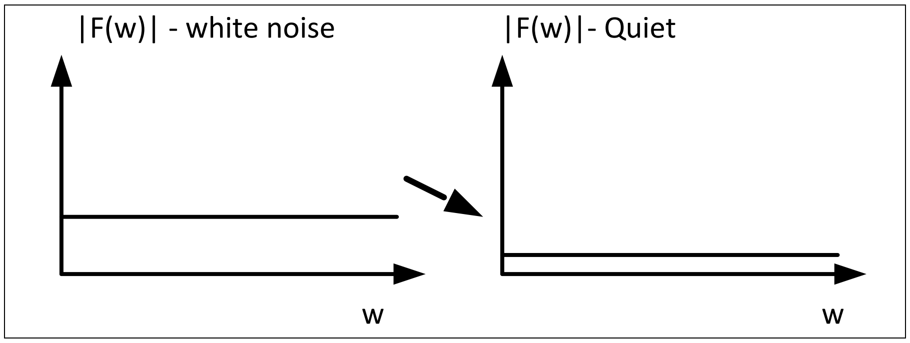


Figure 4.10 Fourier Spectrum Response to Quieting

Next LabVIEW code was written to manage the testing process by controlling the signal generator and detecting when the audio quieted indicating a spurious event. When a quieting even is detected, the code recorded the frequency the signal generator was transmitting when the event occurred and what amplitude the signal was reduced to that caused the event to stop occurring. The LabVIEW code controlled an Agilent signal generator via a GPIO interface using prewritten driver blocks provided with LabVIEW. This test system was used to test four radios:

1. VR-120
2. VX-3
3. SDR-14
4. K-State Microtransceiver

The test setup for these receivers is shown in the following section.

4.3.2.2 Physical Test Setup

Figure 4.11 shows the Spurious Rejection Test System running a test on the VX-3. The lower corner shows the VR-120, which was tested using an identical setup. As shown the Agilent signal generator was connected to the receiver to transmit signals and the audio output of the receiver was connected to the input of the MyDAQ, where the signal was put through an ADC and processed in LabVIEW.

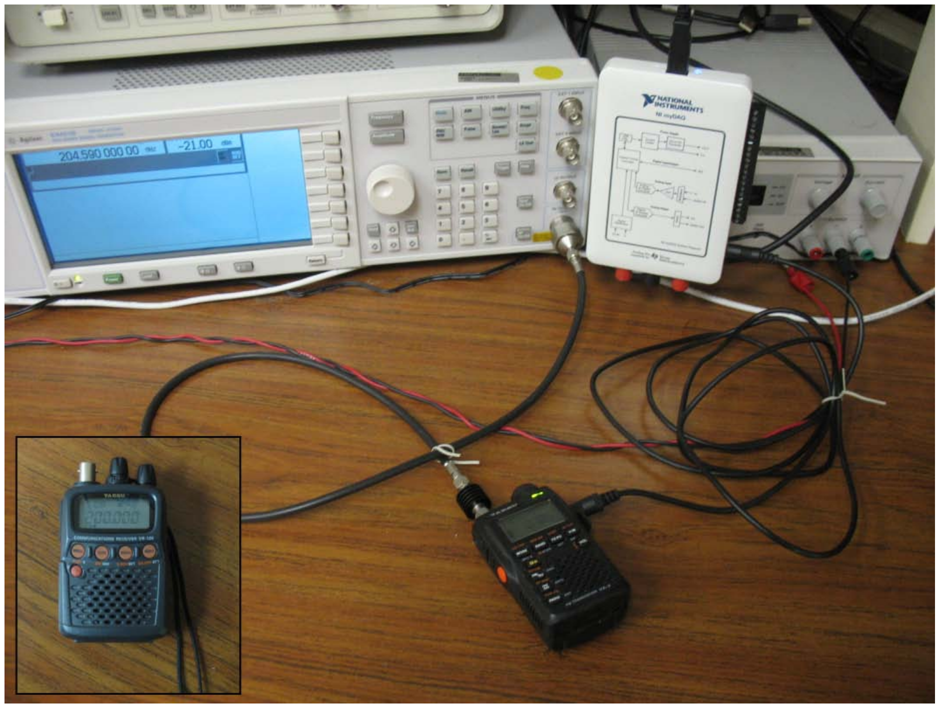


Figure 4.11 – Test Setup for VR-120 & VX-3

Figure 4.12 shows the test setup running a test on the software defined radio. A small screen capture of the SDR software running is shown to the lower right. The audio output from the SDR originated from the software and was output through the computer's audio and fed into the MyDAQ. Again, the Agilent signal generator transmitted the signal to the receiver.

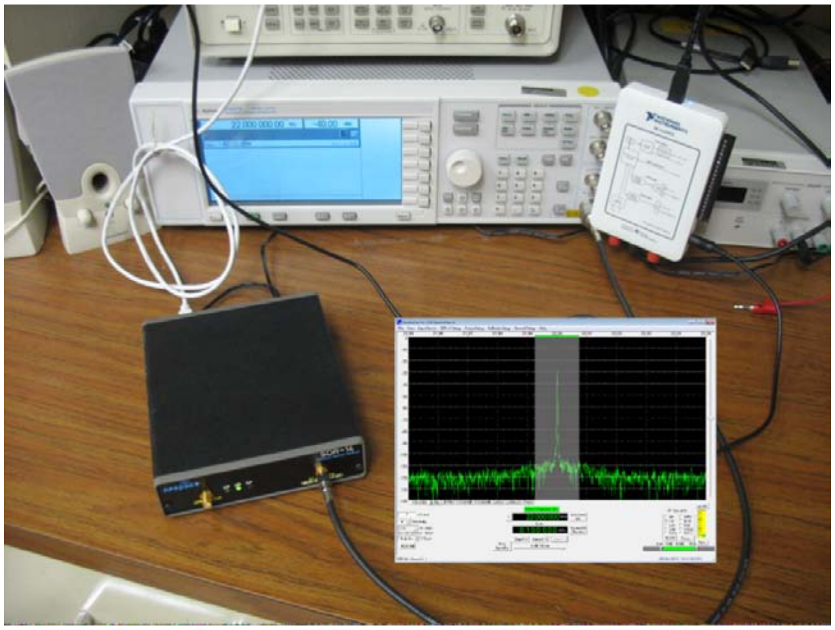


Figure 4.12 – Test Setup for SDR

Figure 4.13 shows the test setup for the K-State Microtransceiver. In this case the signal from the Agilent signal generator was fed into the K-State Microtransceiver. However, the K-State Microtransceiver doesn't have an audio output, so its IF output was sent into the VX-3 receiver tuned to receive the IF frequency. Then the audio output of the VX-3 was fed into the MyDAQ.

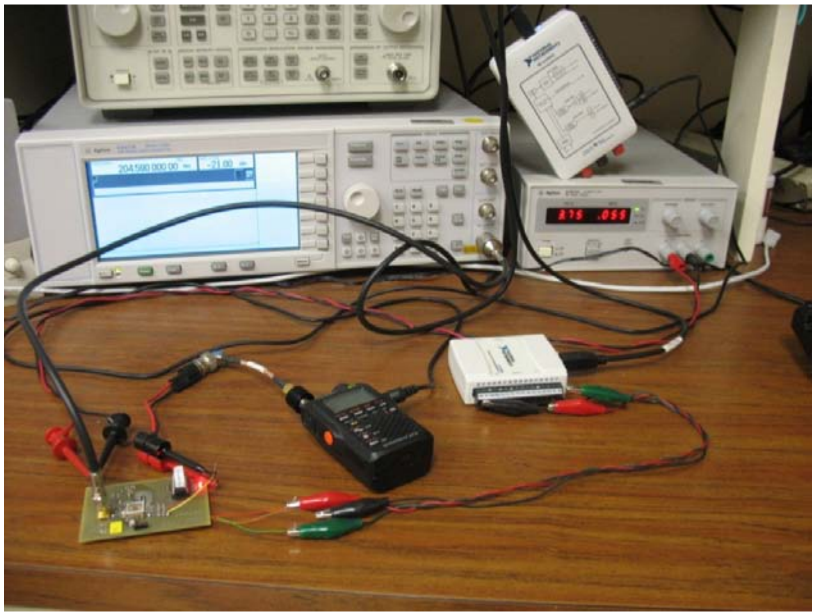


Figure 4.13 – Test Setup for K-State Microtransceiver

4.3.2.2 LabVIEW Code Algorithm

The LabVIEW code developed to automate this Spurious Response Test System included several screens of graphical code. To summarize its functionality Figure 4.14 shows the programmatic flow for the LabVIEW code.

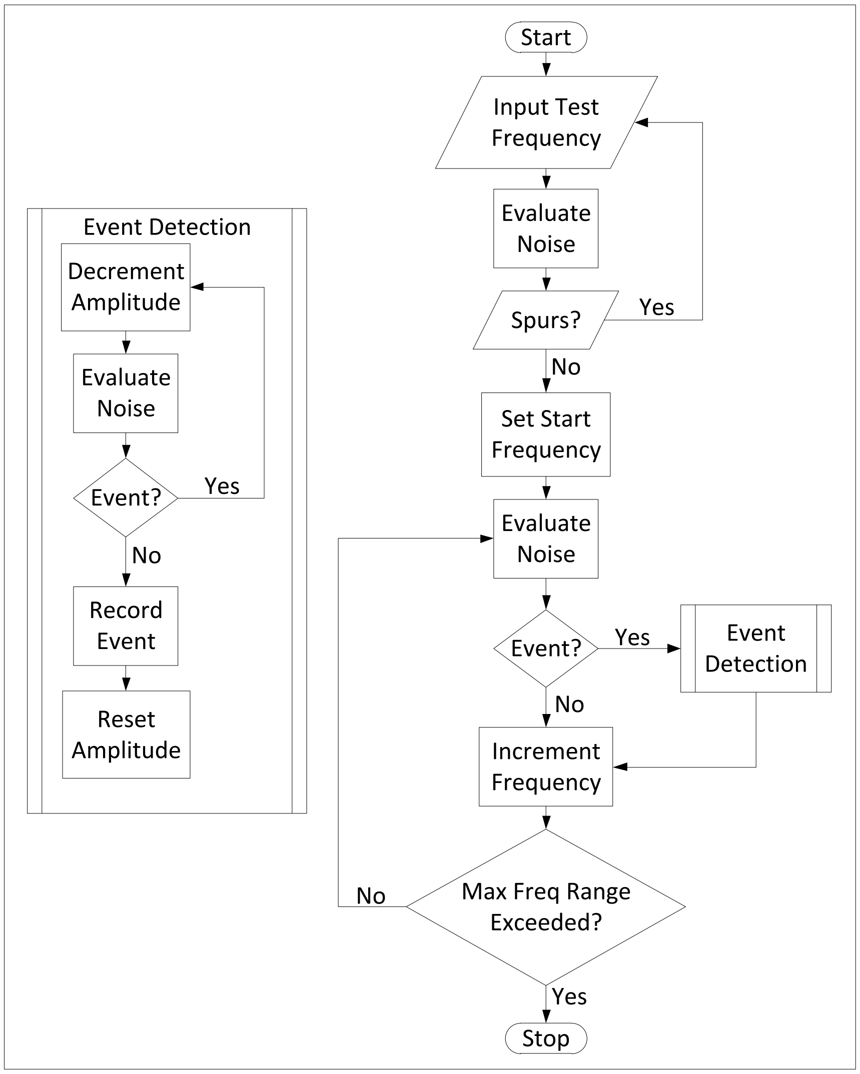


Figure 4.14 – Spurious Rejection Response Test System LabVIEW Code Flowchart

The LabVIEW GUI takes user settings for the start frequency, frequency range, frequency increment step size, the power of the base test signal and some operational settings. Once started the program asks the user to input a frequency to establish a baseline for a noisy output from the receiver. Once the user selects the frequency, the program commands that frequency and the user has the opportunity to decide if that frequency produces a noisy output typical of no signal being received. This test is important to determine whether or not a spur is being received at the frequency used to establish the noise baseline. Once the user confirms that this frequency will establish a correct baseline, the code takes over and controls the test iterations.

The program sets the frequency at the starting frequency and increments through the range in a specified step size equal to or less than the FM receiver's signal bandwidth. If the output at the receiver quiets, indicating a spurious response was generated for this transmitted frequency, the LabVIEW code decrements the transmitted signals power and measures the output again. The code repeats this process until the spur disappears and then records the frequency the spur occurred at and the amplitude the input signal was reduced to when the spur disappeared. Then the program resets the amplitude, increments the frequency and iterates the process of testing for a spur at each frequency. The LabVIEW code is documented in appendix C along with a screen capture of the VI running and instructions for running a test.

4.4 Receiver Block Diagrams

This section introduces each of the four receivers mentioned in section 4.3.2. These receivers spurious rejection responses are explored and compared using the previously described method of measurement in the next section. The architectures used these receivers are two multiple conversion superheterodyne architectures, a single conversion superheterodyne architecture, and a software defined radio. Because the architectures used in these receivers is an important basis for a cogent discussion of their spurious rejection responses, this section begins with a brief overview of each architecture.

4.4.1 VR-120

The VR-120 is a commercially available general coverage, .1 MHz to 1300 MHz, AM and FM receiver that uses a triple conversion superheterodyne architecture. A simplified diagram of the VR-120's architecture, based on a more detailed block diagram in its service manual, is shown in Figure 4.15. This diagram focuses on the FM modulated receive path since all tests used this option on the receiver.

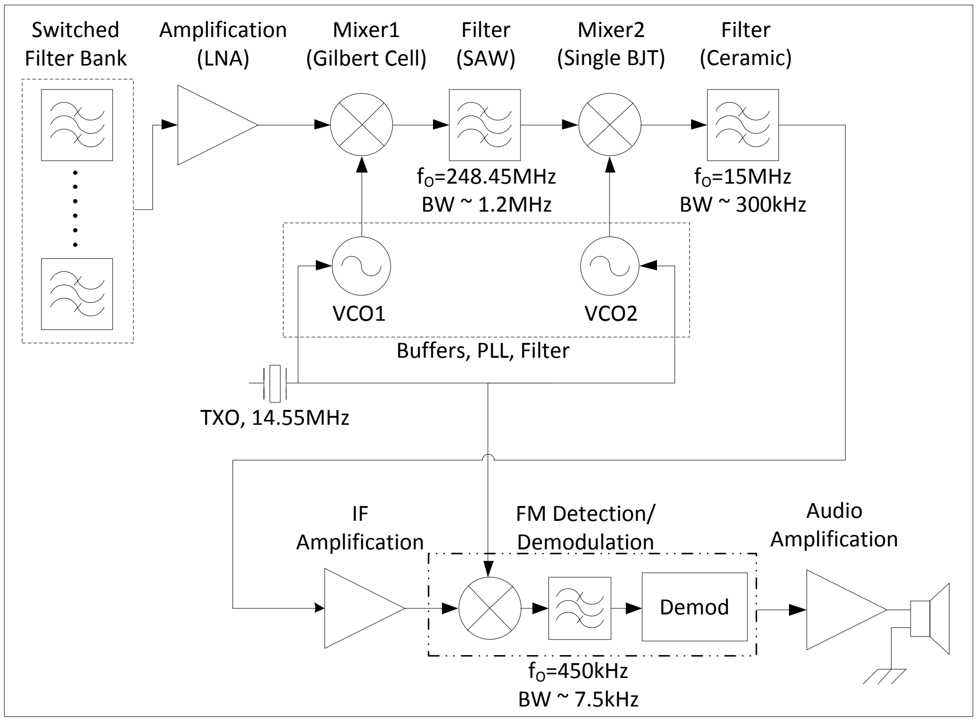


Figure 4.15 – VR-120 Block Diagram - after [18]

As shown, the first stage of the receiver is a bank of preselect filters for each of the bands the receiver can be tuned to detect. The received signal is then fed into a low noise amplification stage and into the first mixer. The signal is mixed to the first IF at 248.45MHz and filtered. Next the signal is mixed to the second IF at 15MHz and filtered again, followed by another amplification stage. Last the signal is mixed down to a final IF of 450kHz, filtered and demodulated.

4.4.2 VX-3

The VX-3 is an amateur radio transceiver which includes a general coverage .5 MHz to 999 MHz receiver that uses a double conversion superheterodyne architecture receiver. The VX-3 is capable of receiving a large number of different bands, similar to the VR-120. The full VX-3 block diagram from the service manual includes a separate receive path for each received band which includes a preselect filter, amplification, the first mixer and various filters. To simplify the diagram only the two receive chains used in collecting the data in this thesis are included along with relevant RF circuitry and the demodulation stage.

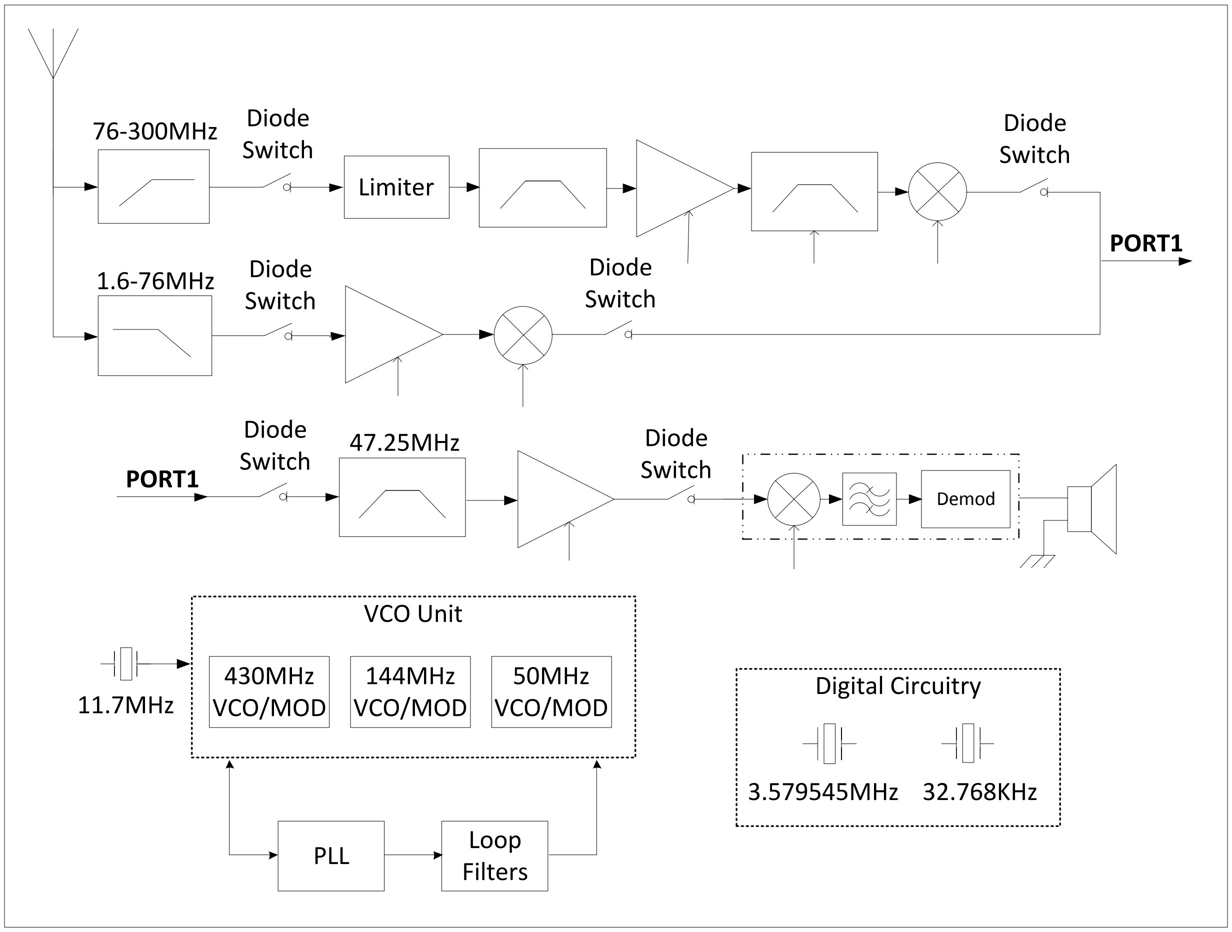


Figure 4.16 – VX-3 Block Diagram - after [19]

As shown in Figure 4.16 the receive chains are each proceeded by a preselect filter and followed by a diode switch used to control the bands processed. The receive chain for the 76-300MHz band shown proceeds into a limiter to limit the maximum power of incoming signals, a filter, an adjustable gain amplifier and a tunable filter. Last, another diode switch is used to direct signal flow before this receive chain enters the path used for all narrowband frequency modulated signals, the start of which is indicated in the figure above by ‘PORT1’. The 1.6-76MHz receive chain follows its first diode switch with an adjustable gain amplifier and a mixer before ending with another diode switch as it terminates in the final IF receive chain. The final IF receive chain of this circuit takes the received signal, now mixed to 47.25MHz, through another diode switch and into the final IF filter. Last, the IF is fed through another adjustable gain amplifier, a limiter and a final diode switch. The demodulator than follows further downconverts the signal to 450 kHz and delivers the information to the audio output.

4.4.3 Software Defined Radio

The next receover is the software defined radio. This SDR was developed for radio enthusiasts and can receive 0 Hz to 30 MHz.

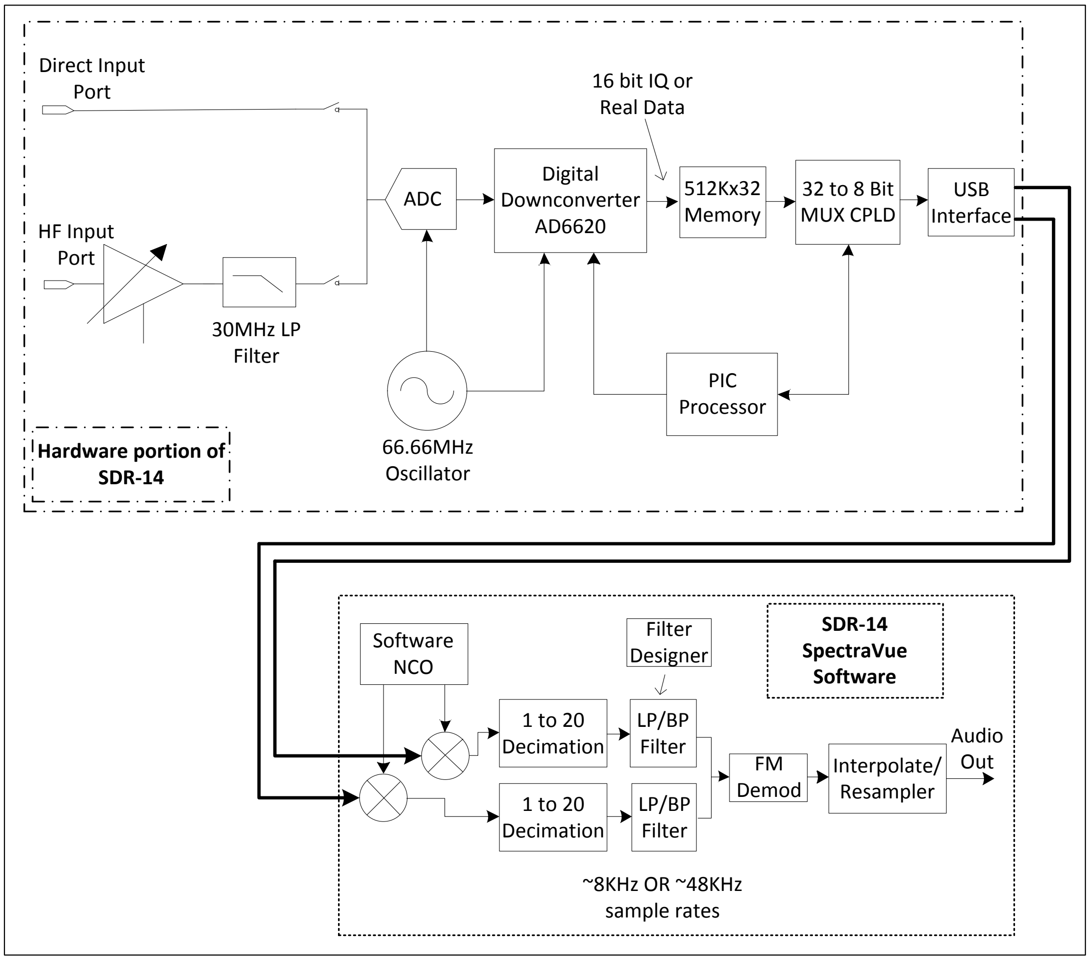


Figure 4.17 – SDR Block Diagram - after [20]

As shown there are two inputs on the SDR, both of which feed into the ADC. The path tested in this work was the ‘HF Input Port’. The ADC is driven with a 66.66MHz oscillator. After the ADC the signal is passed into a digital downconverter. The data is processed and buffered, and then sent via a USB interface to the software which completes the signal processing using decimators, filtering and demodulation. The software uses a numerically controlled oscillator allowing the user to choose how much of the spectrum to view and which frequency to receive.

4.4.4 K-State Microtransceiver

The K-State microtransceiver is a single conversion superheterodyne architecture receiver designed on an integrated circuit and developed and tested at Kansas State University. This transceiver works in the UHF band from approximately 350 MHz to 500 MHz and uses a single IF filter at 10.7 MHz, coupled with an image-reject mixer to mitigate the need for narrow image filtering after the LNA. The block diagram below shows the receive chain of the transceiver.

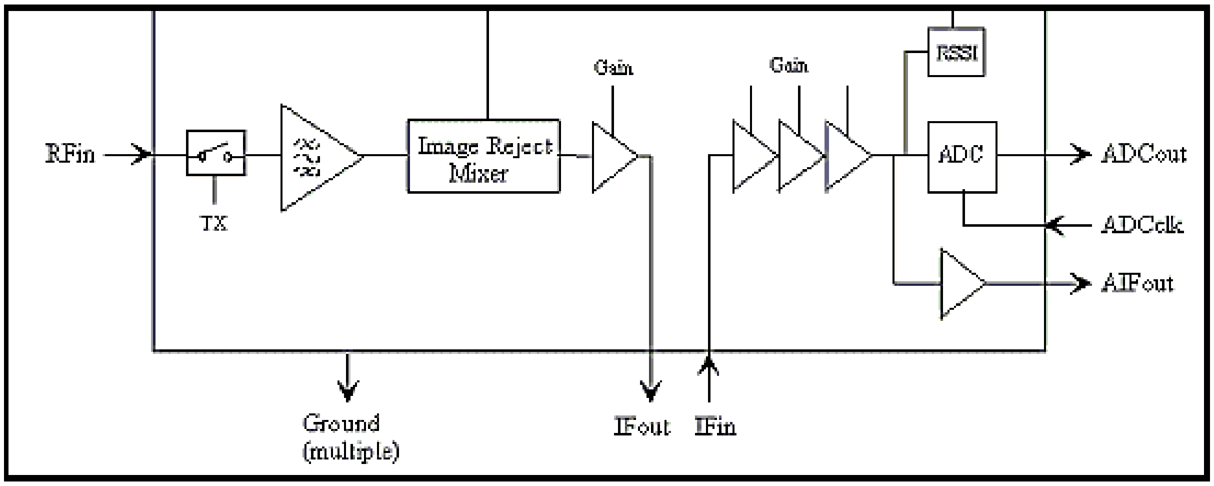


Figure 4.18 – K-State Microtranciever - after [21]

The received signal enters first through the transmit-receive (TR) switch and then goes into the tuned RF LNA. Next the signal is downconverted with an image reject mixer, amplified through an adjustable gain amplifier and sent to an off chip IF filter. Last, additional amplification is available in another chain of adjustable gain amplifiers and then the signal is fed to the ADC and finally the analog IF output ports.

4.5 Receiver Test Results

In this section the test results for the four receivers described section 4.4 are given and analyzed to identify causes of spurious responses. The data is presented in a novel graphical format which allows the various spurious responses to be observed. The data is then summarized in a table followed by a brief discussion of the test results. The first tests run were on the VR-120, VX-3 and the SDR where each receiver was tuned to receiver 3.6MHz. These tests demonstrate the validity of this testing methodology by showing some of the known spurious rejection issues addressed in section 4.3 in the data from the tests. The next set of tests compares these receivers performance tuned to a different frequency using the same preselect filter to their original behavior when tuned to 3.6MHz. The third set of tests evaluates the performance of the VR-120 and the VX-3 tuned to two different frequencies which use different preselect filters and their spurious rejection performance is compared. The last set of tests in this section explores the K-State microtransceiver's spurious rejection. Two revisions of the microtransceiver are tested, one tuned to 392.6MHz using an integer N synthesizer and the other tuned to 435MHz using a fractional N synthesizer. The naming convention adopted in the next sections was as follows:

1. f_{LO} = the local oscillator frequency
2. f_{RX} = the signal being transmitted to the receiver
3. f_{RF} = the signal the receiver is tuned to receive

4.5.1 Checking for Mixer Spurs

Since such emphasis is placed on mixer spurs as problems in receiver's spurious response rejection ability, the data obtained from the following test results was analyzed exhaustively to determine if the spurious responses found were due to mixer spurs. To accomplish this, a simple Octave function was written to analyze the data. The function took in all the data obtained from a test, the first IF frequency, and a few additional parameters. Using this information the function checked all the n and m coefficients up to a pre-designated maximum order to see if mixer spurs could explain the spurious response. The code for this analysis is shown in appendix D.

4.5.2 Three Receivers Tuned to an HF Frequency

The VR-120, the VX-3 and the SDR were tuned to 3.6 MHz and their spurious response rejection was tested. The VR-120 was scanned from 100 kHz to 1 GHz, the VX-3 was scanned over 100 kHz to 100 MHz, and the SDR was scanned from 100 kHz to 30 MHz. The full scan on VR-120 provided interesting data, but the scan took more than 48 hours so the VX-3 was scanned over a reduced range which included the full passband of the preselect filter. The SDR is only able to receive 0 MHz - 30 MHz so this entire range was scanned. The data from these tests are shown in graphical from in Figures 4.19 -4.27. The data is organized in Tables 4.1 – 4.10. In each table the f_{RX} is specified, the LO and IF frequencies and their subharmonics are listed, and the image frequency and the 1/2IF and 1/3IF spurs are listed. Last, each table contains the potential n and m values calculated as described in section 4.2 that may have produced the spurs in the data shown.

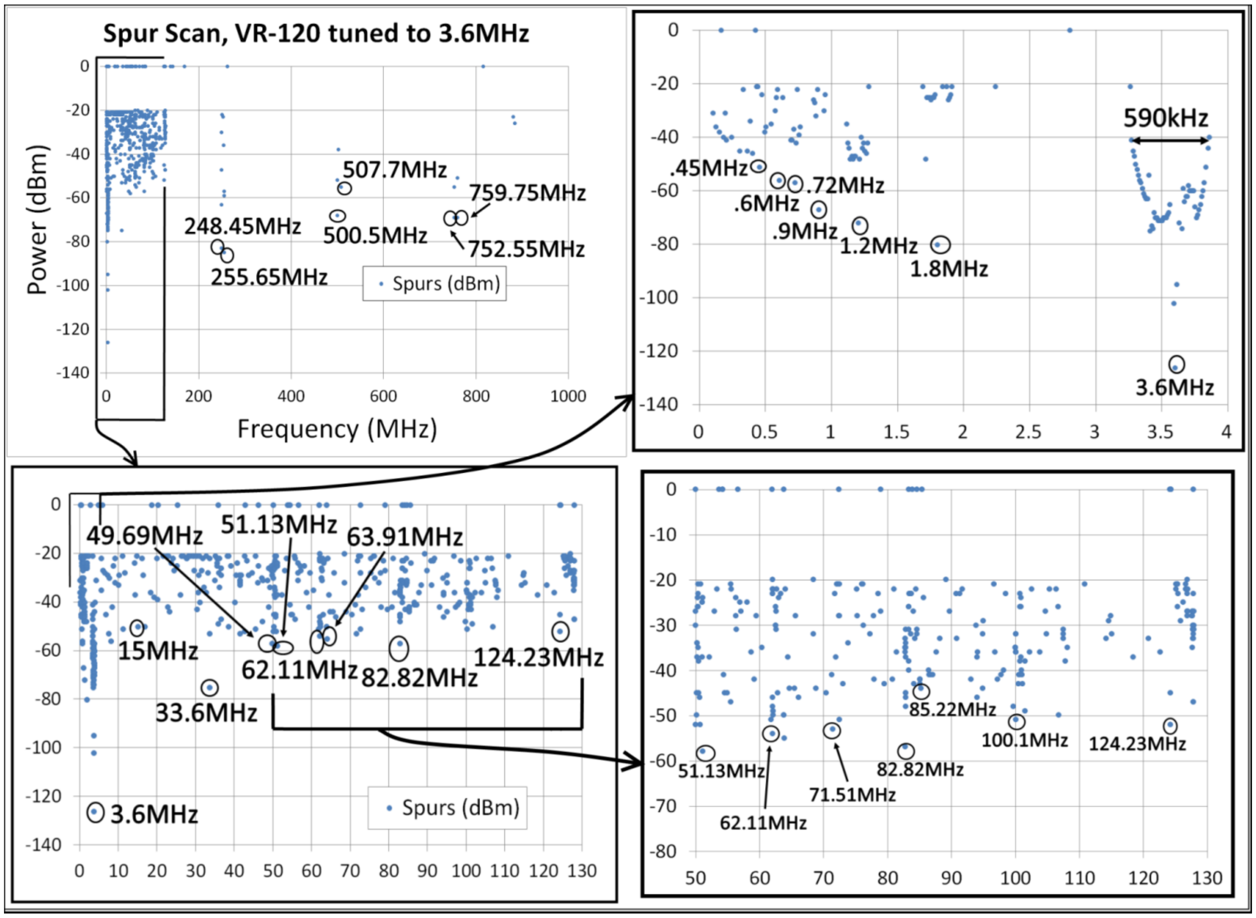


Figure 4.19 – VR-120 3.6 MHz Tune Test Results

VR-120	
f_{RX} (MHz)	3.600
Intermediate Frequencies, IF (MHz)	248.45, 15, .45
Local Oscillator Frequencies, f_{LO} (MHz)	252.05, 263.45, 15.45
Name/Source	Spur Observed (MHz)
Subharmonics of f_{RX}	1.8, 1.2, .9, .72, .6 …
IF1 feedthrough, and subharmonics	248.45, 124.23, 82.82, 62.11, 49.69, 41.41…
IF2 feedthrough, and subharmonics	15, 7.5, …
IF3 feedthrough	0.45
f_{IMAGE}=fLO+f_{IF1}	500.5
$f_{1/2IF}=f_{LO}\pm1/2f_{IF1}$	127.82, 376.275
$f_{1/3IF}=f_{LO}\pm1/3f_{IF1}$	169.23, 334.87

Mixing operations = $f_{IF}=mf_{LO}\pm nf_{RF}$	f_{RF} (MHz)	m	n
	36.52	2	-7
	42.61	2	-6
	51.13	2	-5
	63.91	2	-4
	71.5	-1	7
	72.53	3	-7
	85.22	2	-3
	100.1	-1	5
	101.54	3	-5
	255.65	2	-1
	507.7	3	-1
	752.55	-2	1
	759.75	4	-1

Table 4.1 – VR-120 3.6 MHz Tune Test Results

Figure 4.19 shows four views of the data from the spurious response rejection scan of VR-120 receiver tuned to 3.6 MHz. The first graph at the top right of Figure 4.19 shows the data from 100 kHz to 1 GHz. The spurs beyond about 130 MHz were mostly suppressed except for the image frequency at 500.5 MHz, feedthrough at 248.45 MHz, the first IF, and a few higher frequency spurs, the strongest of which were can be explained as mixer responses with m and n coefficients listed in Table 4.1. The next graph at the bottom left of Figure 4.19 shows a narrowed data range of 100 kHz to 130 MHz. This range of data shows a multitude of spurs. This data demonstrates the receiver's response to subharmonics of the IF by showing spurs at 124.45 MHz, 82.82 MHz, 62.11 MHz, and 49.69MHz. This graph also shows a spur at 15 MHz, implying that the receiver may experience some IF feedthrough from the second IF. Spurious responses at frequencies predicted by the m and n coefficients in Table 4.1 are also shown. It's noteworthy that one of the stronger responses at 33.6 MHz isn't predicted in the tabulated data. The graph to the lower right shows a further narrowed view of the data from 50 MHz to 130 MHz which shows additional spurious responses predicted in table 4.1. The last graph in the upper right shows a close in view of the spurious rejection response near the f_{RX} frequency, 3.6 MHz. This graph in particular highlights the effects of nonlinear amplification. Spurious responses down to the 1/6 subharmonic frequency of f_{RX} are shown. Another notable feature of this close in view is the spurious responses to signals out to roughly 295 kHz on either side of f_{RX}. This response could be due to the phase noise of the LO, or the combined IF bandpass filter responses, or some combination of both. Last, there are two spurious responses immediately adjacent on either side of f_{RX}, significantly less strong, which indicate they are at the edges of the final IF filter's roll off. These signals are ±10 kHz which is consistent with the known ≈7.5 kHz bandwidth of the final IF.

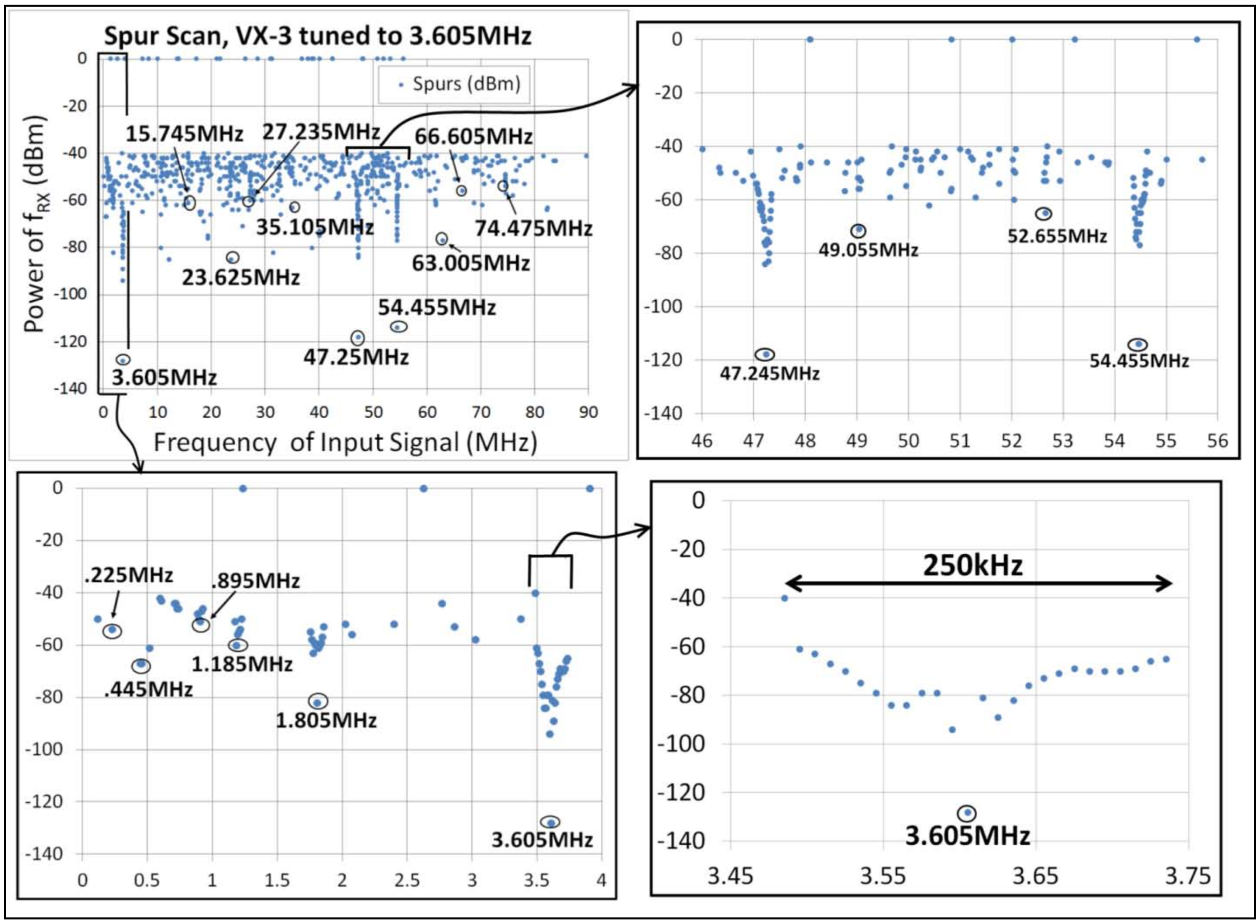


Figure 4.20 – VX-3 3.6 MHz Tune Test Results

VX-3	
f_{RX} (MHz)	3.605
Intermediate Frequencies, IF (MHz)	47.25, .45
Local Oscillator Frequencies, f_{LO} (MHz)	50.85, 46.8

Name/Source	Spur Observed (MHz)		
Subharmonics of f_{RX}	1.8, 1.20, .9, .72, .6…		
IF1 feedthrough, and subharmonics	47.25, 23.625, 15.75, 11.81, 9.45, 7.87, 6.75…		
IF2 feedthrough	0.45, .225…		
$f_{IMAGE}=f_{LO}+f_{IF1}$	98.100		
$f_{1/2IF}=f_{LO}\pm 1/2f_{IF1}$	27.225, 74.475		
$f_{1/3IF}=f_{LO}\pm 1/3f_{IF1}$	35.105, 66.605		
Mixing operations = $f_{IF}=mf_{LO}\pm nf_{RF}$	**f_{RF}**	**m**	**n**
	9.075	2	-6
	13.615	2	-4
	14.015	-1	7
	15.045	3	-7
	18.155	2	-3
	24.525	-1	4
	24.825	-2	6
	28.545	-3	7
	29.575	5	-7
	31.235	4	-5
	32.705	-1	3
	33.305	3	-6
	34.505	5	-6
	39.045	4	-4
	41.405	5	-5
	43.075	-5	7
	44.105	7	-7
	49.045	-1	2
	49.655	-2	3
	49.955	-3	4
	50.135	-4	5
	50.255	-5	6

	51.455	7	-6
	51.575	6	-5
	51.755	5	-4
	52.055	4	-3
	52.655	3	-2
	54.465	2	-1
	57.605	-7	7
	60.305	-5	5
	61.745	7	-5
	66.605	-3	3
	67.205	-7	6
	70.475	-6	5
	74.475	-2	2
	77.185	7	-4
	78.085	4	-2
	83.555	-4	3

Table 4.2 – VX-3 3.6 MHz Tune Test Results

Figure 4.20 shows four graphs of the VX-3's spurious response rejection when the receiver is tuned to 3.605 MHz. The graph at the top left shows the spurious response of the receiver over 100 kHz to 90 MHz. The spurs from 90 MHz to 100 MHz were completely suppressed. The data in this graphs shows strong spurs at the subharmonics of the first IF frequency, 23.625 MHz and 15.745 MHz. A very strong spur from IF feedthrough is shown at 47.25 MHz. Many additional spurs are also shown in this range, a number of which are predicted by mixing operations in Table 4.2. The 1/2IF and 1/3IF spurs are also shown in this data at 27.225 MHz, 74.475 MHz, 35.105 MHz and 66.605 MHz. The graph in the lower left shows a narrowed view of the data from 0 kHz to 4 MHz. The data in this graph shows spurious responses down to the 1/4 subharmonic frequency of f_{RX}, at 1.805 MHz, 1.185 MHz and .895MHz. Additionally, spurious responses at .445 MHz and .225 MHz indicate the receiver fails to block signals at the second IF and its 1/2 subharmonic frequency. The graph at the bottom right shows a close in view from 3.45 MHz to 3.75 MHz which shows a spurious response to signals around f_{RX} that looks very similar to a typical phase noise plot of a LO with a spread of spurious responses ±175 kHz around f_{RX}. The last graph in Figure 4.20 at the top right shows the first IF feedthrough and three responses which are close to values predicted in Table 4.2 at 49.245 MHz, 52.655 MHz and 54.455 MHz. Both the IF feedthrough and the spur at 54.455 MHz show a similar range of nearby spurious responses to the spread observed around f_{RX} indicating that the phase noise of the LO, or a filter bandpass response, or some other unidentified phenomenon is affecting this spur.

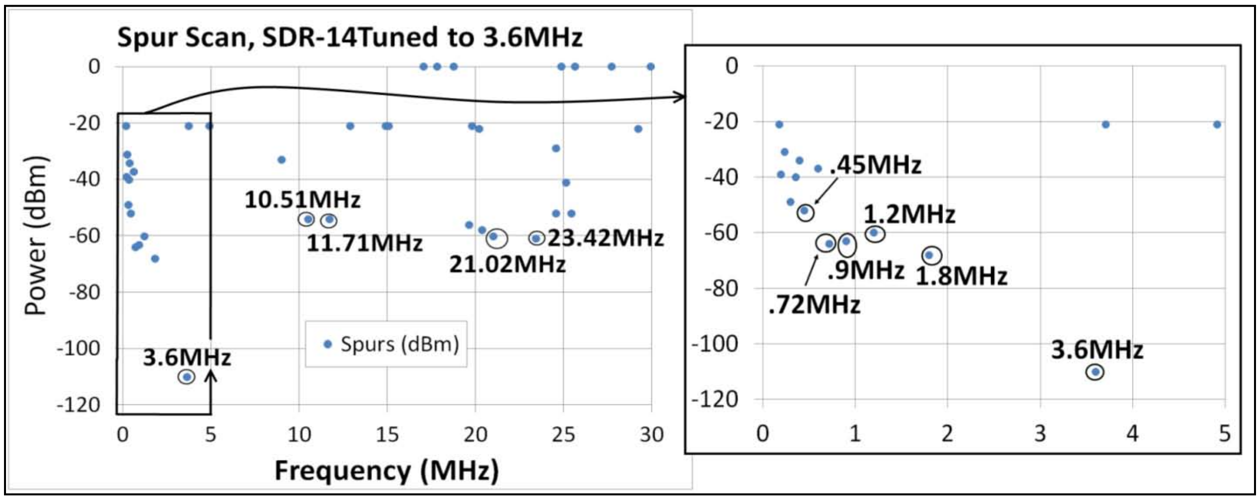


Figure 4.21 – SDR 3.6 MHz Tune Test Results

SDR	
f_{RX} (MHz)	3.60
Intermediate Frequencies, IF (MHz)	Unknown
Local Oscillator Frequencies, f_{LO} (MHz)	Unknown
Name/Source	**Spur Observed (MHz)**
Subharmonics of f_{RX}	1.8, 1.20, .9, .72, .6…
IF1 feedthrough, and subharmonics	Unknown

Table 4.3 – SDR 3.6 MHz Tune Test Results

Figure 4.21 shows two views of the data obtained from the spurious rejection scan from 100kHz to 30 MHz of the SDR tuned to 3.6 MHz. The first graph to the left shows full view of the data. The table for these spurs is brief and incomplete because this receiver architecture departs significantly from the superheterodyne receivers making the calculations of spurious response which were done for the VR-120 and the VX-3 impossible for the SDR. It is possible an image frequency response exists for this receiver, but the LO frequencies are variables defined in the software which are not immediately visible to the user, making it difficult to predict the image frequency by calculation. The same argument is applicable to calculations for other typical spurious responses. Also, this receiver employs many digital elements at software and hardware levels, including an ADC, which has the potential to produce any number of spurious responses. A detailed analysis of these specifics and this architecture is outside the scope of this thesis. However, it's noteworthy that there are a number of spurious responses shown for this receiver. The graph to the right in Figure 4.21 shows a narrowed view of the data from 0 kHz to 5 MHz and it's particularly interesting to note that strong spurious responses occur for frequencies down to the 1/5 subharmonic frequency of the tuned RF frequency. Clearly this receiver architecture is subject to nonlinear amplification behaviors.

The data collected from spurious response rejection scans of the VR-120 and the VX-3 were very similar to one another. Both receivers showed very strong IF feedthrough and mixing behaviors, however a surprisingly consistent and strong spurious response to subharmonics was observed emphasizing the problems due to nonlinear amplification and the need for improved filter solutions in the front end. The data from the SDR was different in many ways, but exhibited an important similarity to the VR-120 and VX-3 in that it shared sensitivity to the subharmonics of the f_{RX}. All three receivers performed poorly for portions of these scans. Even the SDR, which is relatively expensive and draws significant power, suffered from many spurious responses.

It's worth noting here as well, that in the data below there are a number of spurious responses included which are unexplained. However, due to sheer amount of data represented in even a single graph, it is beyond the scope of this thesis to analyze all of the data exhaustively. However, the data demonstrates well known behaviors in receivers supporting the validity of this testing methodology. These tests also show additional insight into receiver's spurious response rejection not emphasized adequately in current research. The graphical display of results lends an easy intuitive understanding of a receiver's ability to block unwanted signals and characterize a receiver's performance.

4.5.3 Three Receivers Tuned to Two Different Frequencies in HF Band

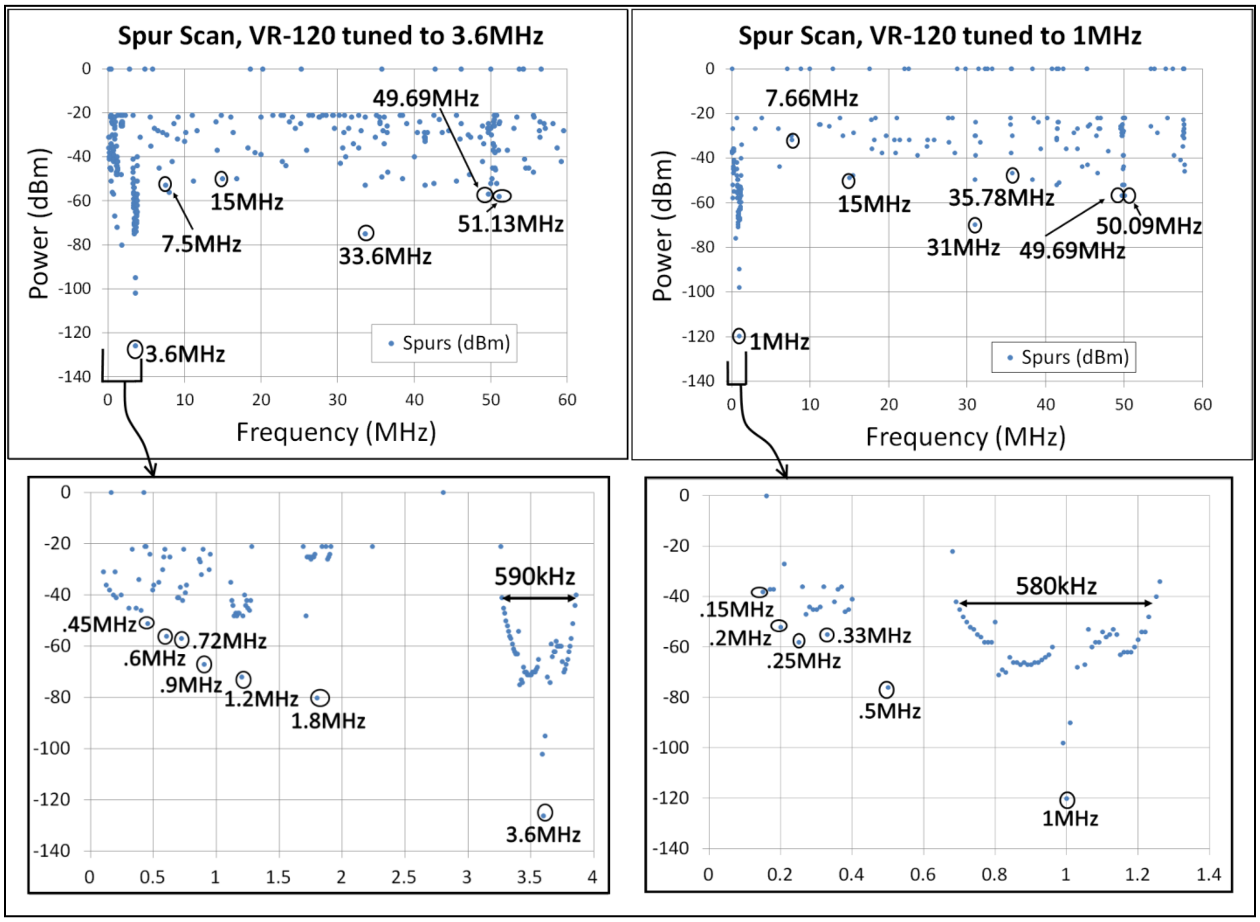


Figure 4.22 – VR-120 3.6 MHz to 1 MHz Tune Comparison Test Results

VR-120	
f_{RX} (MHz)	1.000
Intermediate Frequencies, IF (MHz)	248.45, 15, .45
Local Oscillator Frequencies, f_{LO} (MHz)	249.45, 263.45, 15.45

Name/Source	**Spur Observed (MHz)**		
Subharmonics of f_{RX}	.5, .33, .25, .2, 16…		
IF1 feedthrough, and subharmonics	248.45, 124.23, 82.82, 62.11, 49.69, 41.41…		
IF2 feedthrough	15, 7.5…		
IF3 feedthrough	0.45…		
$f_{IMAGE}=f_{LO}+f_{IF1}$	497.9		
$f_{1/2IF}=f_{LO}\pm 1/2f_{IF1}$	125.225, 373.675		
$f_{1/3IF}=f_{LO}\pm 1/3f_{IF1}$	166.63, 332.26		
Mixing operations = $f_{IF}=mf_{LO}\pm nf_{RF}$	**f_{RF} (MHz)**	**m**	**n**
	35.78	2	-7
	41.74	2	-6
	50.09	2	-5
	250.45	2	-1

Table 4.5 – VR-120 3.6 MHz to 1 MHz Tune Comparison Test Results

Figure 4.22 shows a comparison of the VR-120's spurious rejection response tuned to receive 1 MHz to the VR-120's spurious rejection response tuned receive to 3.6 MHz presented in section 4.5.2. The two graphs to the right provide a graphical display of the data from the new test from 100 kHz to 60 MHz. The image frequency, 1/2IF and 1/3IF spurs, shown in Table 4.5, are outside the range of this scan. The graph at the top right shows several responses including a spur at the second IF, 15 MHz, and another spur very close to its 1/2 subharmonic frequency at 7.66 MHz. This behavior is nearly the same as the first scan with f_{RX} tuned to 3.6 MHz. The 1/5 subharmonic frequency of the first IF is shown clearly at 49.69 MHz showing IF feedthrough is still a major problem for frequencies at the IF and its subharmonics. The spurs at 35.78 MHz and 50.09 MHz are predicted as mixer spurs in Table 4.5 and differ from the first scan as expected, since f_{LO} changes with f_{RX}. The narrowed frequency range from 100 kHz to 1 MHz in the graph to the lower right of Figure 4.22 shows strong spurs down to the 1/6 subharmonic frequency, exactly like the first scan reemphasizing that subharmonics of frequencies that produce spurious responses are a serious issue. Similar to the first scan the spread of spurs right around f_{RX} is about ±290 kHz on either side and two there are two spurious responses are immediately adjacent to f_{RX}, but much reduced in strength. Whether the source of this spread of spurs is LO phase noise or IF bandwidths, the behavior appears invariant when the receiver is tuned to different frequencies.

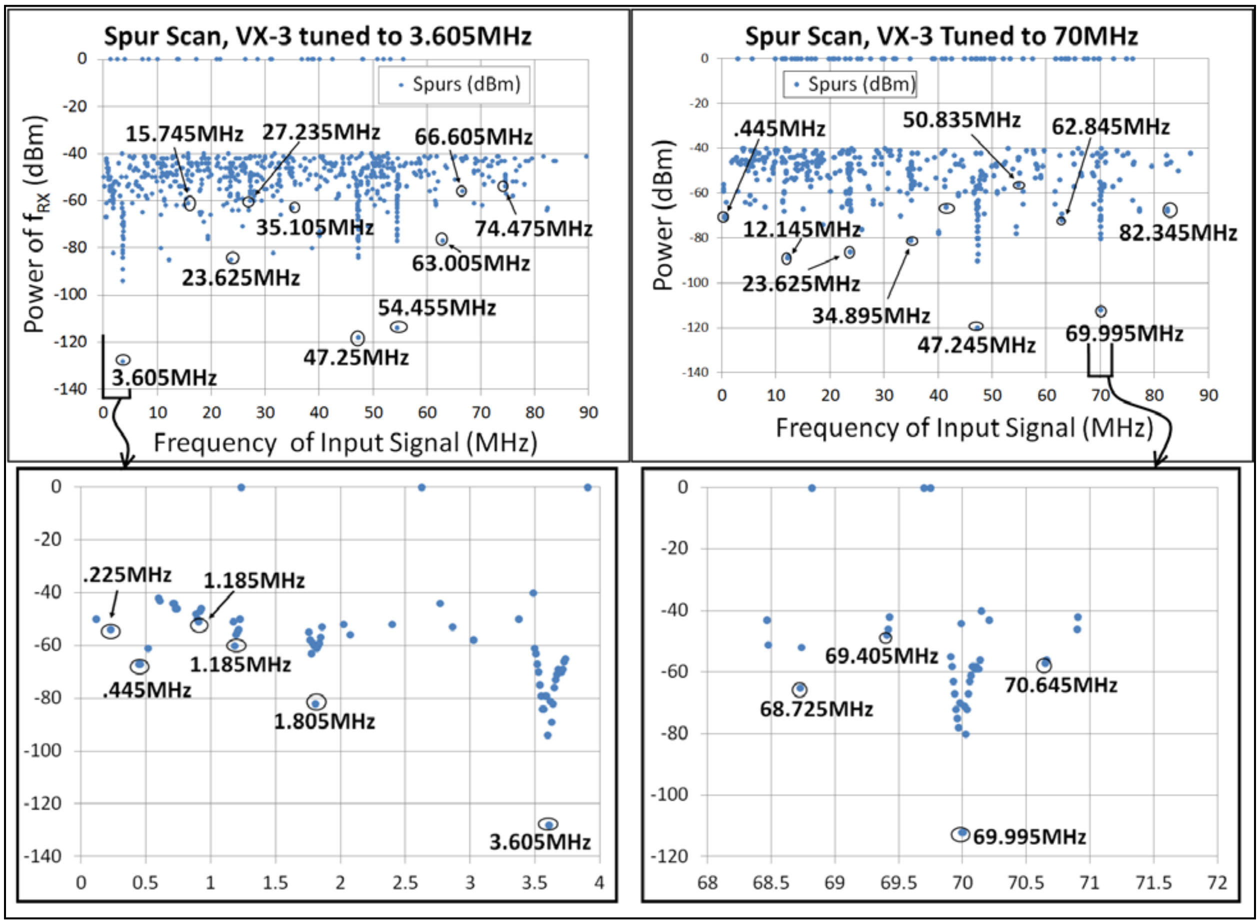


Figure 4.23 – VX-3 3.6 MHz to 70 MHz Tune Comparison Test Results

VX-3	
f_{RX} (MHz)	70.000
Intermediate Frequencies, IF (MHz)	47.25, .45
Local Oscillator Frequencies, f_{LO} (MHz)	22.75, 46.8

Name/Source	**Spur Observed (MHz)**		
Subharmonics of f_{RX}	35, 23.3, 17.5, …		
IF1 feedthrough, and subharmonics	47.25, 23.625, 15.75, 11.81, 9.45, 7.87, 6.75…		
IF2 feedthrough	0.45, .225…		
$f_{IMAGE}=f_{LO}+f_{IF1}$	117.250		
$f_{1/2IF}=f_{LO}\pm 1/2f_{IF1}$	93.625, 140.875		
$f_{1/3IF}=f_{LO}\pm 1/3f_{IF1}$	101.5, 133		
Mixing operations = $f_{IF}=mf_{LO}\pm nf_{RF}$	**f_{RF} (MHz)**	**m**	**n**
	11.665	1	-6
	23.335	1	-3
	35.005	1	-2
	41.125	-1	4
	54.835	-1	3
	62.415	2	-3
	76.125	3	-4
	82.255	-1	2

Table 4.6 – VX-3 3.6 MHz to 70 MHz Tune Comparison Test Results

Figure 4.23 shows a comparison of the VX-3's spurious rejection response tuned to receive 70 MHz to the VX-3's spurious rejection response tuned receive to 3.606 MHz presented in section 4.5.2. The two graphs to the right provide a graphical display of the data from the new test from 100 kHz to 90 MHz. The image frequency, 1/2IF and 1/3IF spurs, shown in Table 4.6, are outside the range of this scan. The graph at the top right shows a number of responses including a strong IF feedthrough and its second harmonic, in this case stronger than the actual f_{RX}, similar to the first VX-3 scan. Several additional mixer spurs are shown as well, predicted in Table 4.6, which differ as expected from the first VX-3 scan since f_{LO} is varied to receive a different f_{RX}. The narrowed frequency range from 68 MHz to 72 MHz shown in the lower right of Figure 4.6 shows a close in view of f_{RX}. This view shows some close-in spurious responses that are not predicted in tabulated data. This view also shows the same spread of spurs roughly ±175 kHz around f_{RX} as observed when the VX-3 was tuned to 3.605MHz, again indicating this behavior is invariant when the f_{RF} is changed.

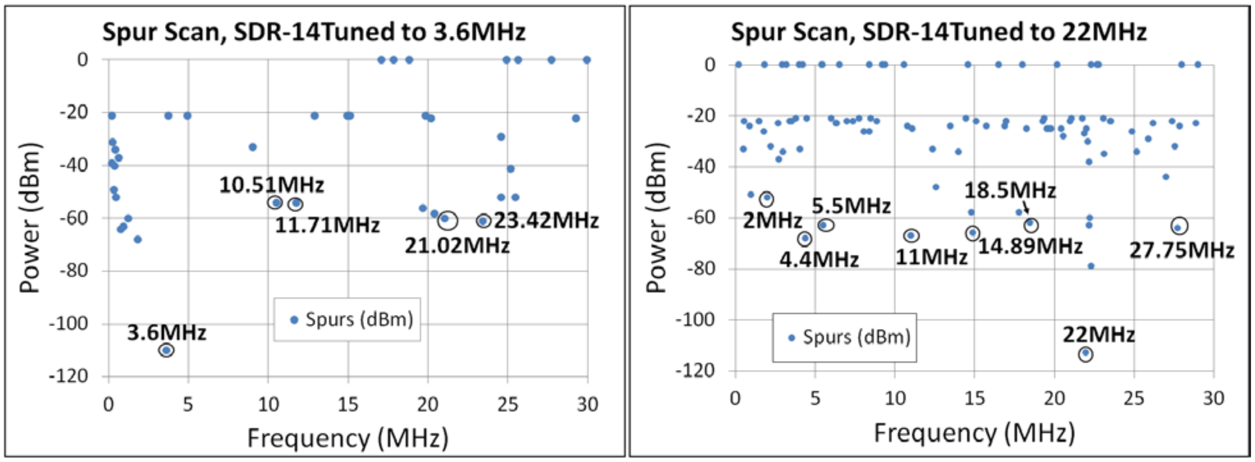


Figure 4.24 – SDR-3 3.6 MHz to 22 MHz Tune Comparison Test Results

SDR	
f_{RX} (MHz)	22
Intermediate Frequencies, IF (MHz)	Unknown
Local Oscillator Frequencies, f_{LO} (MHz)	Unknown
Name/Source	**Spur Observed (MHz)**
Subharmonics of f_{RX}	11, 5.5, 2.75, 1.375, …
IF1 feedthrough, and subharmonics	Unknown

Table 4.7 – SDR 3.6 MHz to 22 MHz Tune Comparison Test Results

Figure 4.24 shows a comparison of the SDR’s spurious responses when tuned to receive 22 MHz to the SDR’s spurious responses when tuned receive to 3.6 MHz. The graph on the right in Figure 4.24 shows the new data from 100 kHz to 30 MHz. As in the first scan there are a number of spurs in the data not predicted in the tabulated data due to the additional complexity of the software driven variable components and digital circuitry. The new data dramatically emphasizes the issue of subharmonics by showing strong second and third order responses at 11 MHz and 5.5 MHz.

This section shows that each of these receiver’s spurious response behaviors contain consistent trends even when f_{RF} is changed. A major problem that causes strong spurious responses in receivers is IF feedthrough and its subharmonics. Signals too close to the f_{RF} create spurs as do harmonics of f_{LO} and f_{RF}, but there are a number of spurs which aren’t explained even by these predictions.

4.5.4 Two Receivers Compared at VHF Bands

This section shows the VR-120's and the VX-3's spurious rejection responses when tuned to different bands that make use of different preselect filters than the last two sections. These responses are compared and their data analyzed. Differences and similarities between these responses and those in the last two sections are addressed

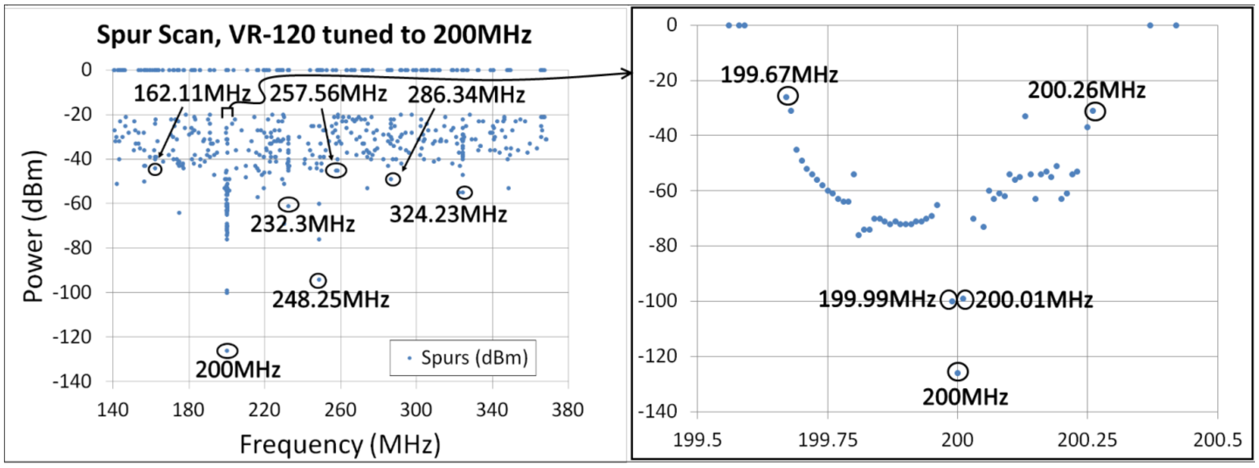


Figure 4.25 – VR-120 120 MHz Tune Comparison Test Results

VR-120	
f_{RX} (MHz)	200.000
Intermediate Frequencies, IF (MHz)	248.45, 15, .45
Local Oscillator Frequencies, f_{LO} (MHz)	448.45, 263.45, 15.45
Name/Source	**Spur Observed (MHz)**
Subharmonics of f_{RX}	100, 66.67, …
IF1 feedthrough, and subharmonics	248.45, 124.23, …
IF2 feedthrough	15, 7.5, …
IF3 feedthrough	0.45, …
$f_{IMAGE}=f_{LO}+f_{IF1}$	696.9
$f_{1/2IF}=f_{LO}\pm1/2f_{IF1}$	324.225, 572.675
$f_{1/3IF}=f_{LO}\pm1/3f_{IF1}$	365.63, 531.26

Mixing operations = f_{IF}=mf_{LO}±nf_{RF}	**f_{RF}**	**m**	**n**
	156.7	3	-7
	162.11	2	-4
	216.15	2	-3
	219.38	3	-5
	229.07	-2	5
	232.3	-1	3
	257.56	4	-6
	284.83	5	-7
	286.34	-2	4
	291.75	-4	7
	309.07	4	-5
	318.76	-3	5
	324.22	2	-2
	348.45	-1	2

Table 4.8 – VR-120 120 MHz Tune Comparison Test Results

Figure 4.25 shows a comparison of the VR-120's spurious rejection response from 140 MHz to 380 MHz, tuned to receive 200 MHz. The overall behavior of the spurious rejection response is quite similar to this receiver's response when tuned to receive 3.6 MHz or 1 MHz. Subharmonics of the f_{RF} and the IF frequencies are outside the range of this scan, however the graph on the left of Figure 4.25 shows strong feedthrough for the first IF at 248.25 MHz. The 1/2IF spur at 324.22 MHz is also plainly observed along with a number of mixer spurs predicted in Table 4.8. A narrowed view of the data from 199.5 MHz to 200.5 MHz is shown on the right of the figure with a range ±295 kHz on either side of f_{RF} of nearby spurious responses. Also there are two slightly strong spurs immediately adjacent to fRF at ±10 kHz indicating the roll off of the final IF filter.

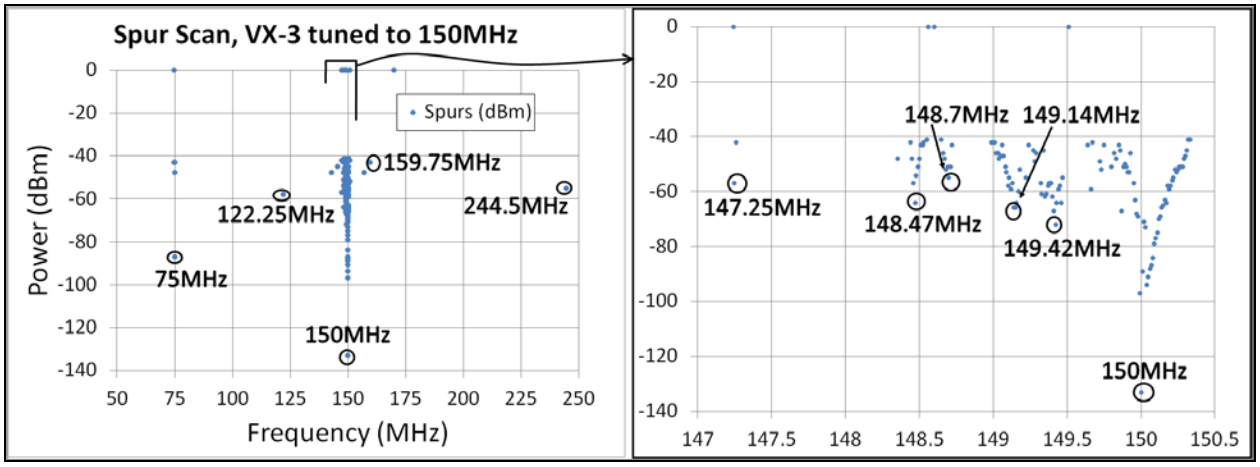


Figure 4.26 – VX-3 150 MHz Tune Comparison Test Results

VX-3			
f_{RX} (MHz)	150.000		
Intermediate Frequencies, IF (MHz)	47.25, .45		
Local Oscillator Frequencies, f_{LO} (MHz)	197.25, 46.8		
Name/Source	**Spur Observed (MHz)**		
Subharmonics of f_{RX}	75, 50, …		
IF1 feedthrough, and subharmonics	47.25, 23.625, …		
IF2 feedthrough	0.45, .225		
$f_{IMAGE}=f_{LO}+f_{IF1}$	244.500		
$f_{1/2IF}=f_{LO}\pm 1/2f_{IF1}$	173.625, 220.875		
$f_{1/3IF}=f_{LO}\pm 1/3f_{IF1}$	181.5, 213		
Mixing operations = $f_{IF}=mf_{LO}\pm nf_{RF}$	**f_{RF}**	**m**	**n**
	75	1	-2
	122.25	-1	2
	147.26	-2	3
	148.35	4	-5
	159.75	-3	4

Table 4.9 – VX-3 150 MHz Tune Comparison Test Results

Figure 4.26 shows a comparison of the VX-3's spurious rejection response from 50 MHz to 250 MHz, tuned to receive 150 MHz. The overall behavior of this spurious rejection response is significantly better than when the VX-3 was tuned to receive 3.6 MHz or 70 MHz. The response is amazingly spur free. The graph on the left shows a strong 1/2 subharmonic frequency response at 75 MHz. The image frequency is shown at 244.5 MHz and a few mixer spurs predicted in Table 4.9 are shown, including 122.25 MHz and 159.75 MHz. The graph on the right shows the same spur spreading around f_{RF} of about ±175 kHz as seen in the test results from the previous two sections. Additionally, a repeated pattern of spurs is shown to the left of the tuned frequency perhaps due to some digital circuitry. Last, a mixer spur is shown at 147.25 MHz.

In this section as well as the previous two, subharmonics are the most common problem. This source of spurs is demonstrated in every test indicating that this issue a universal problem even across varying radio architectures. While other responses were observed, such as IF feedthrough, 1/2IF and 1/3IF spurs, and mixer spurs, there is a large number of unexplained behaviors in these spurious responses. The graphical representation provides insight into the receiver performance and guides the designer when looking at the receiver about which frequencies to consider when assessing spur blocking capabilities.

4.5.5 The K-State Microtransceiver Spurious Rejection Response at UHF

This section looks at the K-State microtransceiver spurious rejection response for two cases: first, the receiver is tuned to receive 392.6 MHz using only the integer N synthesizer and second, the receiver is tuned to receive 435 MHz and is using the fractional N synthesizer. These plots show the spurious responses in both cases demonstrate some expected behaviors and allow an easy comparison to be made about how the fractional N synthesizer impacts the spurious response.

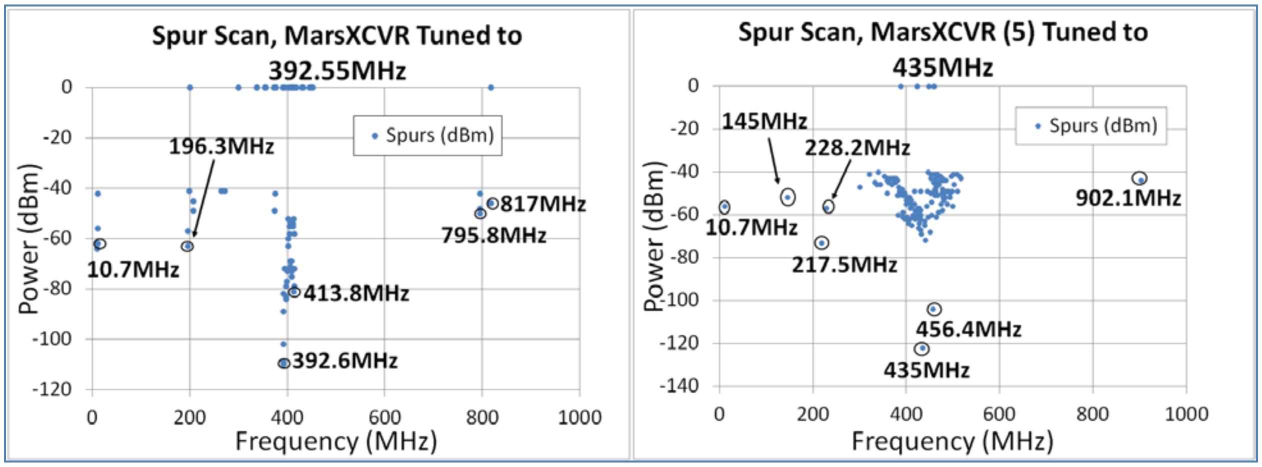


Figure 4.27 – K-State Microtransceiver Without & With the Fractional N-Synthesizer Test Results

K-State Microtransceiver		
f_{RX} (MHz)	435.0	392.6
Intermediate Frequency, IF (MHz)	10.7	10.7
Local Oscillator Frequency, f_{LO} (MHz)	445.7	403.3
Name/Source	**Spur Observed (MHz)**	**Spur Observed (MHz)**
Subharmonics of fRX	217.5, 145, 108.75, 87, 72.5, …	196.3, 130.8, 98.15, 78.52, …
IF1 feedthrough, and subharmonics	10.7, 5.35, …	10.7, 5.35, …
$f_{IMAGE}=f_{LO}+f_{IF1}$	456.4	414
$f_{1/2IF}=f_{LO}\pm1/2f_{IF1}$	440.35, 451.05	397.95, 408.65
$f_{1/3IF}=f_{LO}\pm1/3f_{IF1}$	442.13, 449.26	399.73, 406.86

Table 4.10 – K-State Microtransceiver Without & With the Fractional N-Synthesizer Test Results

Figure 4.27 shows the graphical display of two spurious rejection response tests results from the K-State microtransceiver. The first test shown in the graph on the left was run with 100 kHz steps and the second test shown in the graph to the right is a test run with 50 kHz steps. It was determined that the step size should be 10 kHz for future tests, however these tests still show important behaviors such as IF feedthrough, subharmonic responses, and strong image responses. More importantly it's also clear that, as expected, the use of the fractional N synthesizer increases the number of spurious responses significantly. However, the overall performance of this receiver exceeds the other receivers showing an overall robust spurious rejection response.

Chapter 5 - The Q-Enhanced Filter as a Solution to SDR Architectures

The results in Chapter 4 showed many responses were a result of subharmonic frequencies of the IF frequencies or f_{RF}, or the result of other nonlinearities due to amplification which then mixed with the LO. There were also a number of spurious responses which were unexplained. If a frequency synthesizer is used with significant spurs and/or phase noise the increase in the number of spurious responses rose significantly. These results provoked the question about what would happen to a spurious response if the IF were changed. The Q-enhanced filter provides that type of ability, so if varying the IF were to improve the spurious response significantly, the Q-enhanced filter could be a valuable IF filter in a receiver.

5.1 – Spur Reduction Achieved by Changing the IF

To test how spurious responses changed when the IF was shifted, the spurious response rejection test with the K-State Microtransceiver was rerun where the receiver was tuned to 435 MHz with two different IF filters. This was accomplished by changing the external IF filter and varying the LO accordingly. Figure 5.1 shows these results and Tables 5.1 and 5.2 show some expected responses to these tests respectively.

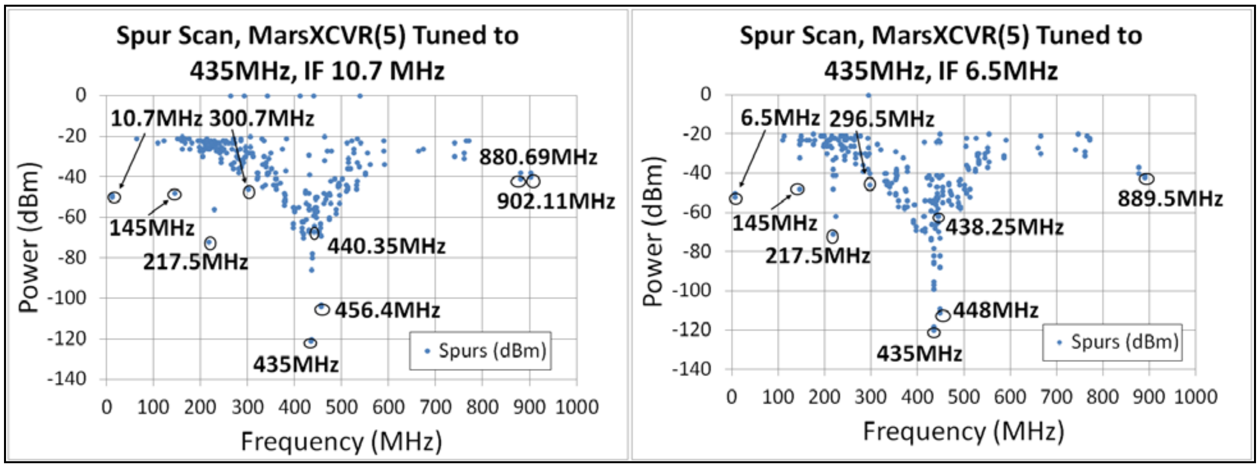


Figure 5.1 – K-State Microtransceiver 435MHz Tune Different IF Comparison Test Results

K-State Microtranciever			
f_{RX} (MHz)	435.000		
Intermediate Frequency, IF (MHz)	10.700		
Local Oscillator Frequency, f_{LO} (MHz)	445.700		
Name/Source	**Spur Observed (MHz)**		
Subharmonics of fRX	217.5, 145, 108.75, 87, 72.5, …		
IF1 feedthrough, and subharmonics	10.7, 5.35, …		
$f_{IMAGE}=f_{LO}+f_{IF1}$	456.4		
$f_{1/2IF}=f_{LO}\pm 1/2f_{IF1}$	440.35, 451.05		
$f_{1/3IF}=f_{LO}\pm 1/3f_{IF1}$	442.13, 449.26		
Mixing operations = $f_{IF}=mf_{LO}\pm nf_{RF}$	f_{RF}	**m**	**n**
	228.2	-1	2
	300.7	-2	3
	331.6	3	-4
	336.95	-3	4
	354.42	4	-5
	358.7	-4	5
	443.56	5	-5
	444.17	7	-7
	447.23	-7	7
	447.84	-5	5
	880.69	2	-1
	902.11	-2	1

Table 5.1 – K-State Microtransceiver Spurious Response, IF= 10.7 MHz

K-State Microtransceiver				
f_{RX} (MHz)	435.000			
Intermediate Frequency, IF (MHz)	6.500			
Local Oscillator Frequency, f_{LO} (MHz)	441.500			
Name/Source	**Spur Observed (MHz)**			
Subharmonics of f_{RX}	217.5, 145, 108.75, 87, 72.5, …			
IF1 feedthrough, and subharmonics	6.5, 3.25, …			
$f_{IMAGE}=f_{LO}+f_{IF1}$	448.0			
$f_{1/2IF}=f_{LO}\pm 1/2f_{IF1}$	438.25, 444.75			
$f_{1/3IF}=f_{LO}\pm 1/3f_{IF1}$	439.33, 443.67			
Mixing operations = $f_{IF}=mf_{LO}\pm nf_{RF}$	**f_{RF}**	**m**	**n**	
	224	-1	2	
	296.5	-2	3	
	329.5	3	-4	
	332.75	-3	4	
	351.9	4	-5	
	354.5	-4	5	
	438.25	2	-2	
	439.33	3	-3	
	440.2	5	-5	
	440.57	7	-7	
	442.8	-5	5	
	443.67	-3	3	
	444.75	-2	2	
	889.5	-2	1	

Table 5.2 – K-State Microtransceiver Spurious Response, IF= 6.5 MHz

Figure 5.1 shows a number of spurious responses for both IF frequencies, including all the expected known problems at subharmonics or from mixing spurs or image frequencies. Upon review, it is also clear that many of those responses *occur at different frequencies*. Furthermore, many of the unexplained responses also seem to differ between the two tests. To emphasize this point Figure 5.2 shows the results from this test in two forms: first, the graph on the left shows both spurious responses on the same plot, and second, the graph on the right shows the two responses with all spurious responses in common between the two tests removed.

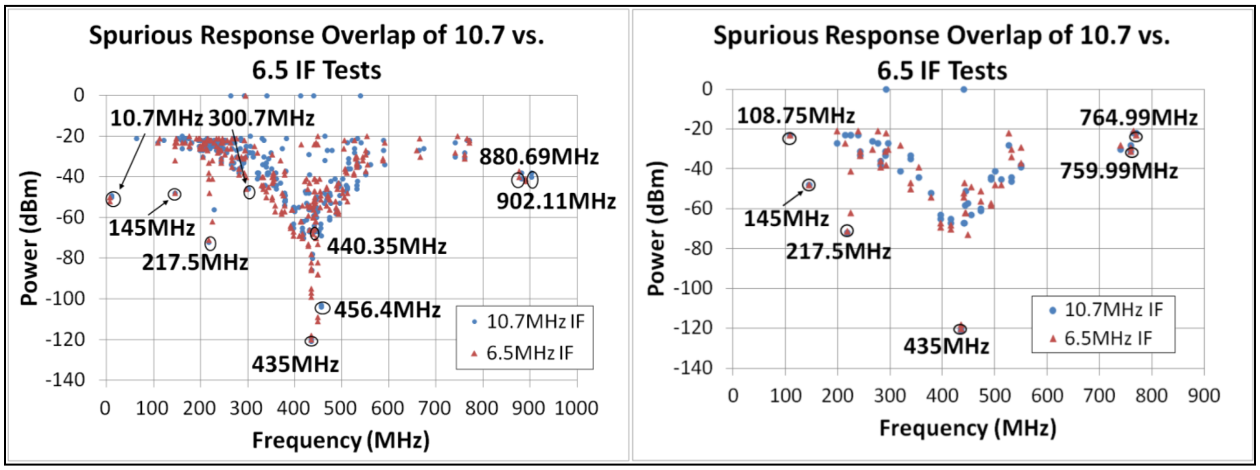


Figure 5.2– K-State Microtransceiver Varied IF Comparison Spur Residual

It's obvious looking at these two plots that the number of spurious responses has been decreased dramatically. All the mixer spurs and the image frequency are gone as expected, but the majority of the unexplained responses are also removed. This is a strong indication that a variable IF frequency could significantly improve a receiver's spurious response if implemented intelligently. Table 5.3 shows an exhaustive list of the spurs which remain after eliminating the responses which are in common within ±100 kHz.

K-State Microtransceiver

IF = 10.7 MHz		IF = 6.5 MHz	
Frequency (MHz)	**Spur (dBm)**	**Frequency (MHz)**	**Spur (dBm)**
108.75	-23	**108.75**	-23
145	-48	**145**	-48
198.61	-27	**198.61**	-21
214.35	-23	**214.4**	-27
217.5	-72	**217.5**	-71
223.92	-23	**223.9**	-41
223.95	-23	**224**	-62
238.35	-23	**238.35**	-21
242.99	-33	**242.99**	-33
243	-31	**243**	-31
266.35	-26	**266.24**	-30

276.6	-27	**276.59**	-21
281.39	-38	**281.39**	-39
281.4	-36	**281.4**	-36
290.99	-33	**290.99**	-31
291	-31	**292.1**	-22
292.2	0	**292.23**	-38
295.35	-27	**295.45**	-30
319.79	-41	**319.79**	-28
338.99	-35	**338.99**	-50
339	-33	**339**	-47
354.42	-44	**354.46**	-39
377.4	-52	**377.4**	-54
396.59	-65	**396.59**	-69
396.6	-63	**396.6**	-67
415.79	-67	**415.79**	-70
415.8	-65	**415.8**	-68
434.99	-121	**434.99**	-120
435	-120	**435**	-118
440.16	0	**440.2**	-54
440.35	-67	**440.24**	-52
442.13	-67	**442.2**	-47
443.56	-58	**443.6**	-62
444.51	-51	**444.65**	-62
447.84	-57	**447.88**	-73
454.2	-63	**454.39**	-49
473.39	-61	**473.39**	-56
473.4	-60	**473.4**	-51
492.59	-44	**492.59**	-58
492.6	-45	**492.6**	-57
500.79	-41	**500.81**	-48
511.8	-45	**511.8**	-48
526.19	-28	**526.19**	-22
530.99	-46	**530.99**	-34
531	-44	**531**	-30

550.19	-39	**550.19**	-37
550.2	-38	**550.25**	-29
739.99	-30	**739.99**	-28
759.99	-31	**759.99**	-31
760	-28	**760**	-29
764.99	-22	**764.99**	-21
769.99	-22	**769.99**	-23

Table 5.3 – K-State Microtransceiver 435MHz Tune Different IF Spur Residual

A more thorough analysis of this data could likely yield insight into the K-State Microtransceiver's spurious response. However, given the significant improvement from just varying the IF, it's possible that more than one tunable filter could be used in a receiver further improving a receiver's spurious response. The Q-enhanced filter is too noisy to be used as a preselect filter, but it could work as a narrow bandpass filter immediately preceding the mixer providing a narrowband tuned – RF capability. The next section explores this receiver architecture concept.

5.2 – A New Architecture Using the Q-Enhanced Filter

Figure 5.3 shows a modified single conversion superheterodyne receiver architecture which uses two tunable variable bandwidth filters. These filters are assumed to be the Q-enhanced filters addressed earlier in this thesis.

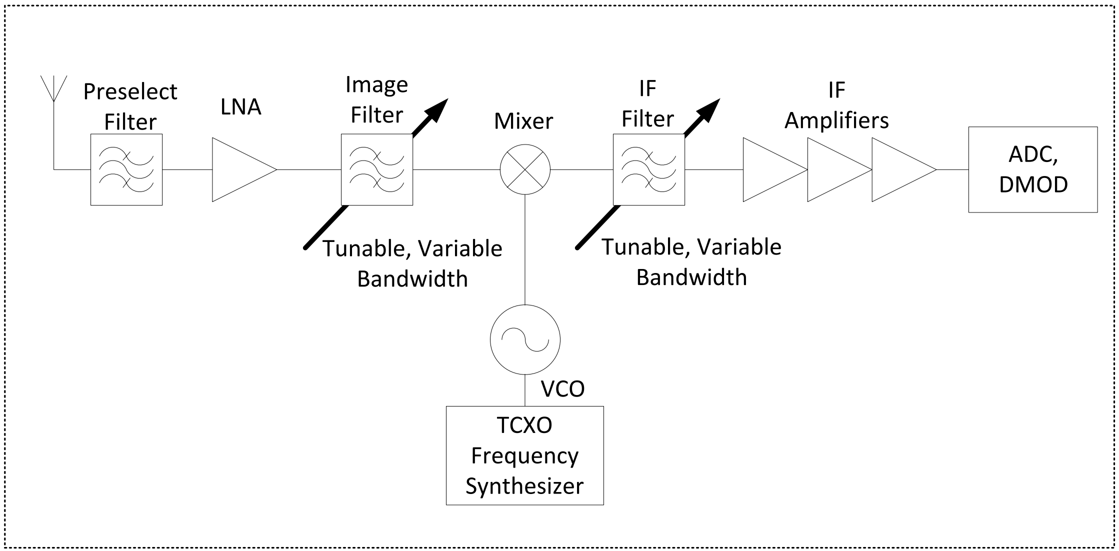

Preselect Filter
LNA
Image Filter
Mixer
IF Filter
IF Amplifiers
ADC, DMOD
Tunable, Variable Bandwidth
Tunable, Variable Bandwidth
VCO
TCXO Frequency Synthesizer

Figure 5.3 – Modified Superheterodyne Architecture Using Q-Enhanced Filters

This architecture would use Q-enhanced as both as an image reject filter and as an IF filter. Tunable filters at these nodes in the receiver would allow significant tunability enabling the same spurious response avoidance capability as shown in section 5.1. Additionally, these filters could provide very narrow fractional bandwidths further reducing the production of spurs. It's important to recognize that these filters would need to be tuned by an intelligent algorithm which avoided spurious responses as it tuned to receive the desired signal. The full nature of this algorithm isn't addressed here, but is mentioned to explain that the potential of this receiver architecture could only be realized with adaptive control of the tunable filters.

Chapter 6 - Conclusion

6.1 System Status Summary

6.1.1 Integrated Circuit Redesign

The Q-enhanced variable bandwidth tunable filter integrated circuit was fully redesigned in a new integrated circuit technology. Problems in the existing design with the frequency divider and amplitude detector circuitry were addressed and solutions implemented. The incorrect loading of the capacitive coupling circuitry in the current design was fixed in the new design. Resistance tuning was implemented in the new design to cancel asymmetry in the passband due to inductive coupling. The biasing of the Q-tuning cells was altered and the gain of the differential cores was dropped to improve the dynamic range at higher Q enhancements. The new designs were simulated to test for functionality.

6.1.2 Software Development

The automated tuning algorithm was improved first to include fine tuning and then to include an optimized binary search routine for determining the fine and course tuning values. The improved algorithm was able to achieve a 2.5MHz bandwidth and maintain that bandwidth when heated from room temperature to 75ºC. The microcontroller code was prepared for the resistance tuning by modifying the portion of the algorithm which programs the filter and receives commands from the test application. Changes needed in the test application code were enumerated, but not implemented.

6.2 Receiver Spurious Response Conclusions

A novel graphical description of receiver's spurious responses was obtained through a test system developed to characterize receivers. Four receivers were tested using this system and their responses were compared and analyzed. The graphical description provides an intuitive understanding of a receiver's abilities to reject frequencies it is not tuned to receive. These test results emphasize that a dominant issue in receiver spur block capability is subharmonics rejection and IF feedthrough. These test results also indicate that the Q-enhanced filter could provide a viable improvement in receiver architectures. A potential receiver architecture was proposed and its viability as a way to improve spurious response rejection in receivers was tested using the K-State microtransceiver.

6.3 Future Work

6.3.1 Filter Layout

The new filter design is due to be fabricated in June. The final schematic design should be reviewed and the layout completed. Additionally, all simulations should be repeated with the extracted data from layout included to account for parasitics and ensure a robust design.

6.3.2 Filter Testing

The new filter should be tested thoroughly to ensure basic functionality, observe if the design corrections were successful in mitigating flaws in the current design, and fully determine the viability of this filter as a potential solution in receiver architectures. A new embedded board should be fabricated to place inductors close together to allow the automatic coupling adjustment algorithms to be researched and developed.

6.3.3 Software

The code changes in the microcontroller should be tested by programming the new chip. Further optimization is possible in the tuning algorithm and increases in tuning speed might be achievable and should be explored. The test application needs to have some additional functionality incorporated to fully test and explore the behavior of the filter. Additional error checking and event handling would be useful in both systems to ensure robust functionality.

6.3.3.1 Fine Tuning Code Future Work

The fine tuning C code on the microcontroller is fully implemented. However, both the course and fine tuning portions of the algorithm currently use linear searches to find the best value and the tuning speed might be significantly increased if this search was optimized. Also, three of the blocks in Figures 3.2-3.5 were highlighted by the use of bold dashed outlines. These blocks should be unnecessary with the successfully redesigned filter so that these values can be to zero when testing, assuming the sensitivity issues in the amplitude detector are corrected.

The fine tuning code required modifying only two files in the original microcontroller code. The original code is documented entirely in [2], but the modified C code is contained in appendix B. The fine tuning code currently makes use only of the settings from the filter test application documented in [2]. However, the frequency tolerance and amplitude detection thresholds for the front and back ends are used in the course tuning portions of the algorithm. It would be useful to extend the functionality of the GUI to control the fine tuning directly by implementing additional thresholds for the frequency amplitude thresholds in the commands sent to the microcontroller.

The C code on the microcontroller needs to be modified to include additional bits in the control word used to program the filter. The GUI should be modified to include the ability to tune the resistance values. Eventually the automated algorithm should be automated to include a look up table which chooses the resistance based on the desired center frequency and bandwidth.

Chapter 7 - Bibliography

[1] R. Strouts, *Automatic Tuning of Q-Enhanced Integrated Differential Bandpass Filters in a Silicon-On-Sapphire Process,* Manhattan, KS: Dept. Elect. Eng., Kansas State University, 2009.

[2] J. Schonberger, *Fourth-Order Q-Enhanced Band-Pass Filter Tuning Algorithm Implementation and Considerations*, Manhattan, KS: Dept. Elect. Eng., Kansas State University, 2010.

[3] B. Kuhn, "Fully integrated bandpass filters for wireless transceivers - Problems and Promises," *Proc. IEEE Midwest Symp. Circuits and Systems*, pp. 69-72, 2002.

[4] A. Boutz, *Inductors in LTCC Utilizing Full Tape Thickness Features,* Manhattan, KS: Dept. Elect. Eng., Kansas State University, 2009.

[5] A. Boutz and B. Kuhn, "Measurement and Performance of Embedded LTCC Inductors Utilizing Full Tape Thickness Feature Conductors," *IMAPS/CICMT,* January 2009.

[6] M. Marcus, "The Future of Sharing Satellite Downlink Bands with Terrestrial Communications," in *2012 IEEE Radio & Wireless Week*, Santa Clara, CA, 2012.

[7] W. B. Kuhn, N. Yanduru and A. Wyszynski, "Q-Enhanced LC Bandpass Filters for Integrated Wireless Applications," *IEEE Transactions on Mricrowave Theory and Techiques,* vol. 46, no. 12, pp. 2577-2586, Dec 1998.

[8] R. Duncan, K. MArtin and A. Sedra, "A Q-Enhanced Active-RLC Bandpass Filter," in *IEEE Int Symp. on Circuits and Systems*, 1993.

[9] W. Kuhn, D. Nobbe, D. Kelly and A. Orsborn, "Dynamic range performance of on-chip RF bandpass filters," *IEEE Transactions on Circuitrs and Systems II: Analog and Digital Signal Processing,* vol. 50, no. 10, pp. 685-694, Oct. 2003.

[10] R. b. S. Ford, "WiNRADiO WR-G31 DDC Excalibur Software Defined Receiver," *QST,* pp. 48-51, January 2012.

[11] T. S. Rappaport, Wireless Communications, Upper Saddle River, NJ: Prentice-Hall, Inc, 2002.

[12] Unknown, "Mixer Spur Chart," P-N Designs. Inc, 12 January 2010. [Online]. Available: http://www.microwaves101.com/encyclopedia/mixer_spurs.cfm. [Accessed 20th April 2012].

[13] R. G. Huenemann, "Receiver Spurious Response Measurements," *IEEE Trans. On Communication Technology,* vol. 17, no. 3, pp. 417-419, 1969.

[14] R. G. Huenemann and C. L. R. Chapman, "Automatic Receiver Spurious Response Measurements - Some Preliminary Results," *IEEE Comminication Technology Group,* pp. 561-564, 1970.

[15] V. I. Mordachev, "Automated Douple- Frequency Testing Technique for Mapping Receiver Interference Responses," *IEEE Transactions on Electromagnetic Compatability,* vol. 42, no. 2, pp. 213-225, 2000.

[16] S. Melton, *Cryogenic Temperature Characteristics of Bulk Silicon and Silicon-on-Sapphire Devices,* Manhattan, KS: Dept. Elect. Eng., Kansas State University, 2012.

[17] W. B. Kuhn, W. Stephenson and A. Elshabini-Riad, "A 200 MHz CMOS Q-Enhanced LC Bandpass Filter," *IEEE Journal of Solid State Circuits,* vol. 31, no. 8, pp. 1112-1122, 1996.

[18] Unknown, "Communications Receiver VR-120, Technical Supplement," VERTEX STANDARD CO., LTD. Printed in Japan, 2001.

[19] "VX-3R Operating Manual," VERTEX STANDARD CO., LTD., Tokyo, Japan.

[20] Unknown, "Spectral Analysis Program," MOETRONIX, 2007.

[21] W. Kuhn, N. E. Lay and e. al, "A Microtransceiver for UHF Proximity Links Including Mars Surface-to-Orbit Applications," *Proceedings of the IEEE,* vol. 95, no. 10, pp. 2019-2044, 2007.

Appendix A - IC Redesign

This section of the appendix details the circuits not discussed in the body of the work. It presents the full schematic and then steps down level by level to explain the various portions of the design, including simulations and test benches where needed.

A.1 Top Level View of Q-Enhanced Filter Schematic

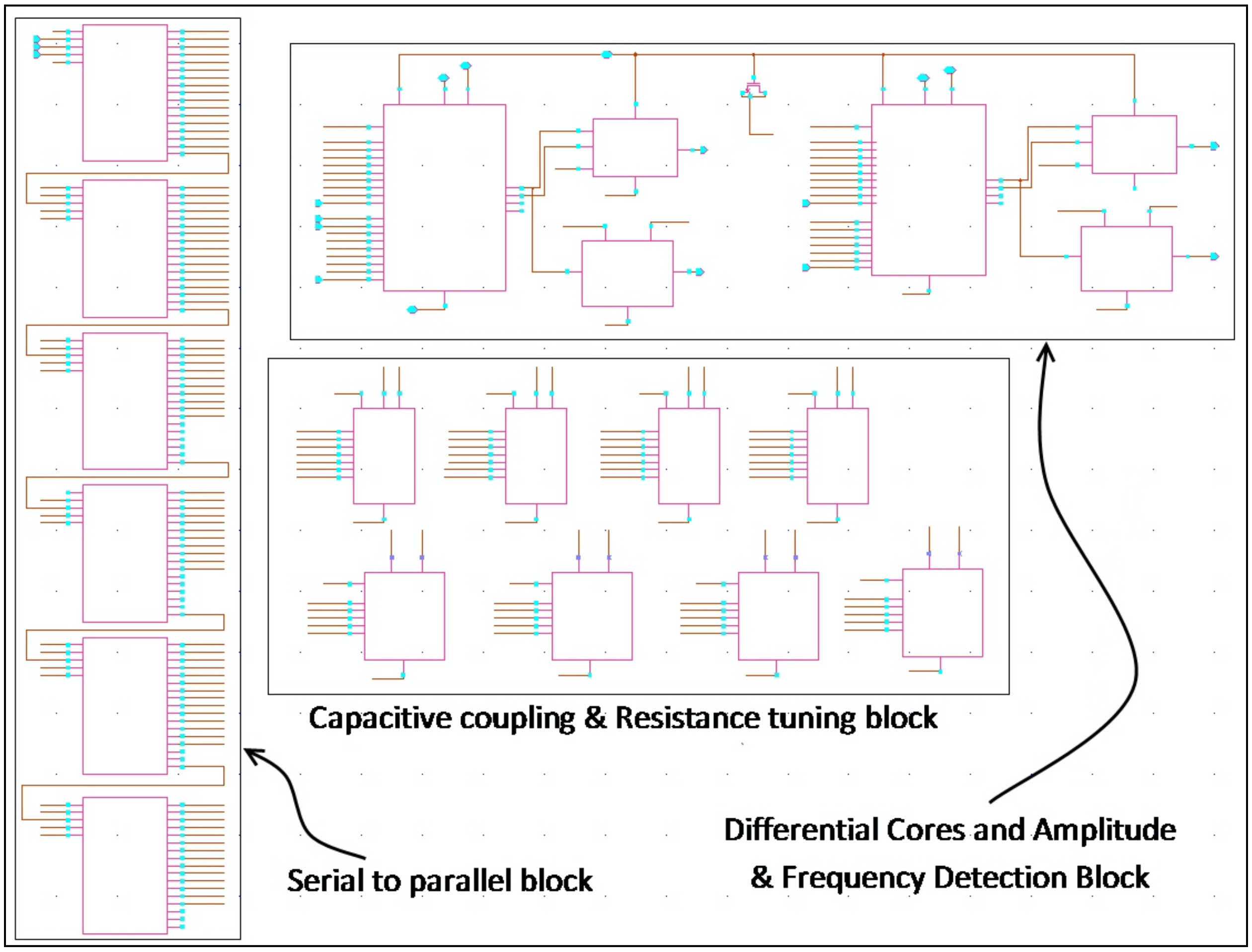

Capacitive coupling & Resistance tuning block
Serial to parallel block
Differential Cores and Amplitude & Frequency Detection Block

Figure A.1 – Top Level View of Q-Enhanced Filter Schematic

This figure shows the top view of the design as seen after stepping into the 'top_qenhanced' symbol in the design. There are three portions to this top level: the serial to parallel portion which takes in all the tuning bits and enable bits as information to set the features of the circuit, the differential cores with the amplitude and frequency detection circuits at their outputs, and the capacitive coupling tuning and resistance tuning portions of the circuit used to refine the passband shape.

A.2 Serial to Parallel Block

This figure shows a close up of the serial to parallel blocks and their wiring. The output of each DFF in the second level is tied to its enable or tuning bit so that when the latch is triggered all the bits which have been shifted into the first level of DFF are passed to the desired circuits. The full chain of DFF in each serial to parallel block is shown next.

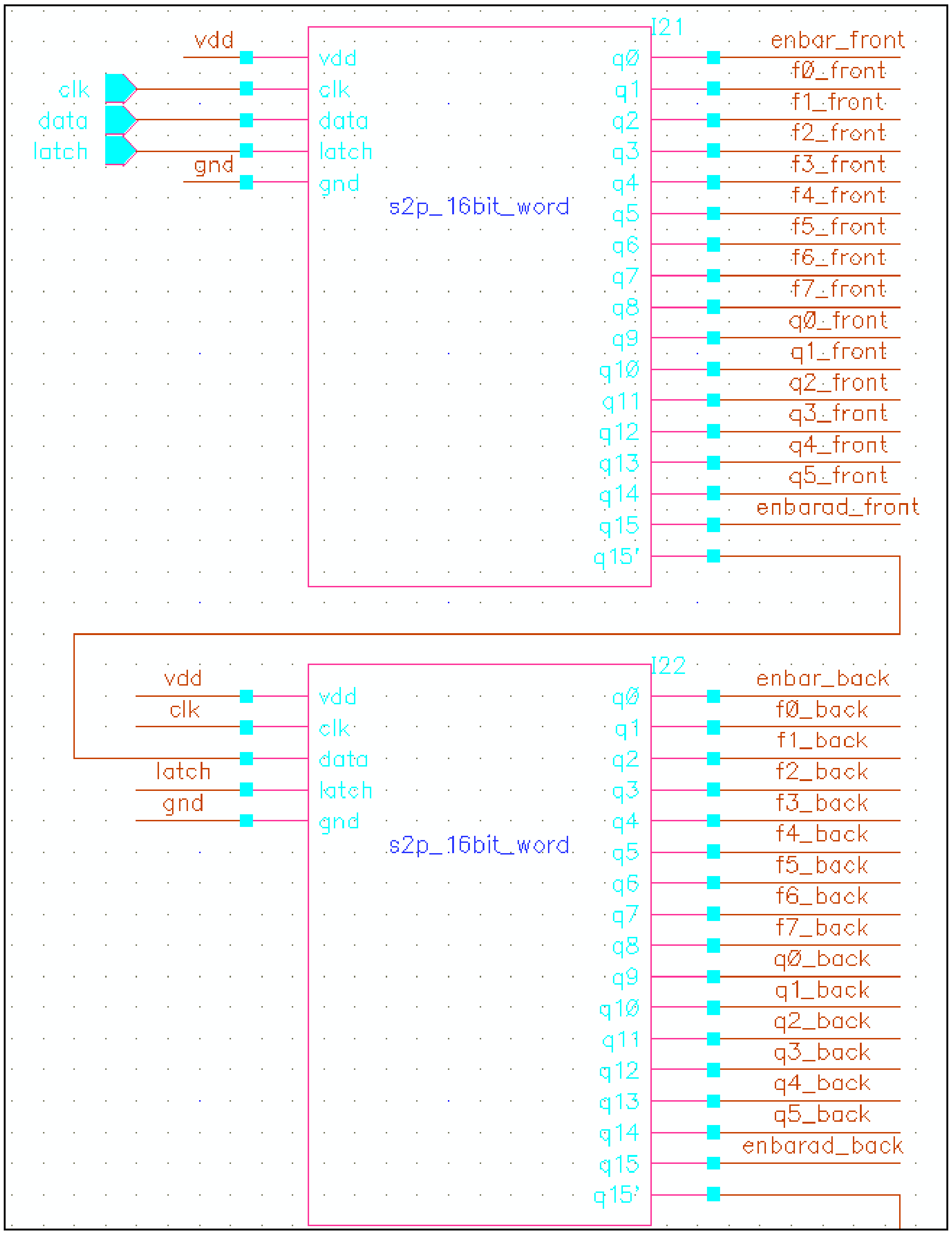


Figure A.2 – Top Level View of Serial to Parallel Register Schematic

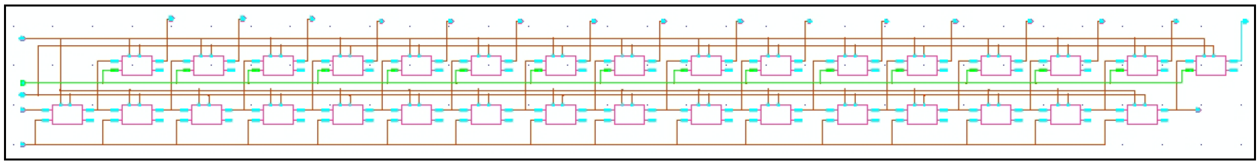


Figure A.3 – Single Cell View of Serial to Parallel Register Schematic

A close up of the first few DFF are shown next to clarify the operation and wiring of these circuits.

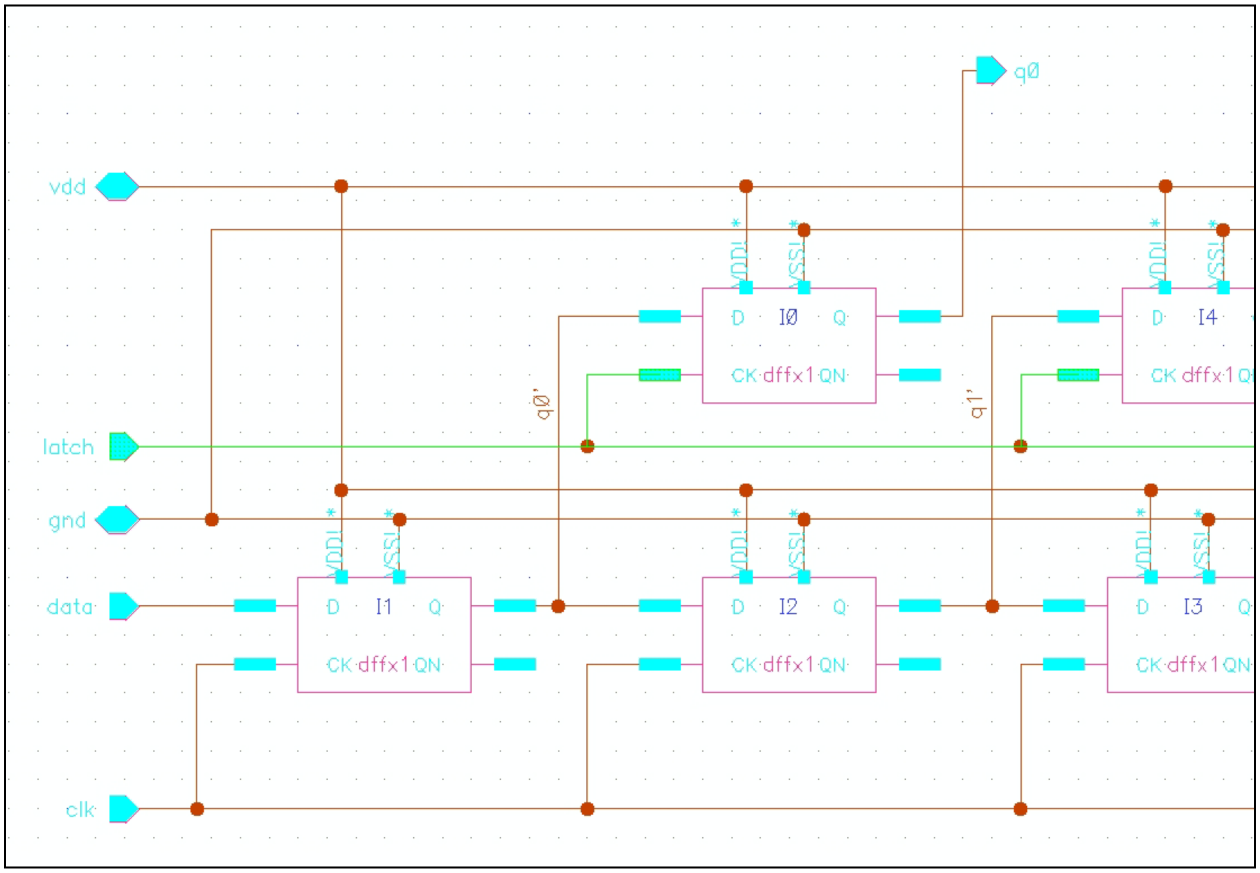


Figure A.4 – Narrowed View of Serial to Parallel Register Single Cell

This circuit was tested using the following test bench set up.

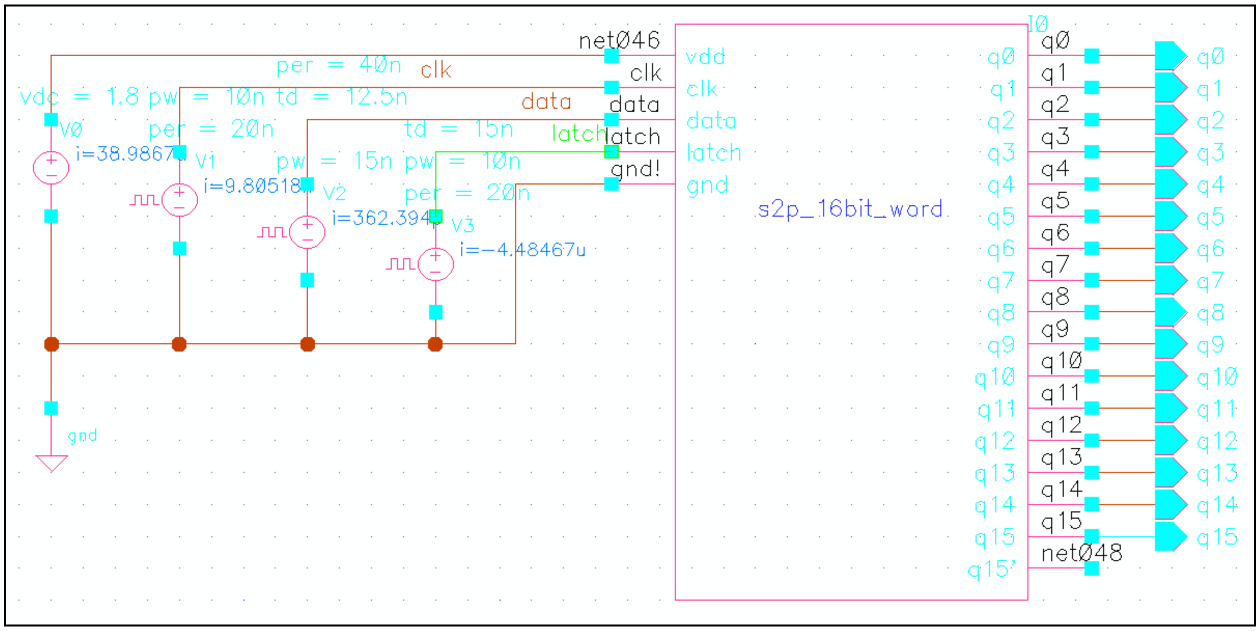


Figure A.5 – Test Bench for Serial to Parallel Register

The simulation output for this circuit is shown in the next two figures.

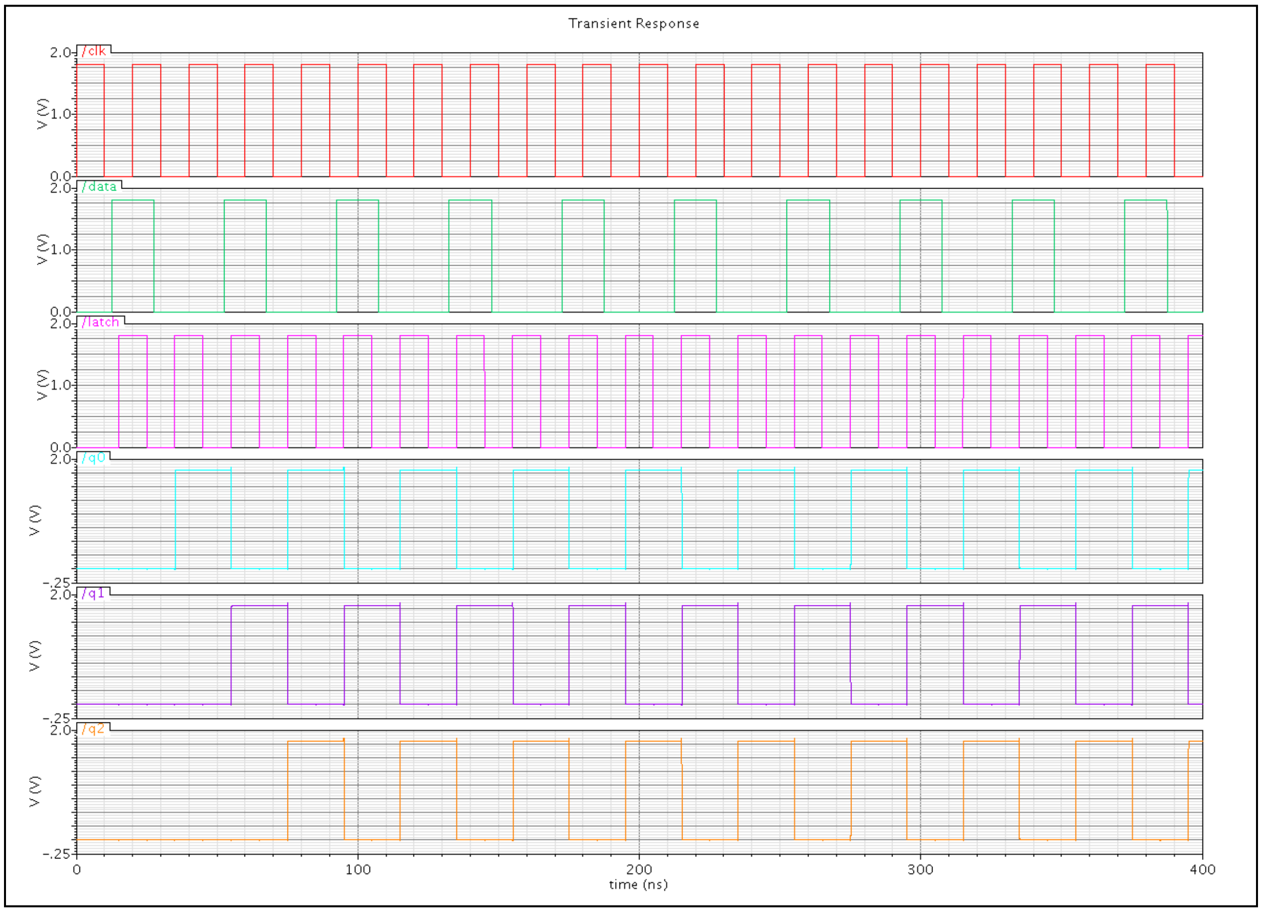


Figure A.6 – Simulation Output for Serial to Parallel Register

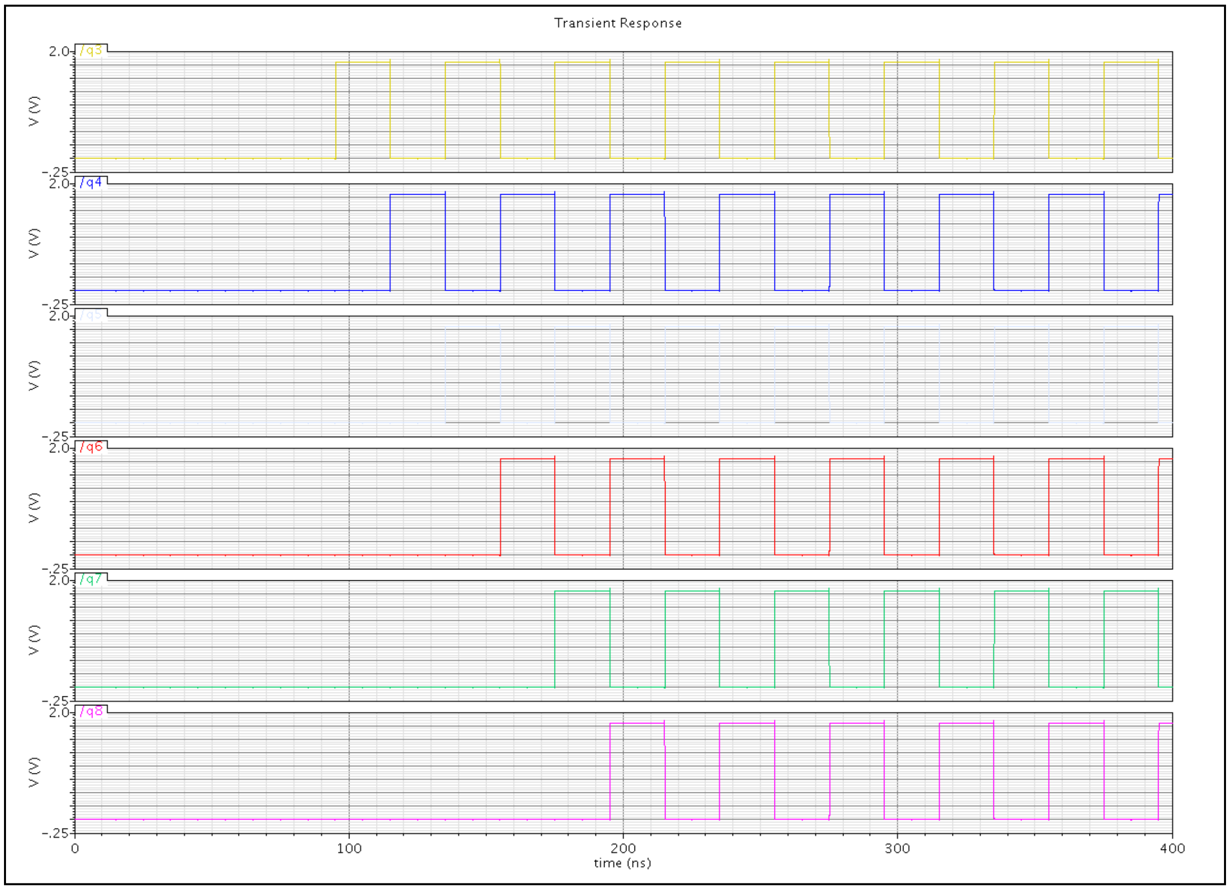


Figure A.7 – Simulation Output for Serial to Parallel Register

A.3 Differential Cores, Amplitude Detector & Frequency Divider Top View

The next figure shows a slightly closer of the differential cores and the amplitude and frequency detection circuits tied at the output. The port outputs tied into the LC tank circuits explained in other sections of this work.

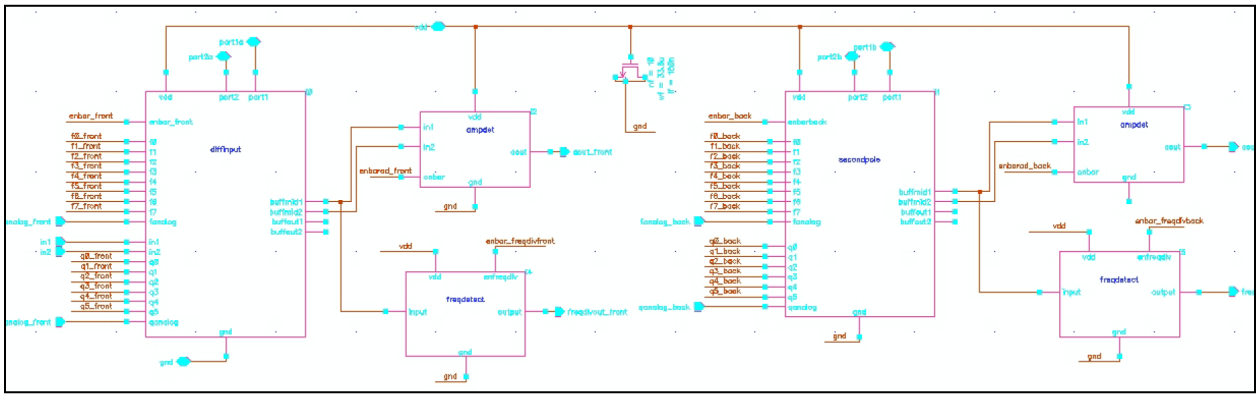


Figure A.8 – Top View of Differential Cores, Frequency Detectors, and Amplitude Detectors

A.3.1 Differential Core, Buffer, & Tuning Block

A.3.1.1 Differential Core

This figure shows the internal portion of a differential core master block which is divided into three portions: the amplifier core, the frequency tuning and Q-enhancement block and the buffer at the output. This block also includes the enable PFET which operates a switch to activate or deactivate the core.

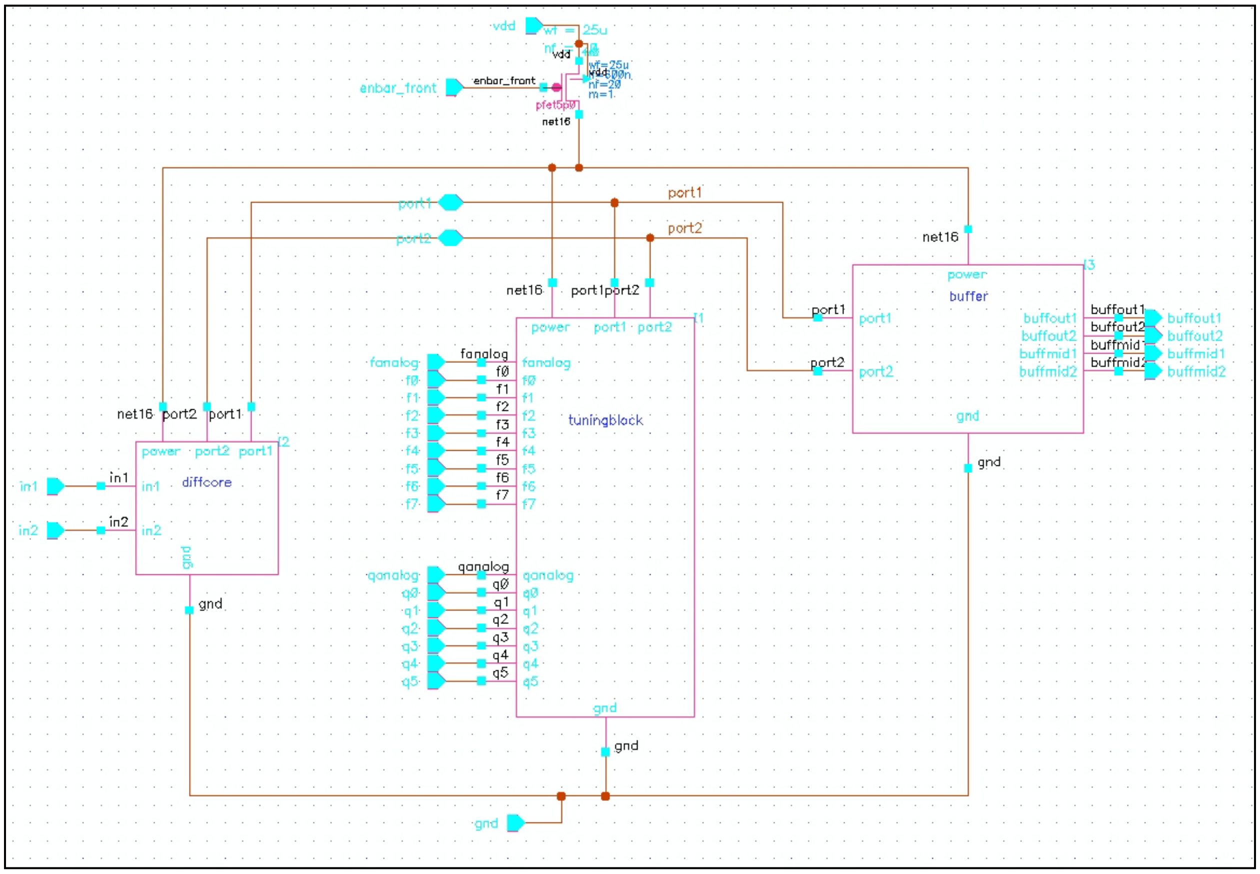


Figure A.9 –Differential Core, Tuning Block, and Buffer

The schematic of the differential core is shown above. The bias points in the circuit are set up with the stack of PFETS shown on the left. The current through the core is set to about 200mV overvoltage and is setting the total current through the core to be about 2mA. The inputs are AC coupled and biased with resistors which set the input impedance. The differential core is built of two legs of intrinsic FETs which are cascoded to drive the LC tank circuits. This circuit is designed to have an unenhanced gain of 1. This circuit is crucial to the overall performance at a system level because it is one of two circuits which limit the output voltage swing, or dynamic range, and the primary circuit which determines the noise floor. The noise floor is lowered with more current, but power consumption is increased. Additionally the gain of this core is dictated by g_m which is impacted by the current through the circuit. The voltage gain of this circuit is known to be

$$G_V = 2g_mR_p$$

where R_p is the load seen from the LC tank circuit at resonance in combination with the total load resulting from all resistances added in parallel with the load of the inductors at resonance when tuning and Q enhancements are active.

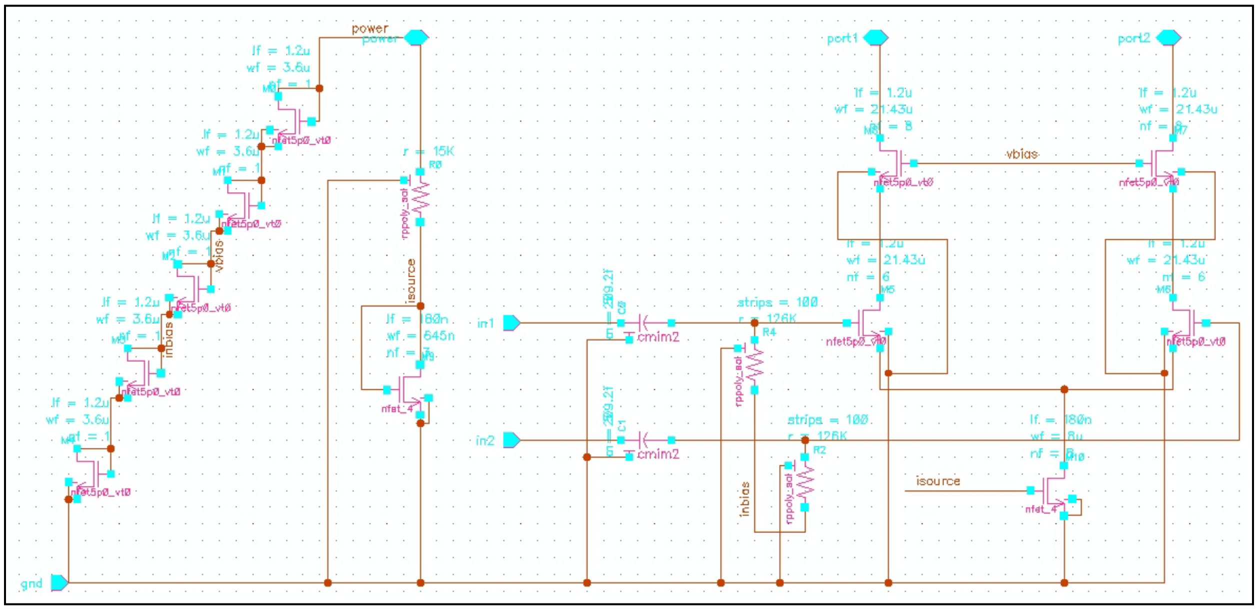


Figure A.9 –Differential Core Schematic

The test bench and simulation output for this circuit are shown below.

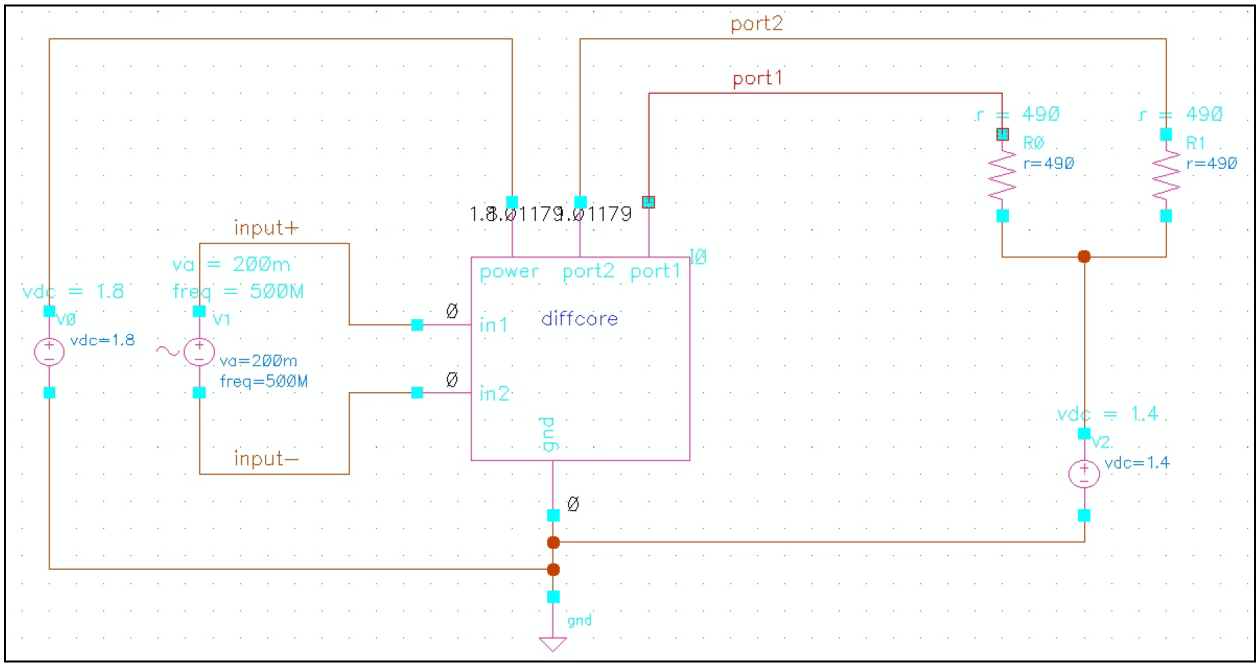


Figure A.10 –Differential Core Test Bench

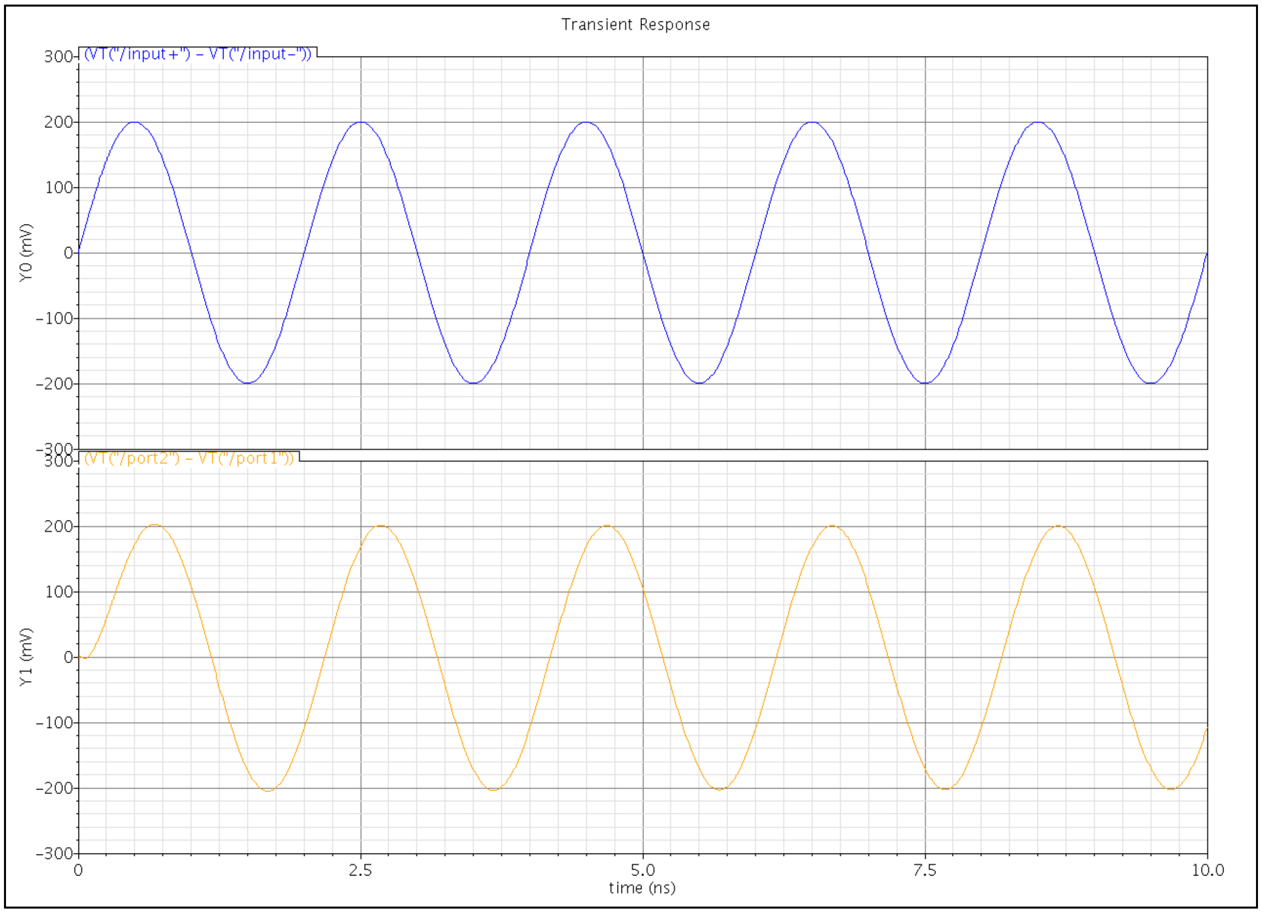


Figure A.11 – Simulation Output of Differential Core

A.3.1.2 Buffer

The next circuit is the buffer. The buffer is designed to provide a stable output to the frequency and amplitude detection circuitry without loading the core amplifier down.

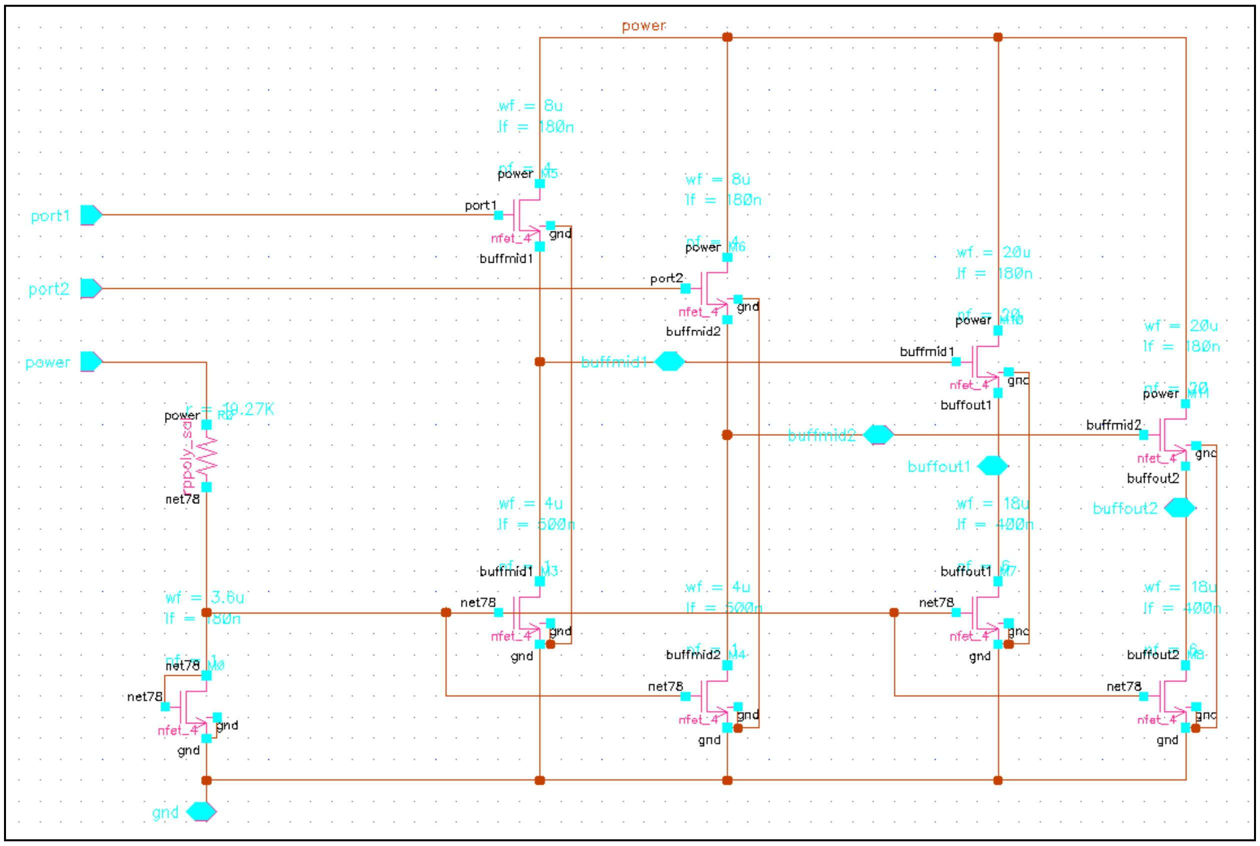


Figure A.12 – Buffer Circuit

The buffer is a differential chain of two common drain, or current follower, circuits. The overall gain is about .6 when driving a 1 kΩ load and the total current draw is about 2-3 mA. The current in each leg of the amplifier is set up by the resistor and FET using the power supplied to the circuit. The output test bench and simulation are shown next.

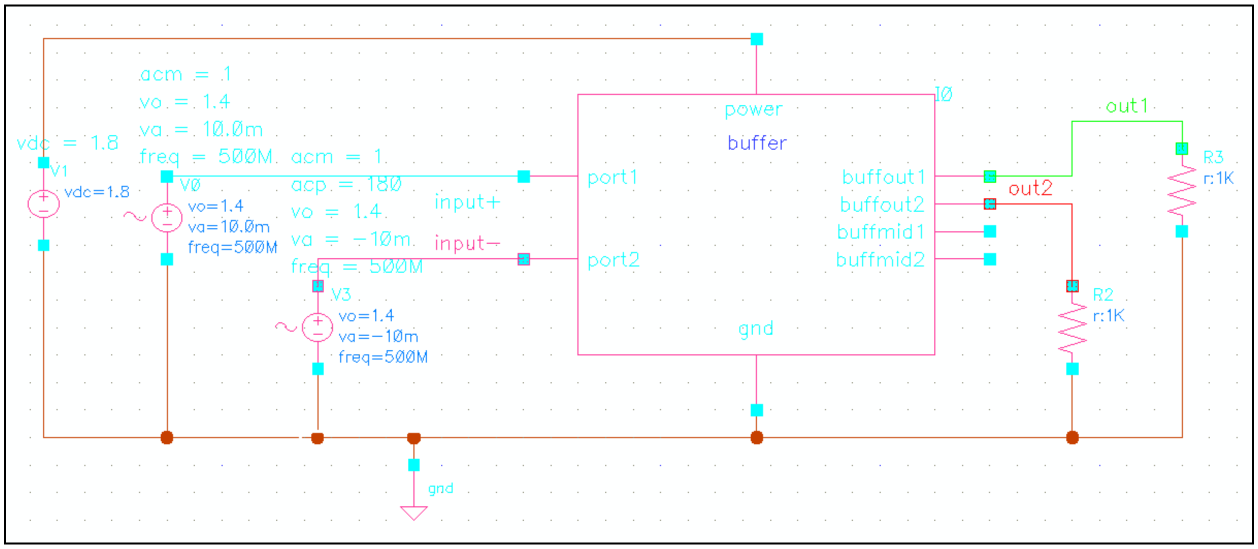


Figure A.13 – Buffer Circuit Testbench

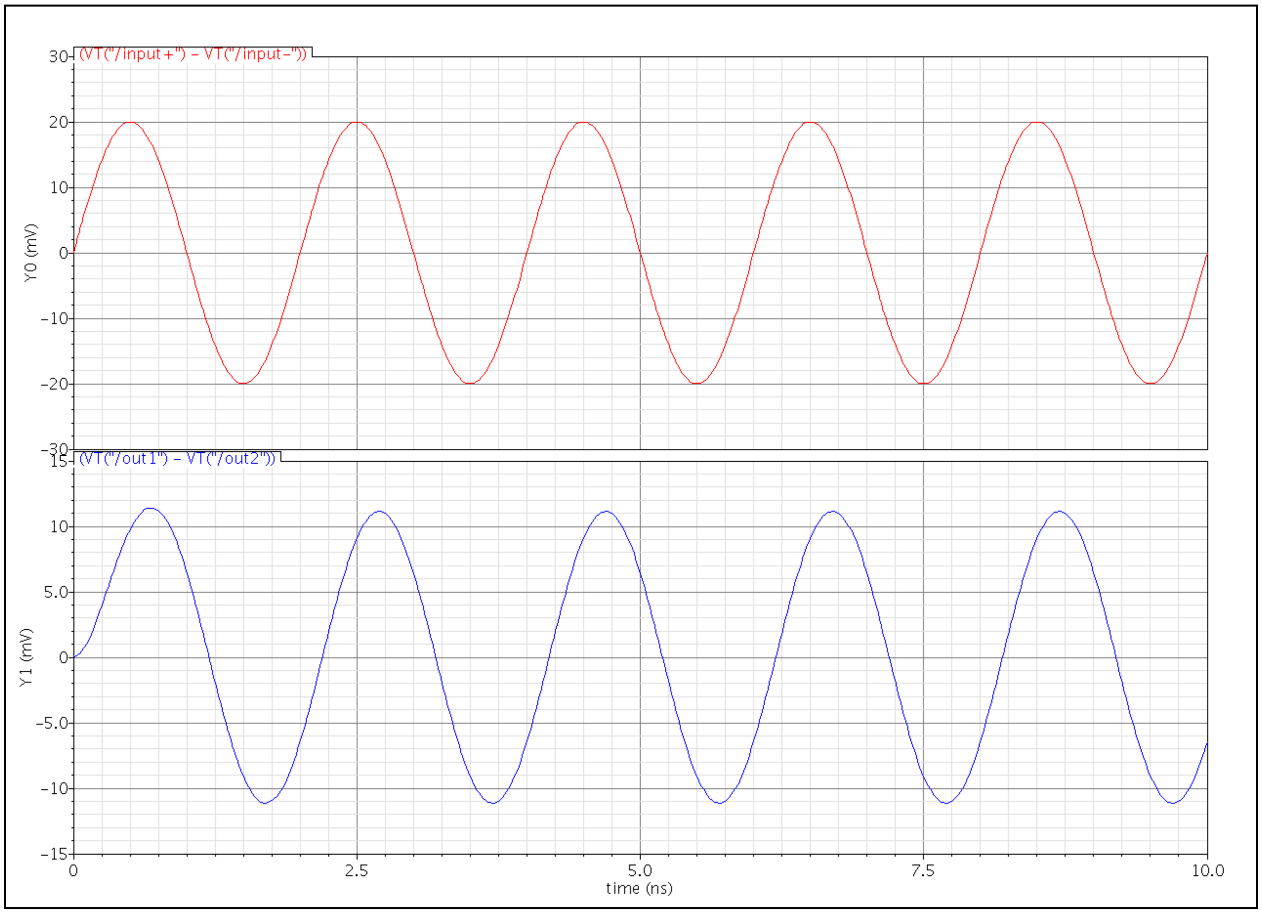


Figure A.14 – Simulation Output of Buffer Circuit

A.3.1.3 Tuning Block

The tuning circuitry used to achieve frequency tuning and Q-enhancement is shown below.

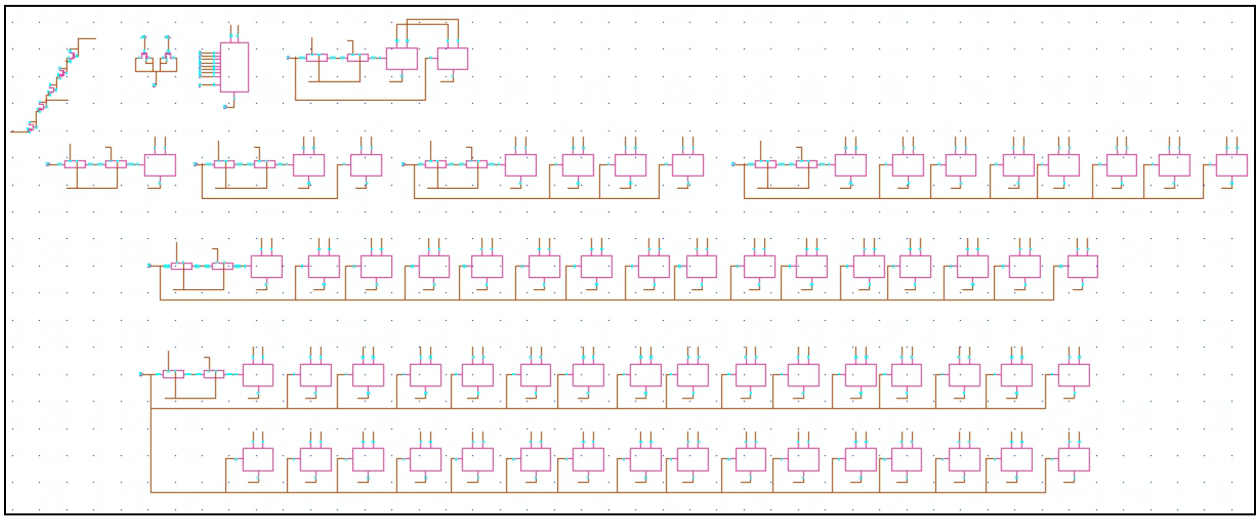


Figure A.15 – Top View of Frequency and Q-Enhancement Tuning Blocks

This is the top view of the tuning circuitry. Close ups of the sections and cells follow with relevant explanation.

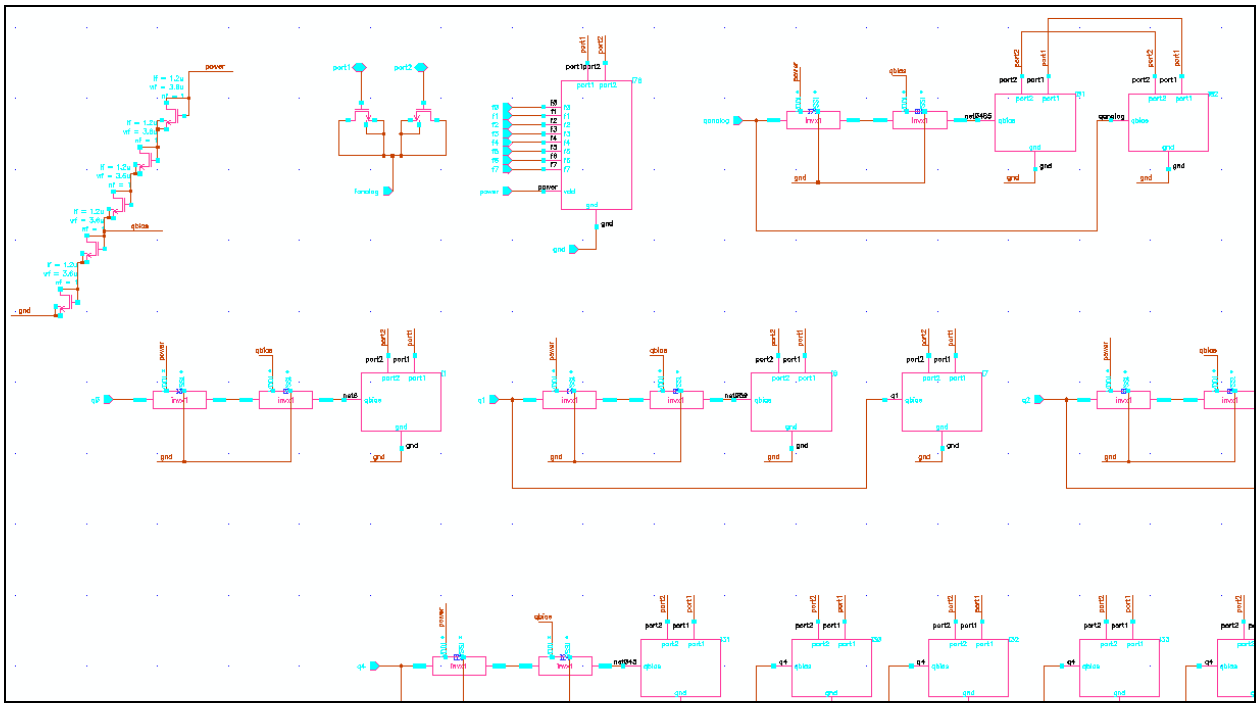


Figure A.16 – Narrowed View of Frequency and Q-Enhancement Tuning Block

This figure shows a close-up of the top left section of the full tuning block. The biasing used for these blocks and sections are shown at the very top left. The analog frequency tuning capacitor is shown next to the right, followed by the digital frequency tuning block. Finally, at the top right the analog Q tuning cell is shown. Each Q cell is run by two inverters. The Q cell is implemented in binary weighted banks.

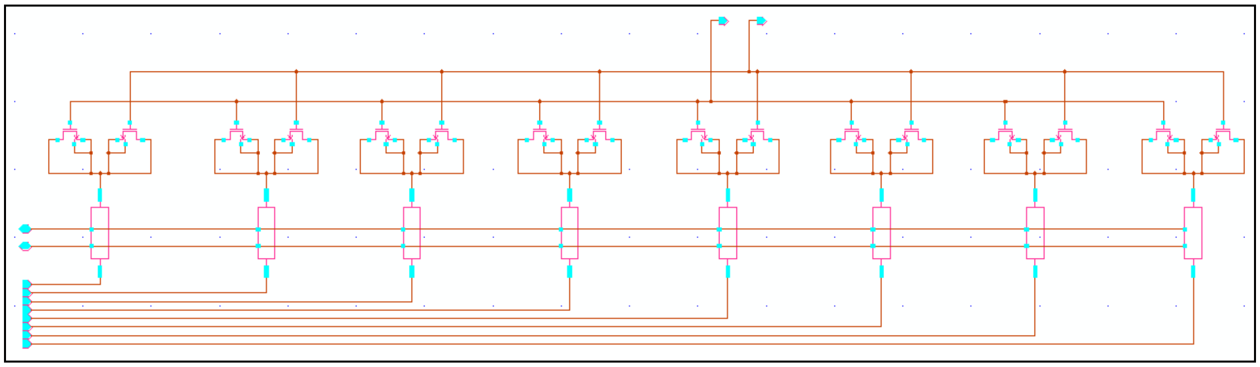


Figure A.17 – Top View of Frequency Tuning Block

The frequency tuning circuit is shown above. To simplify layout the widths and lengths of the FETs were modified to create binary weighted capacitors driven by inverters. A close-up of the first two caps is shown next.

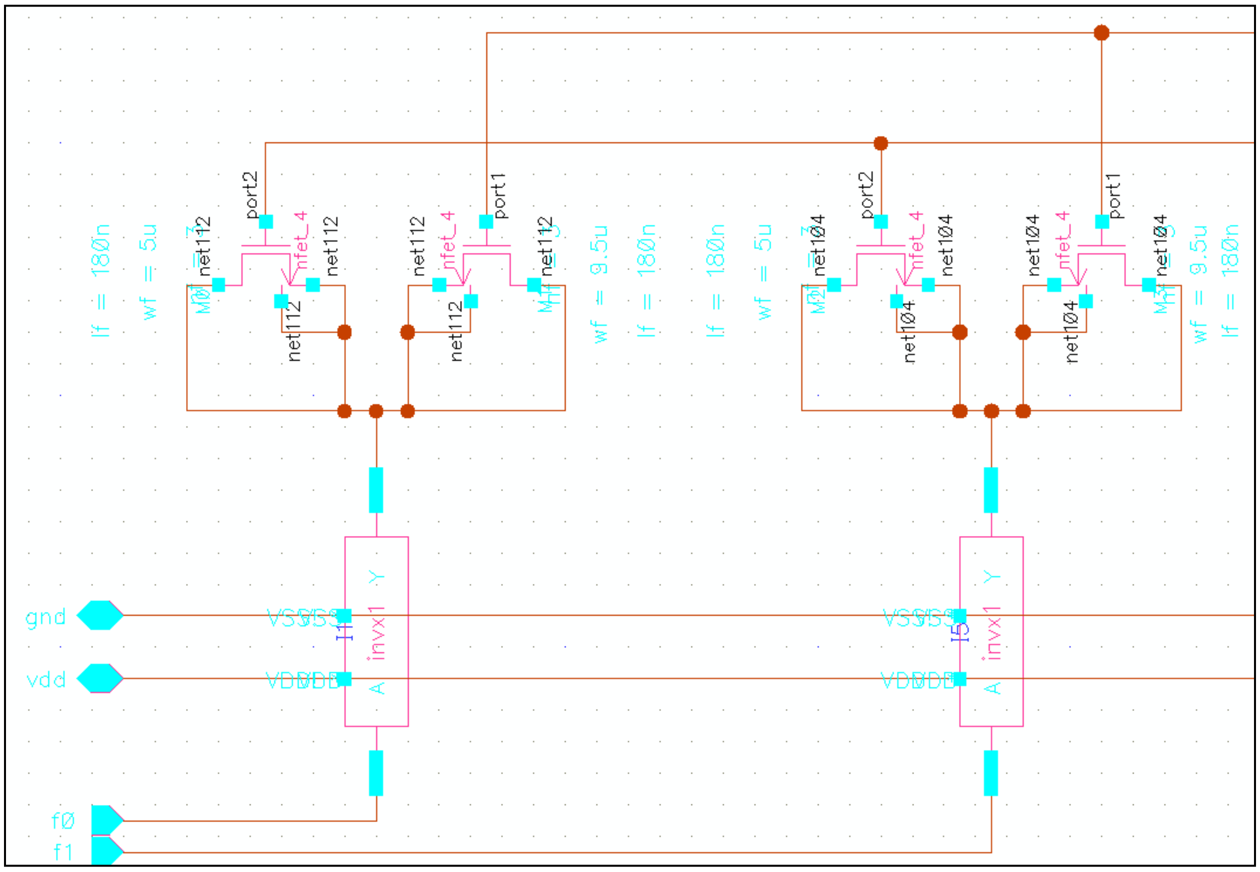


Figure A.18 – Narrowed View of Frequency Tuning Block

As shown these capacitors are implemented as FET capacitors with the drain and source tied together driven by inverters. The source and drain of the FETs are AC grounds in this case so no resistor is needed.

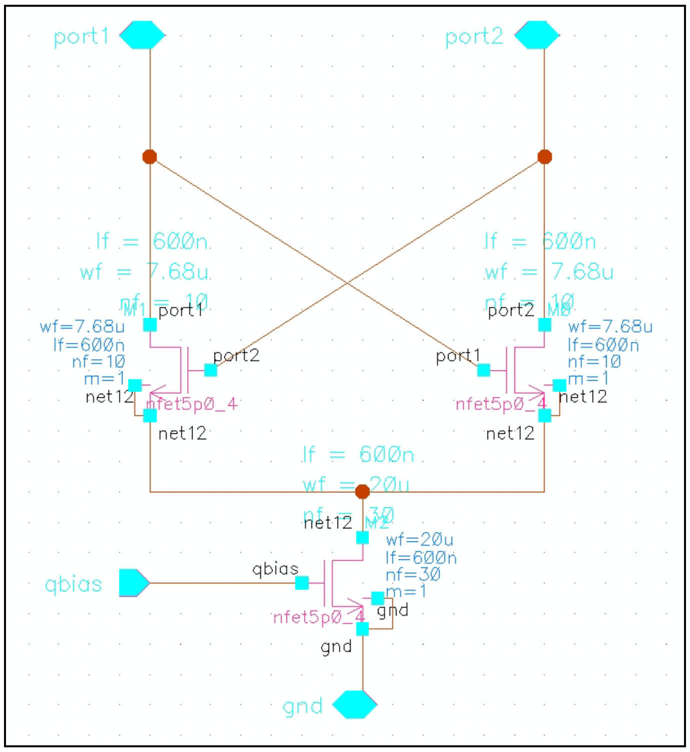


Figure A.19 – Cell View of Q-Enhancement Tuning Block

The basic Q-enhancement cell is shown above. This circuit provides the Q enhancement and is the second circuit which limits the output voltage swing, or dynamic range of the circuit. To improve the system's ability to output large signals high threshold FETs are used for the two legs of the circuit allowing the voltage at the ports to drop as much as possible before pushing these FETs out of the active region. The current source is biased with about 200mV overvoltage.

A.3.2 Resistance Tuning & Capacitive Coupling Top View

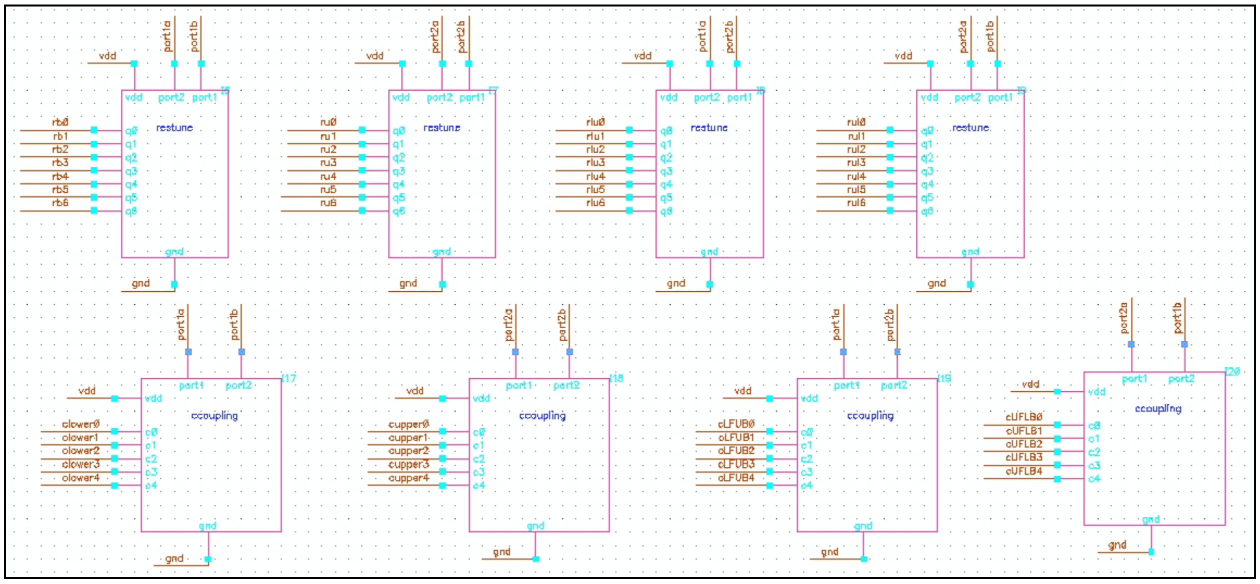


Figure A.20 – Top View of Capacitive Coupling and Resistance Tuning Blocks

This schematic is the capacitive coupling and resistive blocks shown at the top level of the schematic. The resistor coupling is explained at length in the body of the text and isn't addressed here. The capacitive coupling is also referenced in the body of the text, but for sake of thorough documentation the circuits are included here.

Appendix B - C Code

B.1 Fine Tune Code Modifications

```
/************************************************************************************
 * Filename:                    qefilter.c
 * Date:                        June 2010
 * Compiler:                    C30
 * Author:                      Joel Schonberger
 * Company:                     Kansas State University
 * Department:                  Electrical & Computer Engineering
 * Research:                    500 MHz Two-Pole Q-Enhanced Filter Tuning Algorithm
 * Description:                 This file houses the functions needed to implement the QE Filter
 *                              tuning algorithm.
 * -----------------------------------------------------------------------------------------------
 * Updated:                     April 2012
 * Author:                      Chelsi Kovala
 * Changes:                     Modified to include fine tuning in the tuning algorithm and now
                                includes the functions:
                                void fineFrontEndFTune(void)
                                void fineBackEndFTune(void)
                                void fineFrontEndQTune(void)
                                void fineBackEndQTune(void)
                                printFrontEndAnalogQTune()
                                printFrontEndAnalogFTune()
                                printBackEndAnalogQTune()
                                printBackEndAnalogFTune()
 ************************************************************************************/
#include "main.h"

/* Global Variables */
char strCenterFreq[]          = "Center Frequency: ###.# MHz\r\n";
char strFreqTol[]             = "Frequency Tolerance: ##.# MHz\r\n";
char strFrontEndADThresh1[]   = "Front-End AD Threshold 1: ###\r\n";
char strFrontEndADThresh2[]   = "Front-End AD Threshold 2: ###\r\n";
char strFrontEndQOffset[]     = "Front-End Q-Offset: ##\r\n";
char strFrontEndQBackOff[]    = "Front-End Q-BackOff: ##\r\n";
char strFrontEndFOffset[]     = "Front-End F-Offset: ##\r\n";
char strBackEndADThresh1[]    = "Back-End AD Threshold 1: ###\r\n";
char strBackEndADThresh2[]    = "Back-End AD Threshold 2: ###\r\n";
char strBackEndQOffset[]      = "Back-End Q-Offset: ##\r\n";
char strBackEndQBackOff[]     = "Back-End Q-BackOff: ##\r\n";
char strBackEndFOffset[]      = "Back-End F-Offset: ##\r\n";
char strCouplingUpper[]       = "Coupling Upper: ##\r\n";
char strCouplingLower[]       = "Coupling Lower: ##\r\n";
char strCouplingUFLB[]        = "Coupling UFLB: ##\r\n";
char strCouplingLFUB[]        = "Coupling LFUB: ##\r\n";
char strFrontEndAD[]          = "Front-End Amp Detector: ####\r\n";
char strFrontEndNonOsc[]      = "Front-End Non-Osc: ####\r\n";
char strFrontEndFCnt[]        = "Front-End Freq Count: ###.# MHz\r\n";
char strFrontEndDigitalQTune[] = "Front-End Digital Q-Tune: ##\r\n";
char strFrontEndAnalogQTune[]  = "Front-End Analog Q-Tune: ####\r\n";
char strFrontEndDigitalFTune[] = "Front-End Digital F-Tune: ###\r\n";
char strFrontEndAnalogFTune[]  = "Front-End Analog F-Tune: ####\r\n";
char strBackEndAD[]           = "Back-End Amp Detector: ####\r\n";
char strBackEndNonOsc[]       = "Back-End Non-Osc: ####\r\n";
char strBackEndFCnt[]         = "Back-End Freq Count: ###.# MHz\r\n";
char strBackEndDigitalQTune[]  = "Back-End Digital Q-Tune: ##\r\n";
char strBackEndAnalogQTune[]   = "Back-End Analog Q-Tune: ####\r\n";
char strBackEndDigitalFTune[]  = "Back-End Digital F-Tune: ###\r\n";
char strBackEndAnalogFTune[]   = "Back-End Analog F-Tune: ####\r\n";

int FrontEndAD, BackEndAD, FrontEndFCnt, BackEndFCnt, FrontEndNonOsc, BackEndNonOsc;
int PrevFrontEndDigitalQTune, PrevBackEndDigitalQTune;
int filterData[4];
int CenterFreq, FreqTol;
int FrontEndADThresh1, FrontEndADThresh2, FrontEndQOffset, FrontEndQBackOff, FrontEndFOffset;
int BackEndADThresh1, BackEndADThresh2, BackEndQOffset, BackEndQBackOff, BackEndFOffset;
int CouplingUpper, CouplingLower, CouplingUFLB, CouplingLFUB;

struct {
   unsigned int en:1;
   unsigned int fTune:8;
   unsigned int qTune:6;
   unsigned int ADen:1;
} FENDCON;

struct {
   unsigned int en:1;
   unsigned int fTune:8;
   unsigned int qTune:6;
   unsigned int ADen:1;
} BENDCON;

struct {
   unsigned int upper:5;
   unsigned int lower:5;
   unsigned int FDFen:1;
} CAPCON1;

struct {
   unsigned int UFLB:5;
   unsigned int LFUB:5;
   unsigned int FDBen:1;
} CAPCON2;

struct {
   unsigned int FANAF:10;
   unsigned int FANAQ:10;
   unsigned int BANAF:10;
   unsigned int BANAQ:10;
} ANALOG;

struct {
   unsigned int ADFen:1;
   unsigned int FDFen:1;
   unsigned int ADBen:1;
   unsigned int FDBen:1;
   unsigned int RFOn:1;
} DEBUG;

extern const char *console_str_sep;
extern int _PrintFilterSettings;

/*************************************************************************************
 * Function:          initFilter
 * Parameters:        void
 * Return:            void
 * Description:       Initializes the Filter Controls to their minima and disables all
 *                    of the enable variables.
 *************************************************************************************/
void initFilter(void)
{
   disableFrontEnd();
   setFrontEndDigitalFTune(FTUNE_DIG_MIN);
   setFrontEndDigitalQTune(QTUNE_DIG_MIN);
   disableFrontEndAD();
   disableFrontEndFD();

   disableBackEnd();
   setBackEndDigitalFTune(FTUNE_DIG_MIN);
   setBackEndDigitalQTune(QTUNE_DIG_MIN);
   disableBackEndAD();
   disableBackEndFD();

   setCouplingUpper(COUPLING_MIN);
   setCouplingLower(COUPLING_MIN);
```

```
    setCouplingUFLB(COUPLING_MIN);
    setCouplingLFUB(COUPLING_MIN);

    setFrontEndAnalogFTune(ANALOG_MIN);
    setFrontEndAnalogQTune(ANALOG_MIN);
    setBackEndAnalogFTune(ANALOG_MIN);
    setBackEndAnalogQTune(ANALOG_MIN);

    DEBUG.ADFen = 0;
    DEBUG.FDFen = 0;
    DEBUG.ADBen = 0;
    DEBUG.FDBen = 0;
    DEBUG.RFOn = 1;
}

/*************************************************************************************
 * Function:          printFilterOptions
 * Parameters:        void
 * Return:            void
 * Description:       Prints each of the the filter options to the Console.
 *************************************************************************************/
void printFilterOptions(void)
{
    printSeperator();
    printFrontEndStatus();
    printFrontEndADStatus();
    printFrontEndFDStatus();
    printFrontEndDigitalQTune();
    printFrontEndAnalogQTune();
    printFrontEndDigitalFTune();
    printFrontEndAnalogFTune();
    printBackEndStatus();
    printBackEndADStatus();
    printBackEndFDStatus();
    printBackEndDigitalQTune();
    printBackEndAnalogQTune();
    printBackEndDigitalFTune();
    printBackEndAnalogFTune();
    printCouplingUpper();
    printCouplingLower();
    printCouplingUFLB();
    printCouplingLFUB();
    printRFSwitchStatus();
    printSeperator();
}

/*************************************************************************************
 * Function:          printAlgorithmOptions
 * Parameters:        void
 * Return:            void
 * Description:       Prints each of the the algorithm options to the console.
 *************************************************************************************/
void printAlgorithmOptions(void)
{
    printSeperator();
    printCenterFreq();
    printFreqTol();
    printFrontEndADThresh1();
    printFrontEndADThresh2();
    printFrontEndQOffset();
    printFrontEndQBackOff();
    printFrontEndFOffset();
    printBackEndADThresh1();
    printBackEndADThresh2();
    printBackEndQOffset();
    printBackEndQBackOff();
    printBackEndFOffset();
    printCouplingUpper();
    printCouplingLower();
    printCouplingUFLB();
```

```
   printCouplingLFUB();
   printSeperator();
}

/*************************************************************************************
 * Function:              updateFilterData
 * Parameters:            void
 * Return:                void
 * Description:           Formats and stores the filter options to be programmed.
 *************************************************************************************/
void updateFilterData(void)
{
   filterData[0] = ((CAPCON2.FDBen & 1) << 10)              |
                ((CAPCON2.UFLB & COUPLING_MAX) << 5 )    |
                (CAPCON2.LFUB & COUPLING_MAX);
   filterData[1] = ((CAPCON1.FDFen & 1) << 10)              |
                ((CAPCON1.lower & COUPLING_MAX) << 5)    |
                (CAPCON1.upper & COUPLING_MAX);
   filterData[2] = ((BENDCON.ADen & 1) << 15)               |
                ((BENDCON.qTune & QTUNE_DIG_MAX) << 9)   |
                ((BENDCON.fTune & FTUNE_DIG_MAX) << 1)   |
                (BENDCON.en & 1);
   filterData[3] = ((FENDCON.ADen & 1) << 15)               |
                ((FENDCON.qTune & QTUNE_DIG_MAX) << 9)   |
                ((FENDCON.fTune & FTUNE_DIG_MAX) << 1)   |
                (FENDCON.en & 1);
}

/*************************************************************************************
 * Function:              programFilter
 * Parameters:            void
 * Return:                void
 * Description:           The filter is programmed by transmitting the 64 programming bits to
 *                        the serial-to-parallel register of the filter via SPI.  Once all 64
 *                        bits are transmitted, the latch line is raised, a clock pulse is
 *                        generated and the latch line is lowered storing the bits.
 *************************************************************************************/
void programFilter(void)
{
   updateFilterData();

   setPinLow(FILTER_CLK);
   setPinLow(FILTER_DATA);
   setPinLow(FILTER_LATCH);
   enableSPI2();
   writeSPI2(filterData[0]);
   writeSPI2(filterData[1]);
   writeSPI2(filterData[2]);
   writeSPI2(filterData[3]);
   disableSPI2();

   setFilterLatchAsOutput();
   setFilterDataAsOutput();
   setFilterClkAsOutput();
   setPinLow(FILTER_LATCH);
   setPinLow(FILTER_CLK);
   prgmDelay();
   setPinHigh(FILTER_LATCH);
   prgmDelay();
   setPinHigh(FILTER_CLK);
   prgmDelay();
   setPinLow(FILTER_DATA);
   setPinLow(FILTER_LATCH);
   prgmDelay();
   setPinLow(FILTER_CLK);
   prgmDelay();
}

/*************************************************************************************
 * Function:              prgmDelay
 * Parameters:        void
 * Return:            void
 * Description:       A simplistic delay used to ensure timing requirements are met during
 *                    the filter programming process.
 ****************************************************************************************/
void prgmDelay(void)
{
   Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop();
   Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop(); Nop();
}

/****************************************************************************************
 * Function:          updateAnalogTuning
 * Parameters:        void
 * Return:            void
 * Description:       Programs the DACS with the appropriate current Analog F-Tune
 *                    and Q-Tune values.
 ****************************************************************************************/
void updateAnalogTuning(void)
{
   programFrontEndDAC(loadInputRegB(getFrontEndAnalogFTune()));
   programFrontEndDAC(loadInputRegA(getFrontEndAnalogQTune()));
   programFrontEndDAC(loadDACRegsABUpdateOutputsAB());
   programBackEndDAC(loadInputRegB(getBackEndAnalogQTune()));
   programBackEndDAC(loadInputRegA(getBackEndAnalogFTune()));
   programBackEndDAC(loadDACRegsABUpdateOutputsAB());
}

/****************************************************************************************
 * Function:          algorithm
 * Parameters:        void
 * Return:            void
 * Description:       Implementation of the Two-Pole Tuning Algorithm
 *                    using only Digital Controls.
 ****************************************************************************************/
void algorithm(void)
{
   turnRFSwitchOff();
   enableFrontEnd();    // Enable Front-End
   disableFrontEndAD(); // Disble Front-End Amplitude Detector
   disableFrontEndFD(); // Disable Front-End Frequency Divider
   enableBackEnd();     // Enable Back-End
   disableBackEndAD();  // Disable Back-End Amplitude Detector
   disableBackEndFD();  // Disable Back-End Frequency Divider

   coarseFrontEndFTune();
   coarseBackEndFTune();

   setCouplingUpper(CouplingUpper);
   setCouplingLower(CouplingLower);
   setCouplingUFLB(CouplingUFLB);
   setCouplingLFUB(CouplingLFUB);
   programFilter();

   turnRFSwitchOn();    // Algorithm is Complete, so Turn on RF Switch

   if ( _PrintFilterSettings )
   {
      printFilterOptions();
      _PrintFilterSettings = FALSE;
   }
}

/****************************************************************************************
 * Function:          coarseFrontEndFTune
 * Parameters:        void
 * Return:            void
 * Description:       Tunes the Front-End to the desired center frequency within a frequency
 *                    tolerance.  Once within the frequency tolerance the frequency controls
 *                    are adjusted and a comparison is done on which setting brought the pole
 *                          closer to the desired center frequency.
 * Modified:                This course tune function now calls the fine tune function and sets and
 *                          initializes additional variables for fine tuning.
 *********************************************************************************/
void coarseFrontEndFTune(void)
{
   int itt = 0;
   int prevFrontEndFCnt = 0;
   int prevFrontEndFCntDiff = 0, curFrontEndFCntDiff = 0;

   // Set frequency setting to controls mid-point
   setFrontEndDigitalFTune((FTUNE_DIG_MIN + FTUNE_DIG_MAX) >> 1);
              setFrontEndAnalogFTune((ANALOG_MIN + ANALOG_MAX) >> 1);
              updateAnalogTuning();
   FrontEndFCnt = getFrontEndFrequency();

   while ( (FrontEndFCnt < (CenterFreq - FreqTol)) || (FrontEndFCnt > (CenterFreq + FreqTol)) )
   {
      if ( FrontEndFCnt < (CenterFreq - FreqTol) )
                                          decFrontEndDigitalFTune();
                          else if ( FrontEndFCnt > (CenterFreq + FreqTol) )
                                          incFrontEndDigitalFTune();

                          FrontEndFCnt = getFrontEndFrequency();
                          if ( ++itt > MAX_FREQTUNE_ITTS )
      {
         txStrUART1("Max Front-End Frequency Tune Itterations Exceeded...\r\n");
         return;
      }
   }
   // Frequency within Tolerance now find Closest Setting
   prevFrontEndFCnt = FrontEndFCnt;
   prevFrontEndFCntDiff = absDiff(CenterFreq,prevFrontEndFCnt);
   if ( FrontEndFCnt < CenterFreq )
   {
      decFrontEndDigitalFTune();
      FrontEndFCnt = getFrontEndFrequency();
      curFrontEndFCntDiff = absDiff(CenterFreq,FrontEndFCnt);
      if ( curFrontEndFCntDiff > prevFrontEndFCntDiff )
      {
         incFrontEndDigitalFTune();
         programFilter();
      }
   }
   else if ( FrontEndFCnt > CenterFreq )
   {
      incFrontEndDigitalFTune();
      FrontEndFCnt = getFrontEndFrequency();
      curFrontEndFCntDiff = absDiff(CenterFreq,FrontEndFCnt);
      if ( curFrontEndFCntDiff > prevFrontEndFCntDiff )
      {
         decFrontEndDigitalFTune();
         programFilter();
      }
   }

              FrontEndFCnt = getFrontEndFrequency();
              fineFrontEndFTune();
              FrontEndFCnt = getFrontEndFrequency();

   #if _DEBUG_ALGORITHM_ == 1
      txStrUART1("----Freq Tune Essentially Done---\r\n");
      printFrontEndDigitalFTune();
      printFrontEndDigitalQTune();
   #endif

   // Back-Off Q-Enhancement by Set Amount
   setFrontEndDigitalQTune(getFrontEndDigitalQTune() - FrontEndQBackOff);
   #if _DEBUG_ALGORITHM_ == 1
      txStrUART1("----Do Q Back-Off for BW---\r\n");
      printFrontEndDigitalQTune();
   #endif

   // Ensure Filter is Not Oscillating After Q-Enhancement Back-Off (Insufficient Back-Off)
   enableFrontEndAD();
   programFilter();
   FrontEndAD = readFrontEndAD();
   while ( FrontEndAD < (FrontEndNonOsc - FrontEndADThresh1) )
   {
      #if _DEBUG_ALGORITHM_ == 1
         txStrUART1("Still Oscillating after Back-Off...\r\n");
         printFrontEndDigitalQTune();
         printFrontEndAD();
      #endif

      decFrontEndDigitalQTune();
      programFilter();
      FrontEndAD = readFrontEndAD();

      #if _DEBUG_ALGORITHM_ == 1
         printFrontEndDigitalQTune();
         printFrontEndAD();
      #endif
   }
   disableFrontEndAD();

          // Counter the Frequency Shift Caused by Q-Enhancement Back-Off by Increasing Digital F-Tuning
   setFrontEndDigitalFTune(getFrontEndDigitalFTune() + FrontEndFOffset);
   programFilter();
}

/*****************************************************************************************
 * Function:            fineFrontEndFTune
 * Parameters:          void
 * Return:              void
 * Description:         Implements a fine tuning algorithm to linearly find the closest
 *                      achievable frequency using analog tuning on the front end
 *****************************************************************************************/
void fineFrontEndFTune(void)
{
   int itt = 0;
          int count = 0;
          FrontEndFCnt = getFrontEndFrequency();
   int prevFrontEndFCnt = FrontEndFCnt;
          if(FrontEndFCnt > CenterFreq)
          {
                    // Need to set at mid point for this fine tuning approach to work
                    // setFrontEndAnalogFTune(ANALOG_MIN);
                    // updateAnalogTuning();                    //change to update only back or front?
                    while(FrontEndFCnt != CenterFreq)
                    {
                              setFrontEndAnalogFTune(getFrontEndAnalogFTune()+1);
                              updateAnalogTuning();
                              FrontEndFCnt = getFrontEndFrequency();
          if (++itt > MAX_FINEFREQTUNE_ITTS)
          {
             txStrUART1("Max Front-End Frequency Fine Tune Itterations Exceeded...\r\n");
             return;
          }
                    }
                    prevFrontEndFCnt = FrontEndFCnt;
                    while(FrontEndFCnt == CenterFreq)
                    {
                              count++;
                              setFrontEndAnalogFTune(getFrontEndAnalogFTune()+1);
                              updateAnalogTuning();
                              FrontEndFCnt = getFrontEndFrequency();
                    }
                    setFrontEndAnalogFTune(prevFrontEndFCnt+count/2);
          }
```

```
		else if(FrontEndFCnt < CenterFreq)
		{
			// Need to set at mid point for this fine tunning approach to work
			// setFrontEndAnalogFTune(ANALOG_MIN);
			// updateAnalogTuning();			//change to update only back or front?
			while(FrontEndFCnt != CenterFreq)
			{
				setFrontEndAnalogFTune(getFrontEndAnalogFTune()-1);
				updateAnalogTuning();
				FrontEndFCnt = getFrontEndFrequency();
			}
			prevFrontEndFCnt = FrontEndFCnt;
			while(FrontEndFCnt == CenterFreq)
			{
				count++;
				setFrontEndAnalogFTune(getFrontEndAnalogFTune()-1);
				updateAnalogTuning();
				FrontEndFCnt = getFrontEndFrequency();
			}
			setFrontEndAnalogFTune(prevFrontEndFCnt-count/2);
			FrontEndFCnt = getFrontEndFrequency();
		}
		else if(FrontEndFCnt == CenterFreq)
		{
			while(FrontEndFCnt == CenterFreq)
			{
				setFrontEndAnalogFTune(getFrontEndAnalogFTune()+2);
				updateAnalogTuning();
				FrontEndFCnt = getFrontEndFrequency();
			}
			setFrontEndAnalogFTune(getFrontEndAnalogFTune()-4);
			while(FrontEndFCnt == CenterFreq)
			{
				count++;
				setFrontEndAnalogFTune(getFrontEndAnalogFTune()-1);
				updateAnalogTuning();
				FrontEndFCnt = getFrontEndFrequency();
			}
			setFrontEndAnalogFTune(prevFrontEndFCnt-count/2);
		}
}

/*************************************************************************************
 * Function:		fineFrontEndQTune
 * Parameters:		void
 * Return:		void
 * Description:		Uses analog tuning to raise Q tuning as high as possible without
 *			oscillation
 *************************************************************************************/
void fineFrontEndQTune(void)
{
		setFrontEndAnalogQTune(ANALOG_MIN);
   while ( FrontEndAD >= (FrontEndNonOsc-FrontEndADThresh1) )
   {
			setFrontEndAnalogQTune(getFrontEndAnalogQTune()+FrontEndADThresh2);
			updateAnalogTuning();
		FrontEndAD = readFrontEndAD();
			if (getFrontEndAnalogQTune()==ANALOG_MAX) break;
		}
}

/*************************************************************************************
 * Function:		coarseBackEndFTune
 * Parameters:		void
 * Return:		void
 * Description:		Tunes the Back-End to the desired center frequency within a frequency
 *			tolerance.  Once within the frequency tolerance the frequency controls
 *			are adjusted and a comparison is done on which setting brought the pole
 *			closer to the desired center frequency.
 * Modified:		This course tune function now calls the fine tune function and sets and
 *                              initializes the additional variables for fine tuning.
 *************************************************************************************/
void coarseBackEndFTune(void)
{
    int itt = 0;
    int prevBackEndFCnt = 0;
    int prevBackEndFCntDiff = 0, curBackEndFCntDiff = 0;

    // Set frequency setting to controls mid-point
    setBackEndDigitalFTune((FTUNE_DIG_MIN + FTUNE_DIG_MAX) >> 1);
                setBackEndAnalogFTune((ANALOG_MIN + ANALOG_MAX) >> 1);
                updateAnalogTuning();
                BackEndFCnt = getBackEndFrequency();

    while ( (BackEndFCnt < (CenterFreq - FreqTol)) || (BackEndFCnt > (CenterFreq + FreqTol)) )
    {
        if ( BackEndFCnt < (CenterFreq - FreqTol) )
            decBackEndDigitalFTune();
        else if ( BackEndFCnt > (CenterFreq + FreqTol) )
            incBackEndDigitalFTune();

        BackEndFCnt = getBackEndFrequency();
        if ( ++itt > MAX_FREQTUNE_ITTS )
        {
            txStrUART1("Max Back-End Frequency Tune Itterations Exceeded...\r\n");
            return;
        }
    }
    // Frequency within Tolerance now find Closest Setting
    prevBackEndFCnt = BackEndFCnt;
    prevBackEndFCntDiff = absDiff(CenterFreq,prevBackEndFCnt);
    if ( BackEndFCnt < CenterFreq )
    {
        decBackEndDigitalFTune();
        BackEndFCnt = getBackEndFrequency();
        curBackEndFCntDiff = absDiff(CenterFreq,BackEndFCnt);
        if ( curBackEndFCntDiff > prevBackEndFCntDiff )
        {
            incBackEndDigitalFTune();
            programFilter();
        }
    }
    else if ( BackEndFCnt > CenterFreq )
    {
        incBackEndDigitalFTune();
        BackEndFCnt = getBackEndFrequency();
        curBackEndFCntDiff = absDiff(CenterFreq,BackEndFCnt);
        if ( curBackEndFCntDiff > prevBackEndFCntDiff )
        {
            decBackEndDigitalFTune();
            programFilter();
        }
    }

                BackEndFCnt = getBackEndFrequency();
                fineBackEndFTune();
                BackEndFCnt = getBackEndFrequency();

    #if _DEBUG_ALGORITHM_ == 1
        txStrUART1("----Freq Tune Essentially Done---\r\n");
        printBackEndDigitalFTune();
        printBackEndDigitalQTune();
    #endif

    // Back-Off Q-Enhancement by Set Amount
    setBackEndDigitalQTune(getBackEndDigitalQTune() - BackEndQBackOff);
    #if _DEBUG_ALGORITHM_ == 1
        txStrUART1("----Do Q Back-Off for BW---\r\n");
        printBackEndDigitalQTune();
    #endif
    // Ensure Filter is Not Oscillating After Q-Enhancement Back-Off (Insufficient Back-Off)
    enableBackEndAD();
    programFilter();
    BackEndAD = readBackEndAD();
    while ( BackEndAD < (BackEndNonOsc - BackEndADThresh1) )
    {
        #if _DEBUG_ALGORITHM_ == 1
            txStrUART1("Still Oscillating after Back-Off...\r\n");
            printBackEndDigitalQTune();
            printBackEndAD();
        #endif

        decBackEndDigitalQTune();
        programFilter();
        BackEndAD = readBackEndAD();

        #if _DEBUG_ALGORITHM_ == 1
            printBackEndDigitalQTune();
            printBackEndAD();
        #endif
    }

    disableBackEndAD();

    // Counter the Frequency Shift Caused by Q-Enhancement Back-Off by Increasing Digital F-Tuning
    setBackEndDigitalFTune(getBackEndDigitalFTune() + BackEndFOffset);
    programFilter();
}

/*************************************************************************************
 * Function:            fineBackEndFTune
 * Parameters:          void
 * Return:              void
 * Description:         Implements a fine tuning algorithm to linearly find the closest
 *                      achievable frequency using analog tuning on the front end
 *************************************************************************************/
void fineBackEndFTune(void)
{
    int itt = 0;
                int count = 0;
                BackEndFCnt = getBackEndFrequency();
    int prevBackEndFCnt = BackEndFCnt;
                if(BackEndFCnt > CenterFreq)
                {
                            // Need to set at mid point for this fine tuning approach to work
                            // setBackEndAnalogFTune(ANALOG_MIN);
                            // updateAnalogTuning();                    //change to update only back or front?
                            while(BackEndFCnt != CenterFreq)
                            {
                                        setBackEndAnalogFTune(getBackEndAnalogFTune()+1);
                                        updateAnalogTuning();
                                        BackEndFCnt = getBackEndFrequency();
                if (++itt > MAX_FINEFREQTUNE_ITTS)
                {
                    txStrUART1("Max Back-End Frequency Fine Tune Itterations Exceeded...\r\n");
                    return;
                }
                            }
                            prevBackEndFCnt = BackEndFCnt;
                            while(BackEndFCnt == CenterFreq)
                            {
                                        count++;
                                        setBackEndAnalogFTune(getBackEndAnalogFTune()+1);
                                        updateAnalogTuning();
                                        BackEndFCnt = getBackEndFrequency();
                            }
                            setBackEndAnalogFTune(prevBackEndFCnt+count/2);
                }
                else if(BackEndFCnt < CenterFreq)
```

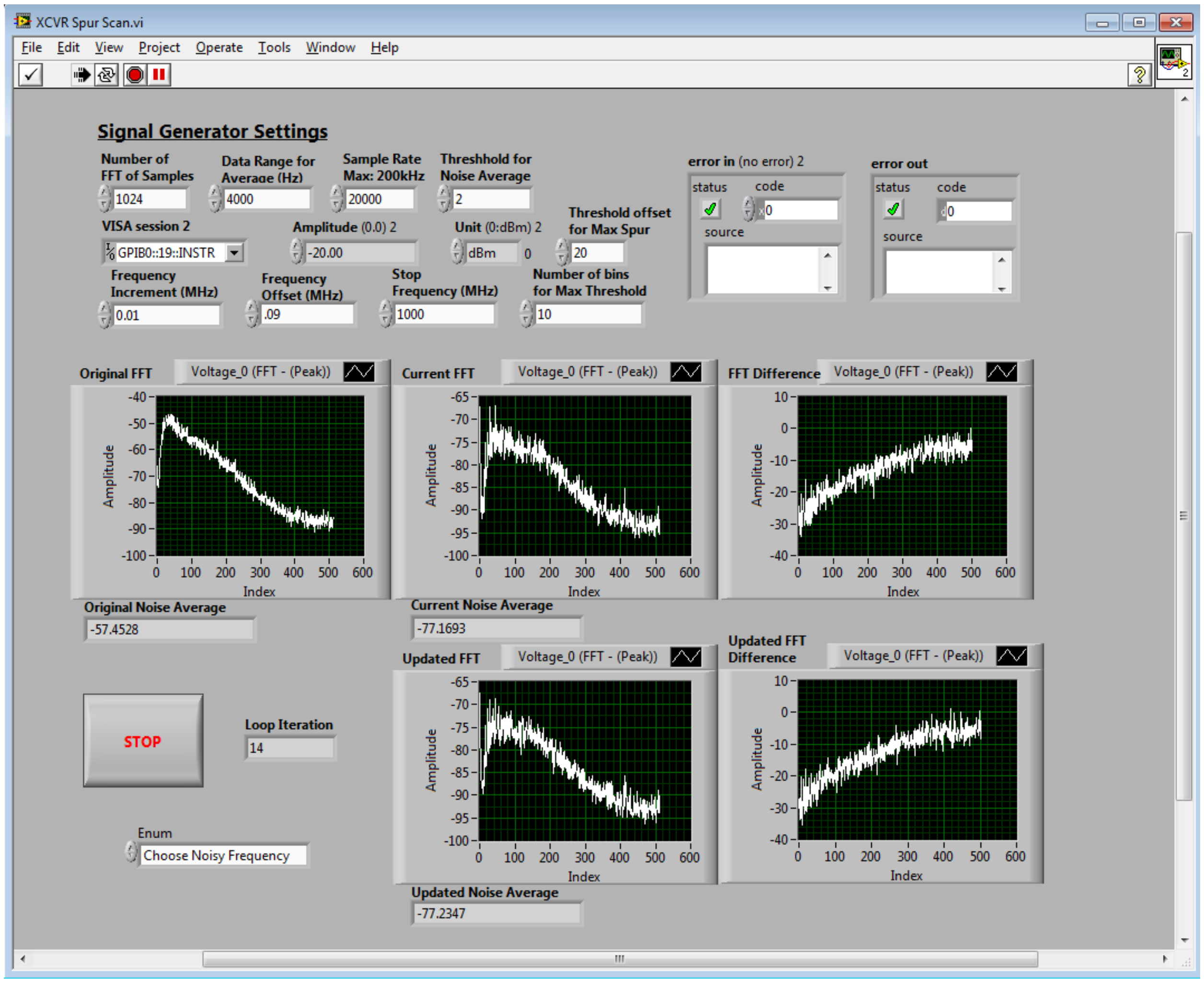
```
		{
			// Need to set at mid point for this fine tunning approach to work
			// setBackEndAnalogFTune(ANALOG_MIN);
			// updateAnalogTuning();				//change to update only back or Back?
			while(BackEndFCnt != CenterFreq)
			{
				setBackEndAnalogFTune(getBackEndAnalogFTune()-1);
				updateAnalogTuning();
				BackEndFCnt = getBackEndFrequency();
			}
			prevBackEndFCnt = BackEndFCnt;
			while(BackEndFCnt == CenterFreq)
			{
				count++;
				setBackEndAnalogFTune(getBackEndAnalogFTune()-1);
				updateAnalogTuning();
				BackEndFCnt = getBackEndFrequency();
			}
			setBackEndAnalogFTune(prevBackEndFCnt-count/2);
			BackEndFCnt = getBackEndFrequency();
		}
		else if(BackEndFCnt == CenterFreq)
		{
			while(BackEndFCnt == CenterFreq)
			{
				setBackEndAnalogFTune(getBackEndAnalogFTune()+2);
				updateAnalogTuning();
				BackEndFCnt = getBackEndFrequency();
			}
			setBackEndAnalogFTune(getBackEndAnalogFTune()-4);
			while(BackEndFCnt == CenterFreq)
			{
				count++;
				setBackEndAnalogFTune(getBackEndAnalogFTune()-1);
				updateAnalogTuning();
				BackEndFCnt = getBackEndFrequency();
			}
			setBackEndAnalogFTune(prevBackEndFCnt-count/2);
		}
}

/**************************************************************************************
 * Function:		fineBackEndQTune
 * Parameters:		void
 * Return:		void
 * Description:		Uses analog tuning to raise Q tuning as high as possible without
 *			oscillation
 **************************************************************************************/

void fineBackEndQTune(void)
{
		setBackEndAnalogQTune(ANALOG_MIN);
   while ( BackEndAD >= (BackEndNonOsc - BackEndADThresh1) )
   {
			setBackEndAnalogQTune(getBackEndAnalogQTune()+BackEndADThresh2);
			updateAnalogTuning();

			BackEndAD = readBackEndAD();
			if (getBackEndAnalogQTune()==ANALOG_MAX) break;
		}
}

/**************************************************************************************
 * Function:		getFrontEndFrequency
 * Parameters:		void
 * Return:		int - Current Front-End Frequency Count
 * Description:		Configures the filter so a reliable Front-End frequency count can
 *			be returned.
 **************************************************************************************/
int getFrontEndFrequency(void)
```

```
{
   int fCnt;
   PrevBackEndDigitalQTune = getBackEndDigitalQTune();

   disableBackEndAD();  // Disable Back-End Amplitude Detector
   disableBackEndFD();  // Disable Back-End Frequency Divider
   setBackEndDigitalQTune(QTUNE_DIG_MIN); // Degrade Back-End Q-Enhancement

   enableFrontEndFD();  // Enable Front-End Frequency Divider
   findFrontEndCriticalOsc();
   fCnt = readFrontEndFD();
   disableFrontEndFD(); // Disable Front-End Frequency Divider

   // Remove Excess Q-Enhancement Needed for Dependable Frequency Divider Reading
   setFrontEndDigitalQTune(getFrontEndDigitalQTune() - FrontEndQOffset);

   // Restore Previous Back-End Digital Q-Tune Value
   setBackEndDigitalQTune(PrevBackEndDigitalQTune);
   programFilter();     // Apply Filter Settings

   return fCnt;
}

/*************************************************************************************
 * Function:          getBackEndFrequency
 * Parameters:        void
 * Return:            int - Current Back-End Frequency Count
 * Description:       Configures the filter so a reliable Back-End frequency count can
 *                    be returned.
 *************************************************************************************/
int getBackEndFrequency(void)
{
   int fCnt;
   PrevFrontEndDigitalQTune = getFrontEndDigitalQTune();

   disableFrontEndAD(); // Disable Front-End Amplitude Detector
   disableFrontEndFD(); // Disable Front-End Frequency Divider
   setFrontEndDigitalQTune(QTUNE_DIG_MIN); // Degrade Front-End Q-Enhancement

   enableBackEndFD();   // Enable Back-End Frequency Divider

          findBackEndCriticalOsc();
   fCnt = readBackEndFD();
   disableBackEndFD();  // Disable Back-End Frequency Divider

   // Remove Excess Q-Enhancement Needed for Dependable Frequency Divider Reading
   setBackEndDigitalQTune(getBackEndDigitalQTune() - BackEndQOffset);
   // Restore Previous Front-End Digital Q-Tune Value
   setFrontEndDigitalQTune(PrevFrontEndDigitalQTune);
   programFilter();     // Apply Filter Settings

   return fCnt;
}

/*************************************************************************************
 * Function:          findFrontEndCriticalOsc
 * Parameters:        void
 * Return:            void
 * Description:       Sets Front-End Q-Enhancement to 0 and reads the Amplitude Detector
 *                    to determine the Non-Oscillation reading.   It then increases
 *                    Q-Enhancement until the Amplitude Detector reading drops below
 *                    the Non-Oscillation reading minus a set threshold value indicating that
 *                    the Front-End is oscillating.  To ensure that a valid Frequency Divider
 *                    reading is obtainable, the Q-Enhancement is increased by a set offset.
 * Modified:          This fuction now includes fine Q tuning
 *************************************************************************************/
void findFrontEndCriticalOsc(void)
{
   int prevFrontEndDigitalQTune = -1;
   FrontEndNonOsc = ANALOG_MIN;
   FrontEndAD = ANALOG_MAX;

              setFrontEndAnalogQTune(ANALOG_MIN);
              updateAnalogTuning();

   setFrontEndDigitalQTune(QTUNE_DIG_MIN); // Set Front-End Q-Enhancment to Minimum
   enableFrontEndAD();                     // Enable Front-End Amplitude Detector
   programFilter();                        // Apply Filter Settings
   FrontEndNonOsc = readFrontEndAD();      // Store Front-End Amplitude Detector Reading

   #if _DEBUG_CRITICALOSC_ == 1
      printFrontEndNonOsc();
      printFrontEndDigitalQTune();
   #endif

   // Increase Q-Enhancement Until Front-End is Oscillating
   while ( FrontEndAD >= (FrontEndNonOsc - FrontEndADThresh1) )
   {
      if ( prevFrontEndDigitalQTune == getFrontEndDigitalQTune() )
      {
         txStrUART1("---> Could Not Obtain Front-End Critical Oscillation <---\r\n");
         return;
      }
      prevFrontEndDigitalQTune = getFrontEndDigitalQTune();

      incFrontEndDigitalQTune();          // Increment Front-End Digital Q-Tune
      programFilter();                    // Apply Filter Settings
      FrontEndAD = readFrontEndAD();      // Store Front-End Amplitude Detector Reading

      #if _DEBUG_CRITICALOSC_ == 1
         printFrontEndAD();
         printFrontEndDigitalQTune();
      #endif


   }

              decFrontEndDigitalQTune();
              decFrontEndDigitalQTune();
              programFilter();                  // Apply Filter Settings
              prevFrontEndDigitalQTune = getFrontEndDigitalQTune();
              FrontEndAD = readFrontEndAD();      // Store Front-End Amplitude Detector Reading

              fineFrontEndQTune();
   disableFrontEndAD();                   // Disable Front-End Amplitude Detector

   // Ensure Oscillation for Dependable Frequency Divider Readings
   setFrontEndDigitalQTune(getFrontEndDigitalQTune() + FrontEndQOffset);
   programFilter();                       // Apply Filter Settings
}

/*************************************************************************************
 * Function:            findBackEndCriticalOsc
 * Parameters:          void
 * Return:               void
 * Description:         Sets Back-End Q-Enhancement to 0 and reads the Amplitude Detector
 *                      to determine the Non-Oscillation reading.   It then increases
 *                      Q-Enhancement until the Amplitude Detector reading drops below
 *                      the Non-Oscillation reading minus a set threshold value indicating that
 *                      the Back-End is oscillating.  To ensure that a valid Frequency Divider
 *                      reading is obtainable, the Q-Enhancement is increased by a set offset.
 * Modified:            This function now includes fine Q tuning
 *************************************************************************************/
void findBackEndCriticalOsc(void)
{
   int prevBackEndDigitalQTune = -1;
   BackEndNonOsc = ANALOG_MIN;
   BackEndAD = ANALOG_MAX;

              setBackEndAnalogQTune(ANALOG_MIN);
```

```
            updateAnalogTuning();

    setBackEndDigitalQTune(QTUNE_DIG_MIN);
    enableBackEndAD();
    programFilter();
    BackEndNonOsc = readBackEndAD();

    #if _DEBUG_CRITICALOSC_ == 1
        printBackEndNonOsc();
        printBackEndDigitalQTune();
    #endif

    while ( BackEndAD >= (BackEndNonOsc - BackEndADThresh1) )
    {
        if ( prevBackEndDigitalQTune == getBackEndDigitalQTune() )
        {
            txStrUART1("---> Could Not Obtain Back-End Critical Oscillation <---\r\n");
            return;
        }
        prevBackEndDigitalQTune = getBackEndDigitalQTune();
        incBackEndDigitalQTune();
        programFilter();
        BackEndAD = readBackEndAD();

        #if _DEBUG_CRITICALOSC_ == 1
            printBackEndAD();
            printBackEndDigitalQTune();
        #endif
    }

            decBackEndDigitalQTune();
            decBackEndDigitalQTune();
            programFilter();            // Apply Filter Settings
            prevBackEndDigitalQTune = getBackEndDigitalQTune();
            BackEndAD = readBackEndAD();     // Store Front-End Amplitude Detector Reading

            fineBackEndQTune();
    disableBackEndAD();

    // Ensure Oscillation for Dependable Frequency Divider Readings
    setBackEndDigitalQTune(getBackEndDigitalQTune() + BackEndQOffset);
    programFilter();

}
/* End of File */


/***************************************************************************************
 * Filename:          qefilter.h
 * Date:              June 2010
 * Compiler:          C30
 * Author:            Joel Schonberger
 * Company:           Kansas State University
 * Department:        Electrical & Computer Engineering
 * Research:          500 MHz Two-Pole Q-Enhanced Filter Tuning Algorithm
 * Discription:       This file houses the preprocessor definitions and function prototypes
 *                    needed by the QE Filter Tuning Algorithm.
 * -------------------------------------------------------------------------------------------------------------------------
 * Updated:           April 2012
 * Author:            Chelsi Kovala
 * Changes:           Modified to include function defintions:
                      void fineFrontEndFTune(void)
                      void fineBackEndFTune(void)
                      void fineFrontEndQTune(void)
                      void fineBackEndQTune(void)
                      printFrontEndAnalogQTune()
                      printFrontEndAnalogFTune()
                      printBackEndAnalogQTune()
                      printBackEndAnalogFTune()
                      Modified to include variables:
                              MAX_FINEFREQTUNE_ITTS
 *****************************************************************************************/
        #ifndef _QEFILTER_H
        #define _QEFILTER_H

        /* Preprocessor Definitions & Macros */
        #define FTUNE_DIG_MIN                 165 // Limit the Frequency Range for Reliable Frequency Divider Ouputs
        #define FTUNE_DIG_MAX                 255
        #define QTUNE_DIG_MIN                 0
        #define QTUNE_DIG_MAX                 63
        #define COUPLING_MIN                  0
        #define COUPLING_MAX                  31
        #define ANALOG_MIN                    0
        #define ANALOG_MAX                    1023        // Joel chose this value
        #define MAX_FREQTUNE_ITTS            100
        #define MAX_CRITOSC_ITTS             64
        #define MAX_FINEFREQTUNE_ITTS         1023/2      // = (1023/2)/inc or dec amount fine tuning is using - inc or dec


        #define isFrontEndEnabled()           (!FENDCON.en ? 1 : 0)
        #define enableFrontEnd()              FENDCON.en    = 0 // Active-Low Enable
        #define disableFrontEnd()             FENDCON.en    = 1
        #define isFrontEndADEnabled()         (!FENDCON.ADen ? 1 : 0)
        #define enableFrontEndAD()            FENDCON.ADen  = 0 // Active-Low Enable
        #define disableFrontEndAD()           FENDCON.ADen  = 1
        #define isFrontEndFDEnabled()         (!CAPCON1.FDFen ? 1 : 0)
        #define enableFrontEndFD()            CAPCON1.FDFen = 0 // Active-Low Enable
        #define disableFrontEndFD()           CAPCON1.FDFen = 1
        #define getFrontEndDigitalFTune()     FENDCON.fTune
        #define setFrontEndDigitalFTune(val)  FENDCON.fTune = ((val) > FTUNE_DIG_MAX ? FTUNE_DIG_MAX : ((val) <
FTUNE_DIG_MIN ? FTUNE_DIG_MIN : (val)))
        #define incFrontEndDigitalFTune()     setFrontEndDigitalFTune(FENDCON.fTune + 1)
        #define decFrontEndDigitalFTune()     setFrontEndDigitalFTune(FENDCON.fTune - 1)
        #define getFrontEndDigitalQTune()     FENDCON.qTune
        #define setFrontEndDigitalQTune(val)  FENDCON.qTune = ((val) > QTUNE_DIG_MAX ? QTUNE_DIG_MAX : ((val) <
QTUNE_DIG_MIN ? QTUNE_DIG_MIN : (val)))
        #define incFrontEndDigitalQTune()     setFrontEndDigitalQTune(FENDCON.qTune + 1)
        #define decFrontEndDigitalQTune()     setFrontEndDigitalQTune(FENDCON.qTune - 1)
        #define isBackEndEnabled()            (!BENDCON.en ? 1 : 0)
        #define enableBackEnd()               BENDCON.en = 0 // Active-Low Enable
        #define disableBackEnd()              BENDCON.en = 1
        #define isBackEndADEnabled()          (!BENDCON.ADen ? 1 : 0)
        #define enableBackEndAD()             BENDCON.ADen = 0 // Active-Low Enable
        #define disableBackEndAD()            BENDCON.ADen = 1
        #define isBackEndFDEnabled()          (!CAPCON2.FDBen ? 1 : 0)
        #define enableBackEndFD()             CAPCON2.FDBen = 0 // Active-Low Enable
        #define disableBackEndFD()            CAPCON2.FDBen = 1
        #define getBackEndDigitalFTune()      BENDCON.fTune
        #define setBackEndDigitalFTune(val)   BENDCON.fTune = ((val) > FTUNE_DIG_MAX ? FTUNE_DIG_MAX : ((val) <
FTUNE_DIG_MIN ? FTUNE_DIG_MIN : (val)))
        #define incBackEndDigitalFTune()      setBackEndDigitalFTune(BENDCON.fTune + 1)
        #define decBackEndDigitalFTune()      setBackEndDigitalFTune(BENDCON.fTune - 1)
        #define getBackEndDigitalQTune()      BENDCON.qTune
        #define setBackEndDigitalQTune(val)   BENDCON.qTune = ((val) > QTUNE_DIG_MAX ? QTUNE_DIG_MAX : ((val) <
QTUNE_DIG_MIN ? QTUNE_DIG_MIN : (val)))
        #define incBackEndDigitalQTune()      setBackEndDigitalQTune(BENDCON.qTune + 1)
        #define decBackEndDigitalQTune()      setBackEndDigitalQTune(BENDCON.qTune - 1)
        #define getCouplingUpper()            CAPCON1.upper
        #define setCouplingUpper(val)         CAPCON1.upper = ((val) > COUPLING_MAX ? COUPLING_MAX : ((val) <
COUPLING_MIN ? COUPLING_MIN : (val)))
        #define getCouplingLower()            CAPCON1.lower
        #define setCouplingLower(val)         CAPCON1.lower = ((val) > COUPLING_MAX ? COUPLING_MAX : ((val) <
COUPLING_MIN ? COUPLING_MIN : (val)))
        #define getCouplingUFLB()             CAPCON2.UFLB
        #define setCouplingUFLB(val)          CAPCON2.UFLB  = ((val) > COUPLING_MAX ? COUPLING_MAX : ((val) <
COUPLING_MIN ? COUPLING_MIN : (val)))
        #define getCouplingLFUB()             CAPCON2.LFUB
        #define setCouplingLFUB(val)          CAPCON2.LFUB  = ((val) > COUPLING_MAX ? COUPLING_MAX : ((val) <
COUPLING_MIN ? COUPLING_MIN : (val)))
        #define getFrontEndAnalogFTune()      ANALOG.FANAF
	#define setFrontEndAnalogFTune(val)	ANALOG.FANAF = ((val) > ANALOG_MAX ? ANALOG_MAX : ((val) <
ANALOG_MIN ? ANALOG_MIN : (val)))
	#define getFrontEndAnalogQTune()	ANALOG.FANAQ
	#define setFrontEndAnalogQTune(val)	ANALOG.FANAQ = ((val) > ANALOG_MAX ? ANALOG_MAX : ((val) <
ANALOG_MIN ? ANALOG_MIN : (val)))
	#define getBackEndAnalogFTune()	ANALOG.BANAF
	#define setBackEndAnalogFTune(val)	ANALOG.BANAF = ((val) > ANALOG_MAX ? ANALOG_MAX : ((val) <
ANALOG_MIN ? ANALOG_MIN : (val)))
	#define getBackEndAnalogQTune()	ANALOG.BANAQ
	#define setBackEndAnalogQTune(val)	ANALOG.BANAQ = ((val) > ANALOG_MAX ? ANALOG_MAX : ((val) <
ANALOG_MIN ? ANALOG_MIN : (val)))
	#define turnRFSwitchOn()	setPinLow(RFSW);  DEBUG.RFOn = 0
	#define turnRFSwitchOff()	setPinHigh(RFSW); DEBUG.RFOn = 1
	#define isRFSwitchOn()	!DEBUG.RFOn
	#define printFrontEndStatus()	(isFrontEndEnabled() ? txStrUART1("Front-End Enabled\r\n") : txStrUART1("Front-
End Disabled\r\n"))
	#define printFrontEndADStatus()	(isFrontEndADEnabled() ? txStrUART1("Front-End AD Enabled\r\n") :
txStrUART1("Front-End AD Disabled\r\n"))
	#define printFrontEndFDStatus()	(isFrontEndFDEnabled() ? txStrUART1("Front-End FD Enabled\r\n") :
txStrUART1("Front-End FD Disabled\r\n"))
	#define printFrontEndAD()	strPopulate16Bit(strFrontEndAD,FrontEndAD,'#',4)
	#define printFrontEndNonOsc()	strPopulate16Bit(strFrontEndNonOsc,FrontEndNonOsc,'#',4)
	#define printFrontEndFCnt()	strPopulate16Bit(strFrontEndFCnt,FrontEndFCnt,'#',4)
	#define printFrontEndDigitalQTune()	strPopulate16Bit(strFrontEndDigitalQTune,getFrontEndDigitalQTune(),'#',2)
	#define printFrontEndAnalogQTune()	strPopulate16Bit(strFrontEndAnalogQTune,getFrontEndAnalogQTune(),'#',4)
	#define printFrontEndDigitalFTune()	strPopulate16Bit(strFrontEndDigitalFTune,getFrontEndDigitalFTune(),'#',3)
	#define printFrontEndAnalogFTune()	strPopulate16Bit(strFrontEndAnalogFTune,getFrontEndAnalogFTune(),'#',4)
	#define printBackEndStatus()	(isBackEndEnabled() ? txStrUART1("Back-End Enabled\r\n") : txStrUART1("Back-End
Disabled\r\n"))
	#define printBackEndADStatus()	(isBackEndADEnabled() ? txStrUART1("Back-End AD Enabled\r\n") :
txStrUART1("Back-End AD Disabled\r\n"))
	#define printBackEndFDStatus()	(isBackEndFDEnabled() ? txStrUART1("Back-End FD Enabled\r\n") :
txStrUART1("Back-End FD Disabled\r\n"))
	#define printBackEndAD()	strPopulate16Bit(strBackEndAD,BackEndAD,'#',4)
	#define printBackEndNonOsc()	strPopulate16Bit(strBackEndNonOsc,BackEndNonOsc,'#',4)
	#define printBackEndFCnt()	strPopulate16Bit(strBackEndFCnt,BackEndFCnt,'#',4)
	#define printBackEndDigitalQTune()	strPopulate16Bit(strBackEndDigitalQTune,getBackEndDigitalQTune(),'#',2)
	#define printBackEndAnalogQTune()	strPopulate16Bit(strBackEndAnalogQTune,getBackEndAnalogQTune(),'#',4)
	#define printBackEndDigitalFTune()	strPopulate16Bit(strBackEndDigitalFTune,getBackEndDigitalFTune(),'#',3)
	#define printBackEndAnalogFTune()	strPopulate16Bit(strBackEndAnalogFTune,getBackEndAnalogFTune(),'#',4)
	#define printCouplingUpper()	strPopulate16Bit(strCouplingUpper,CouplingUpper,'#',2)
	#define printCouplingLower()	strPopulate16Bit(strCouplingLower,CouplingLower,'#',2)
	#define printCouplingUFLB()	strPopulate16Bit(strCouplingUFLB,CouplingUFLB,'#',2)
	#define printCouplingLFUB()	strPopulate16Bit(strCouplingLFUB,CouplingLFUB,'#',2)
	#define printRFSwitchStatus()	(isRFSwitchOn() ? txStrUART1("RF Switch On\r\n") : txStrUART1("RF Switch
Off\r\n"))
	#define printCenterFreq()	strPopulate16Bit(strCenterFreq,CenterFreq,'#',4)
	#define printFreqTol()	strPopulate16Bit(strFreqTol,FreqTol,'#',3)
	#define printFrontEndADThresh1()	strPopulate16Bit(strFrontEndADThresh1,FrontEndADThresh1,'#',3)
	#define printFrontEndADThresh2()	strPopulate16Bit(strFrontEndADThresh2,FrontEndADThresh2,'#',3)
	#define printFrontEndQOffset()	strPopulate16Bit(strFrontEndQOffset,FrontEndQOffset,'#',2)
	#define printFrontEndQBackOff()	strPopulate16Bit(strFrontEndQBackOff,FrontEndQBackOff,'#',2)
	#define printFrontEndFOffset()	strPopulate16Bit(strFrontEndFOffset,FrontEndFOffset,'#',2)
	#define printBackEndADThresh1()	strPopulate16Bit(strBackEndADThresh1,BackEndADThresh1,'#',3)
	#define printBackEndADThresh2()	strPopulate16Bit(strBackEndADThresh2,BackEndADThresh2,'#',3)
	#define printBackEndQOffset()	strPopulate16Bit(strBackEndQOffset,BackEndQOffset,'#',2)
	#define printBackEndQBackOff()	strPopulate16Bit(strBackEndQBackOff,BackEndQBackOff,'#',2)
	#define printBackEndFOffset()	strPopulate16Bit(strBackEndFOffset,BackEndFOffset,'#',2)

	/* Function Prototypes */
	void initFilter();
	void printFilterOptions(void);
	void printAlgorithmOptions(void);
	void updateFilterData(void);
	void programFilter(void);
	void prgmDelay(void);
	void updateAnalogTuning(void);
	void algorithm(void);
	void coarseFrontEndFTune(void);
	void coarseBackEndFTune(void);
```

```
int getFrontEndFrequency(void);
int getBackEndFrequency(void);
void findFrontEndCriticalOsc(void);
void findBackEndCriticalOsc(void);
void fineFrontEndFTune(void);
void fineBackEndFTune(void);
void fineFrontEndQTune(void);
void fineBackEndQTune(void);
#endif
/* End of File */
```

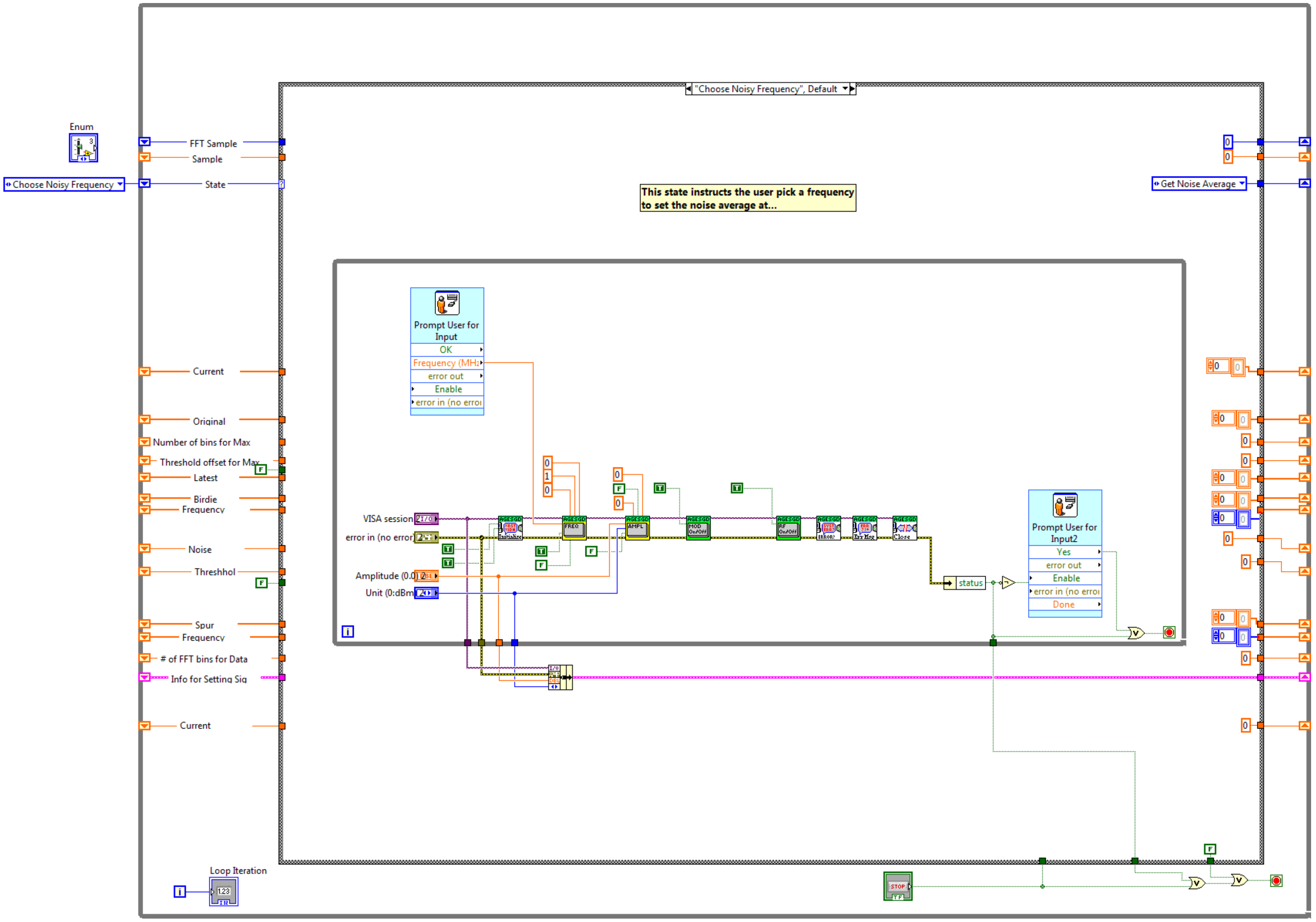
Appendix C - National Instruments LabVIEW Code



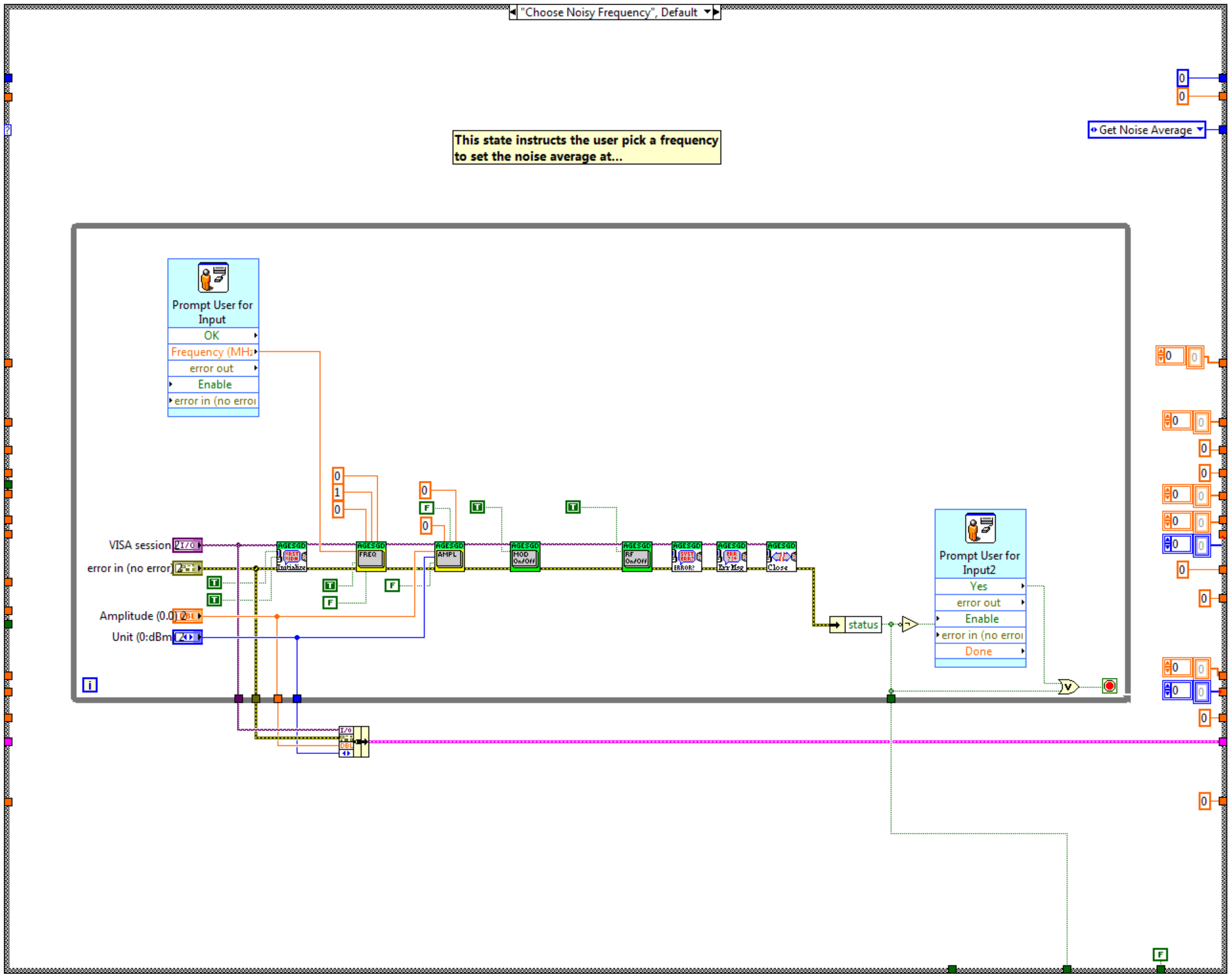
Signal Generator Settings

Number of FFT of Samples	1024
Data Range for Average (Hz)	4000
Sample Rate Max: 200kHz	20000
Threshhold for Noise Average	2
VISA session 2	GPIB0::19::INSTR
Amplitude (0.0) 2	-20.00
Unit (0:dBm) 2	dBm 0
Threshold offset for Max Spur	20
Frequency Increment (MHz)	0.01
Frequency Offset (MHz)	.09
Stop Frequency (MHz)	1000
Number of bins for Max Threshold	10
error in (no error) 2	status ✓ code 0 source
error out	status ✓ code 0 source
Original Noise Average	-57.4528
Current Noise Average	-77.1693
Updated Noise Average	-77.2347
Loop Iteration	14
Enum	Choose Noisy Frequency

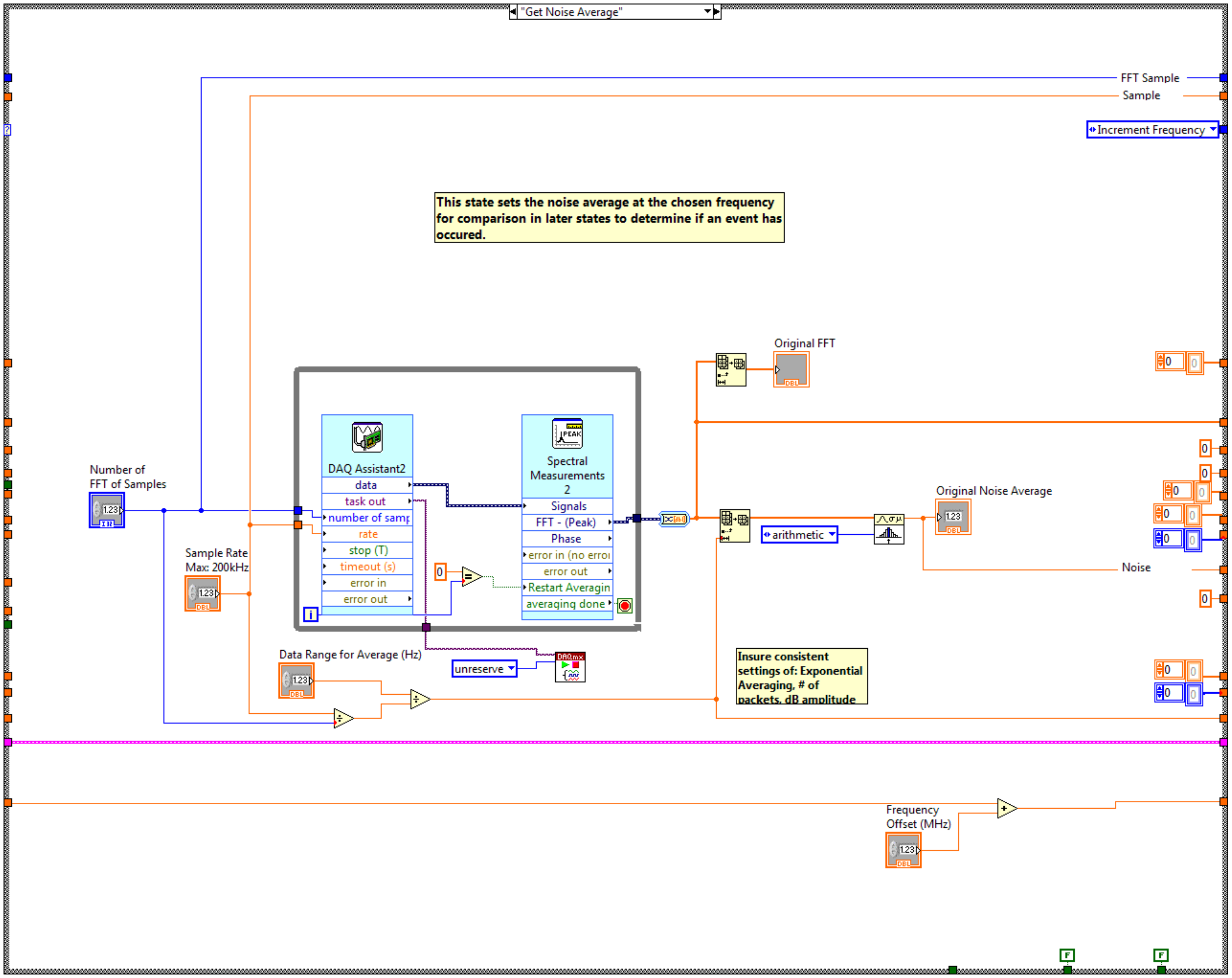
Figure C.1 – Screen Shot of NI Code ‘XCVR_Spur_Scan’ Running



Enum
Choose Noisy Frequency
FFT Sample
Sample
State
"Choose Noisy Frequency", Default
This state instructs the user pick a frequency to set the noise average at...
Get Noise Average
Prompt User for Input
OK
Frequency (MHz
error out
Enable
error in (no error
Current
Original
Number of bins for Max
Threshold offset for Max
Latest
Birdie Frequency
Noise
Threshhol
Spur Frequency
# of FFT bins for Data
Info for Setting Sig
Current
VISA session
error in (no error
Amplitude (0.0)
Unit (0:dBm
status
Prompt User for Input2
Yes
error out
Enable
error in (no error
Done
Loop Iteration

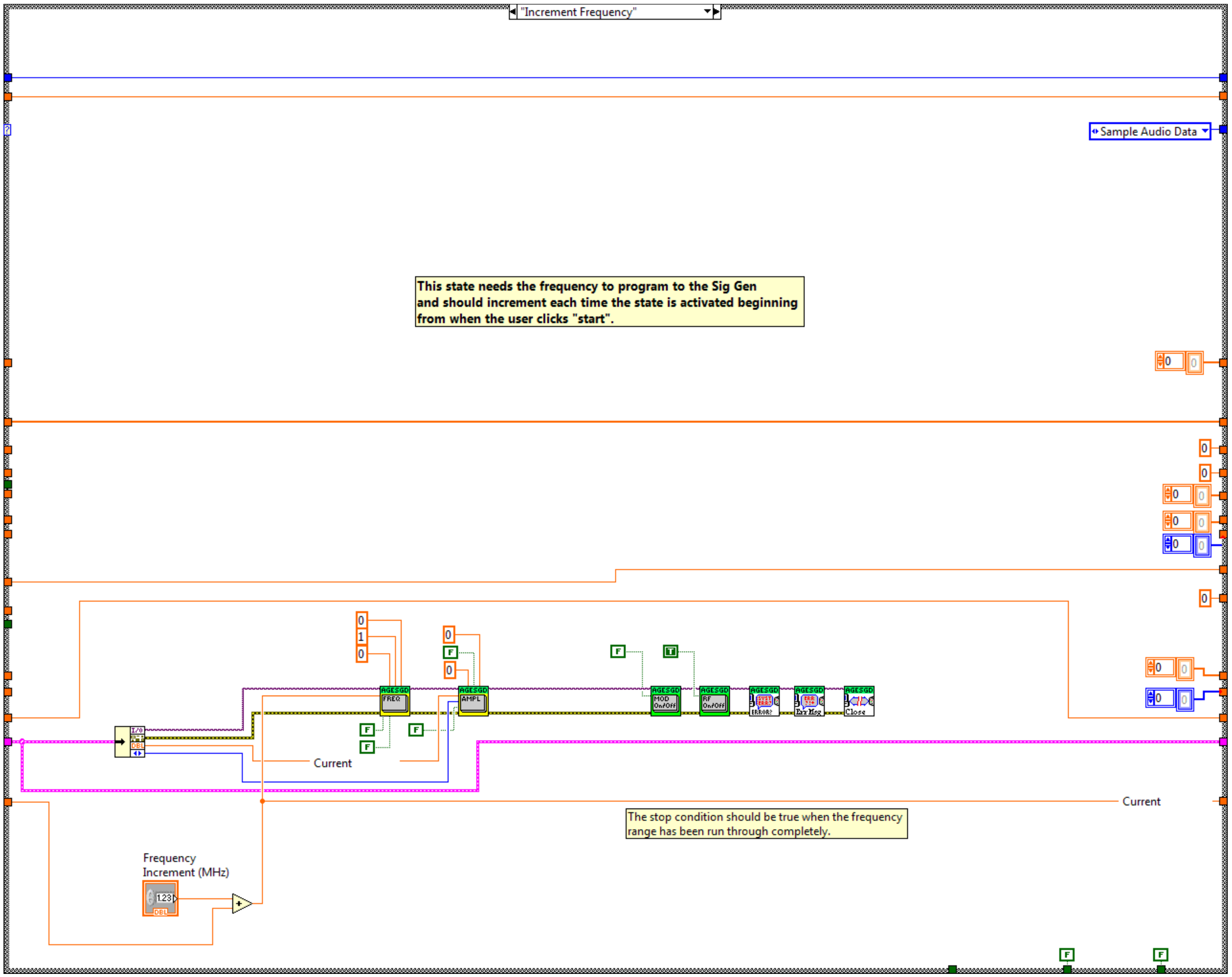


This state instructs the user pick a frequency to set the noise average at...



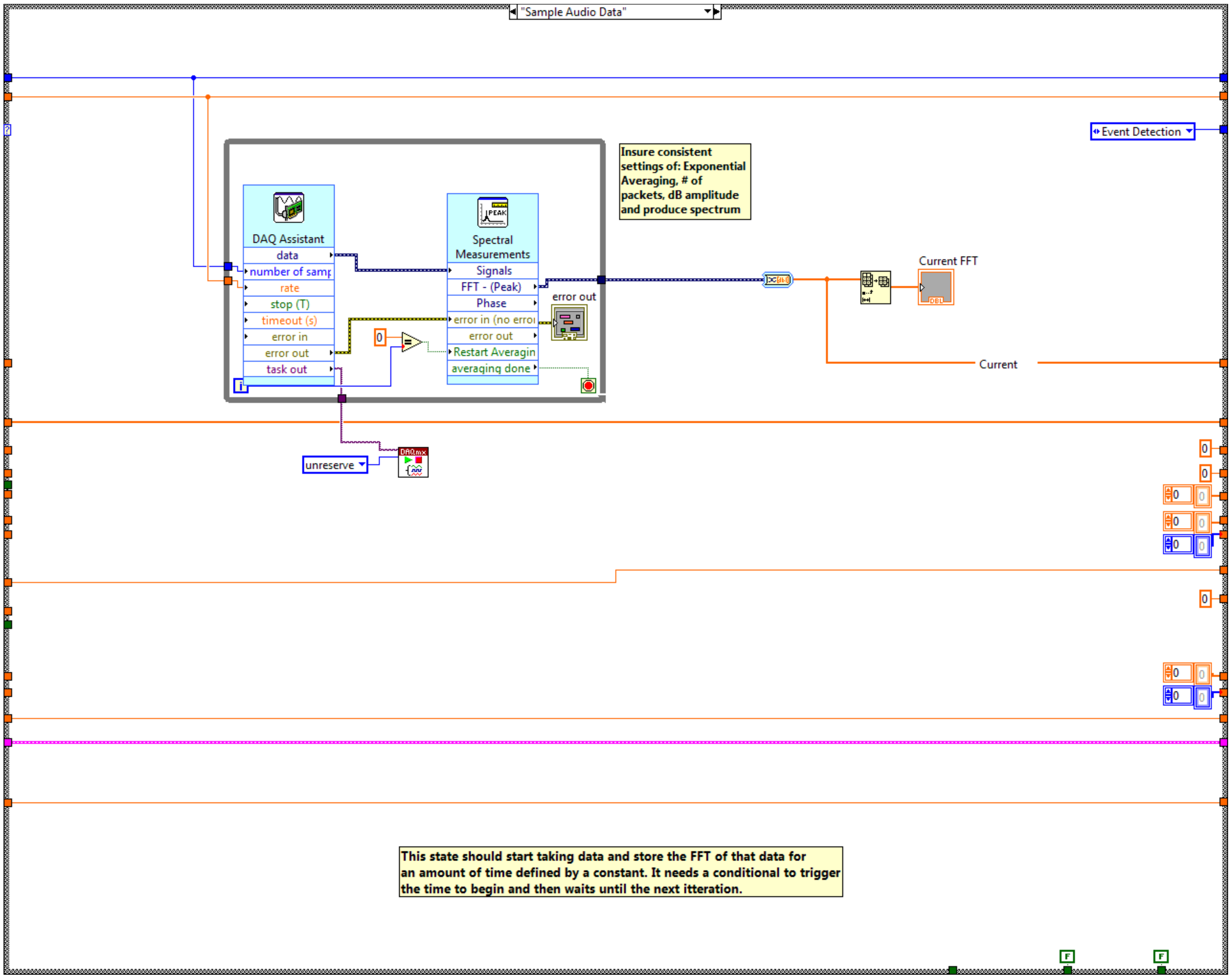
This state sets the noise average at the chosen frequency for comparison in later states to determine if an event has occured.

Insure consistent settings of: Exponential Averaging, # of packets, dB amplitude



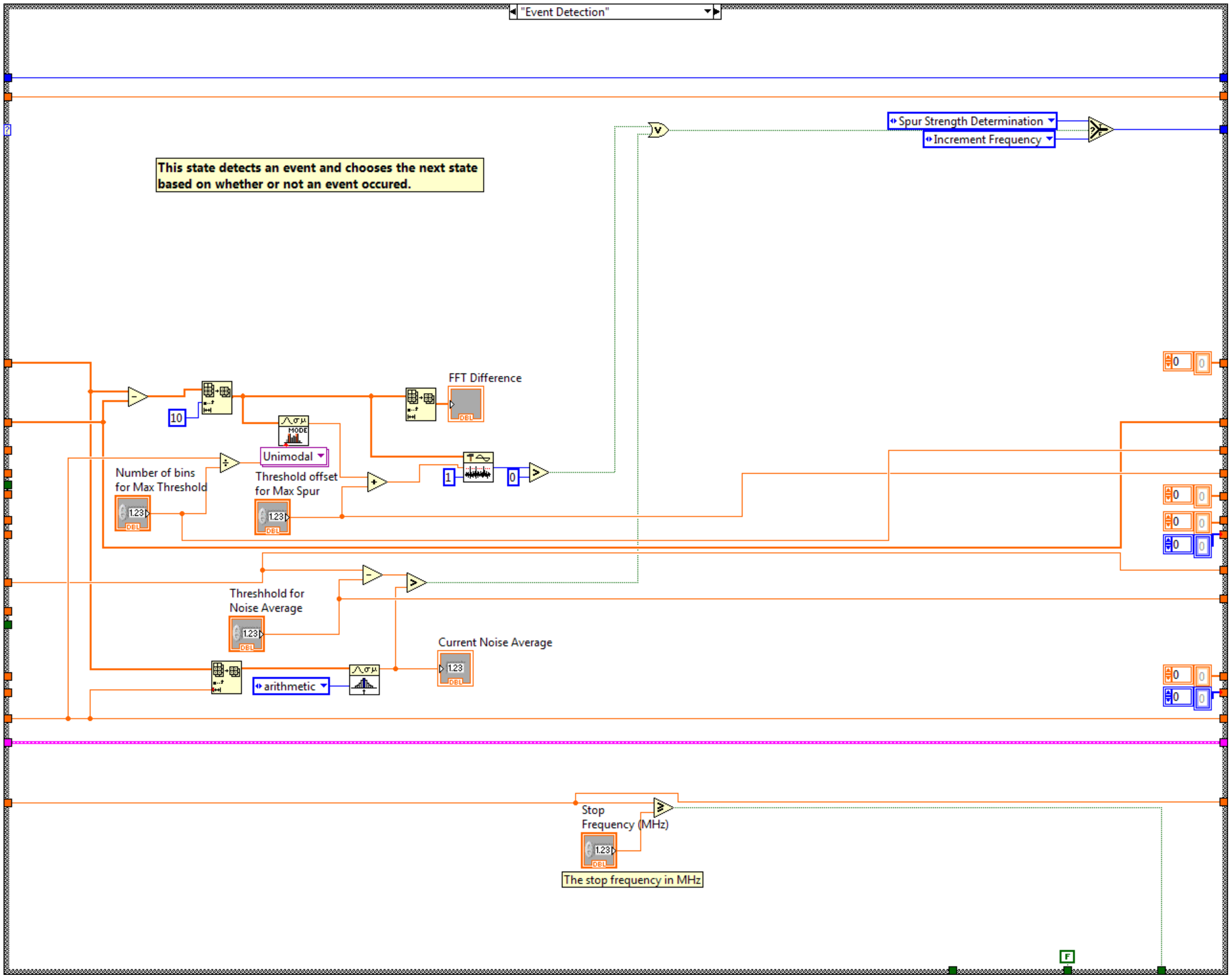
This state needs the frequency to program to the Sig Gen and should increment each time the state is activated beginning from when the user clicks "start".

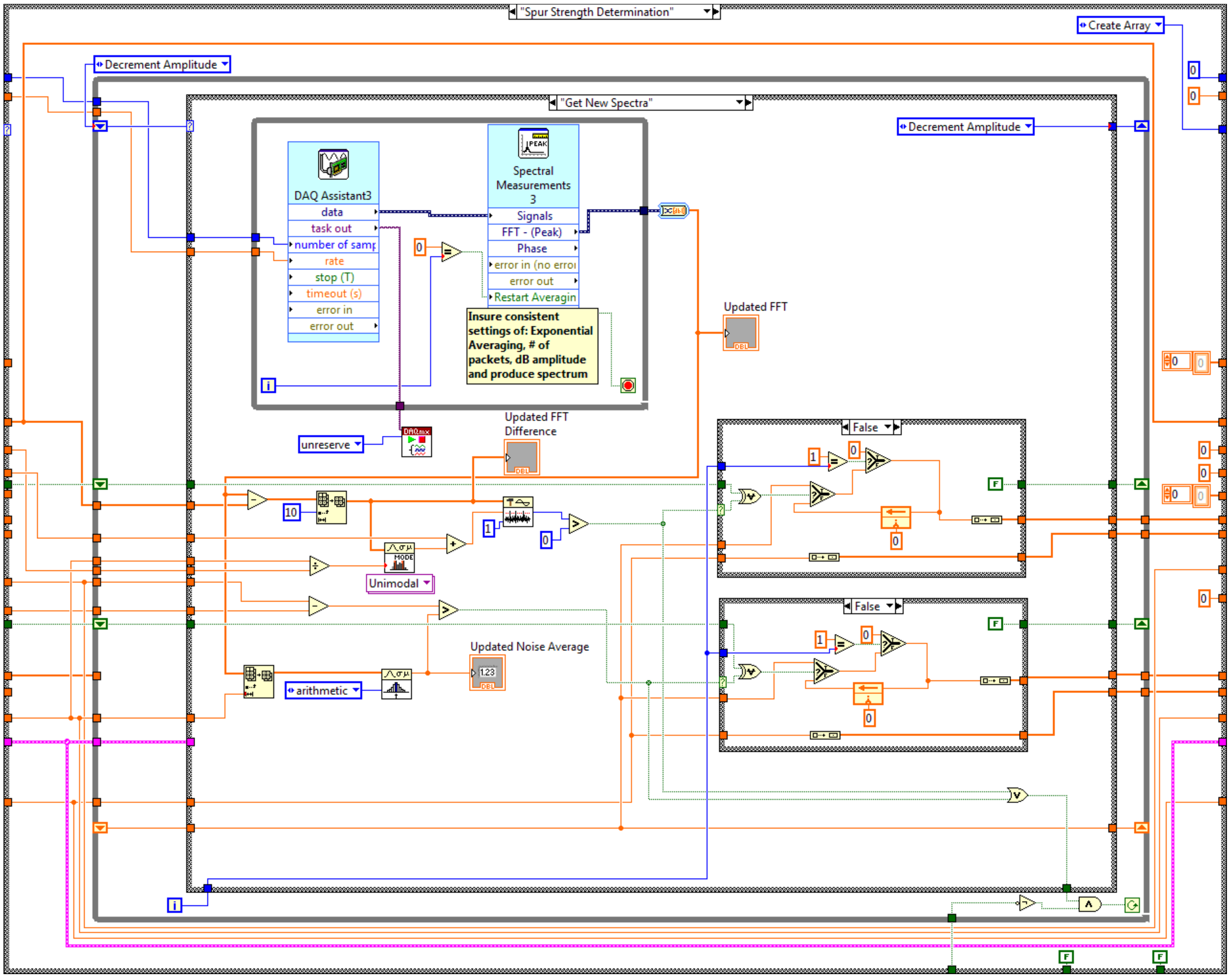
The stop condition should be true when the frequency range has been run through completely.



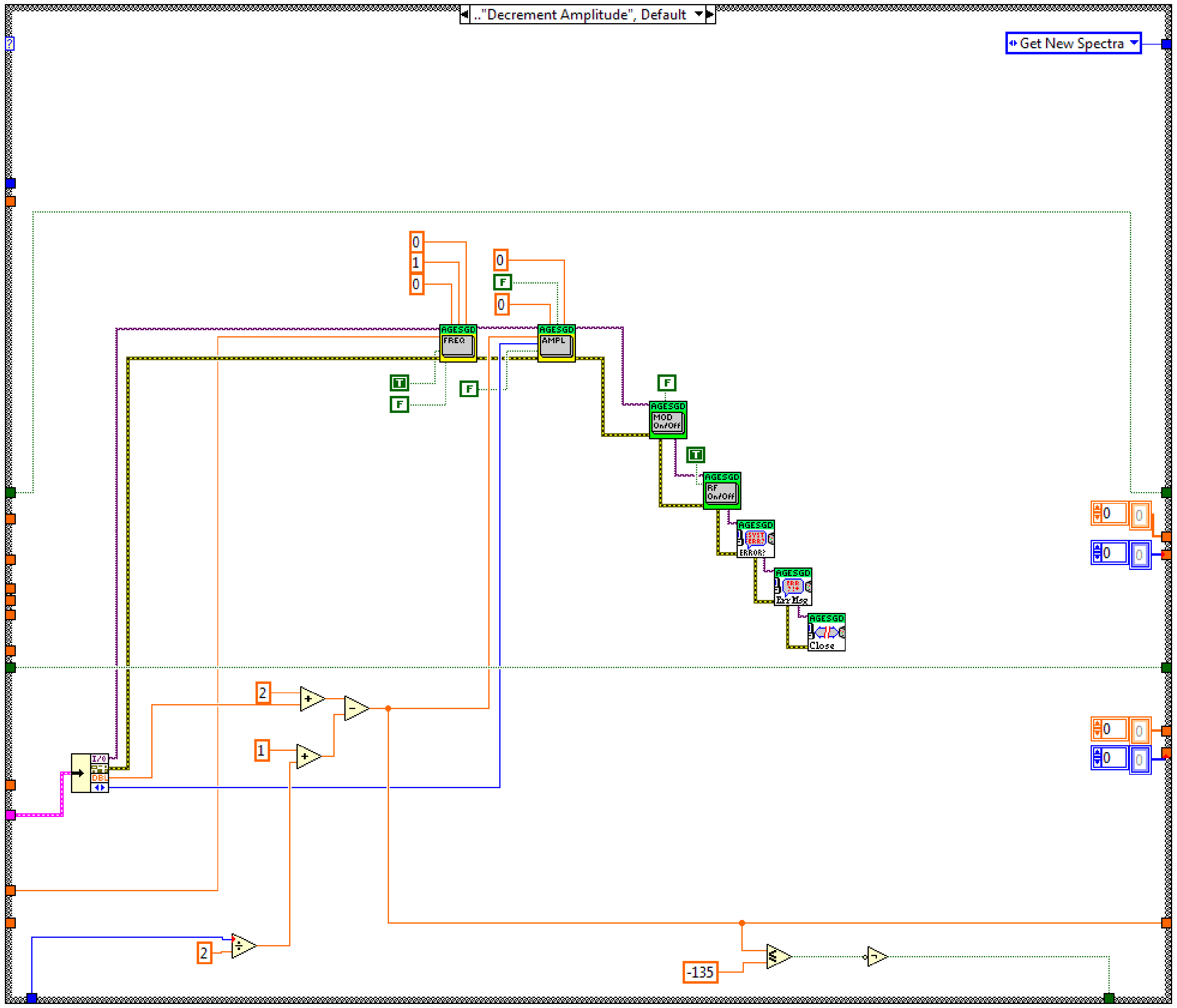
Insure consistent settings of: Exponential Averaging, # of packets, dB amplitude and produce spectrum

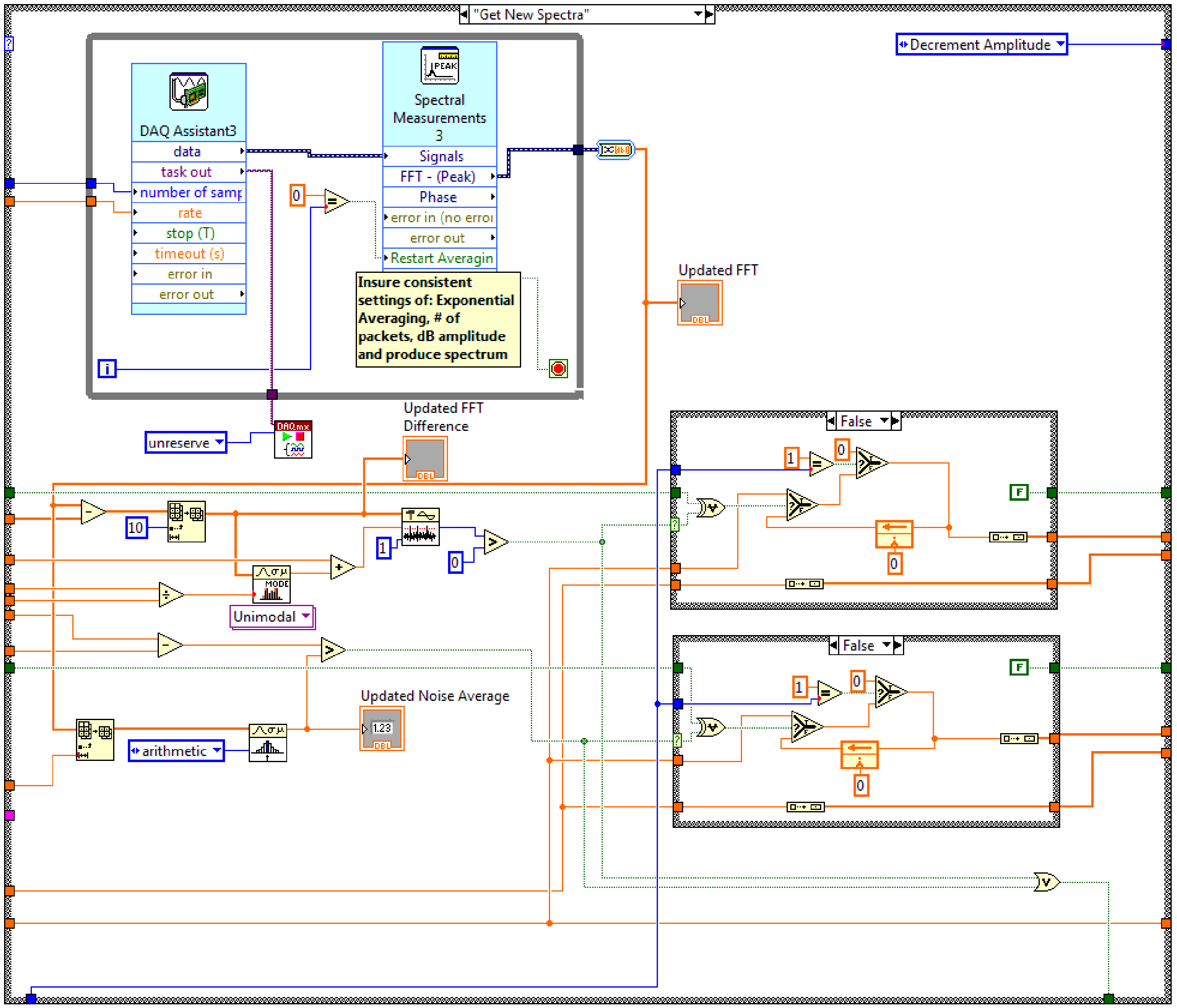
This state should start taking data and store the FFT of that data for an amount of time defined by a constant. It needs a conditional to trigger the time to begin and then waits until the next iteration.

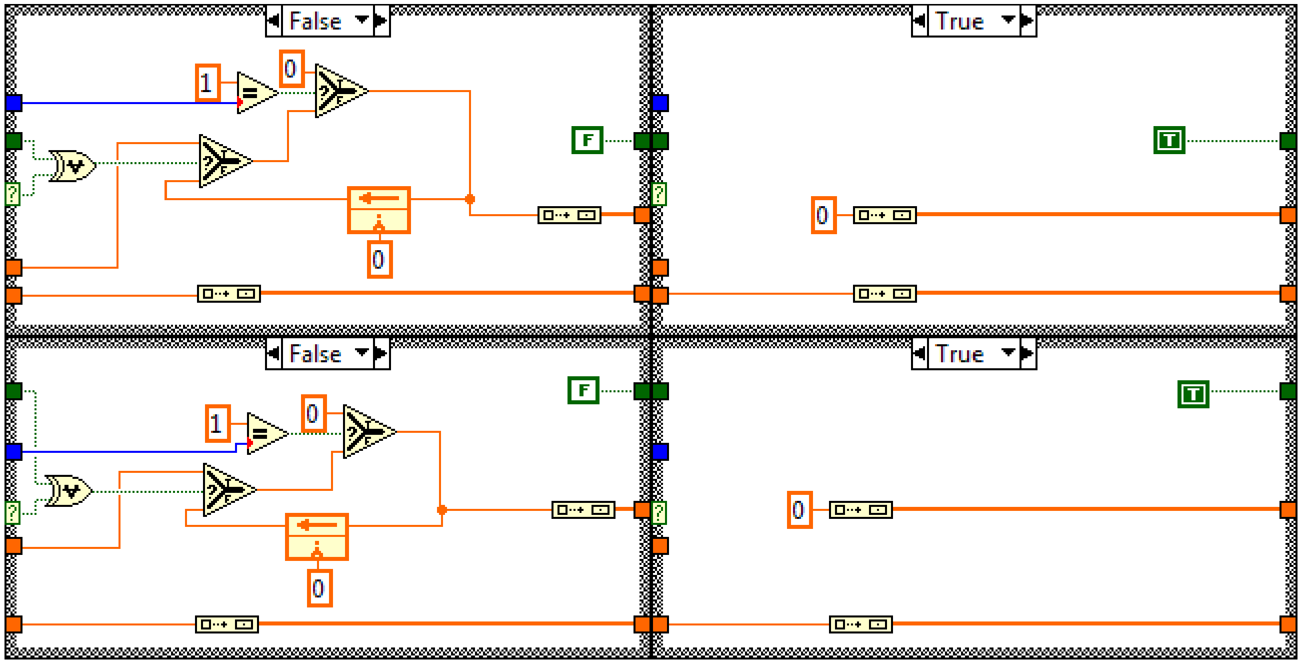

"Event Detection"
This state detects an event and chooses the next state based on whether or not an event occured.
Spur Strength Determination
Increment Frequency
FFT Difference
10
Unimodal
Number of bins for Max Threshold
Threshold offset for Max Spur
1
0
Threshhold for Noise Average
Current Noise Average
arithmetic
Stop Frequency (MHz)
The stop frequency in MHz
F

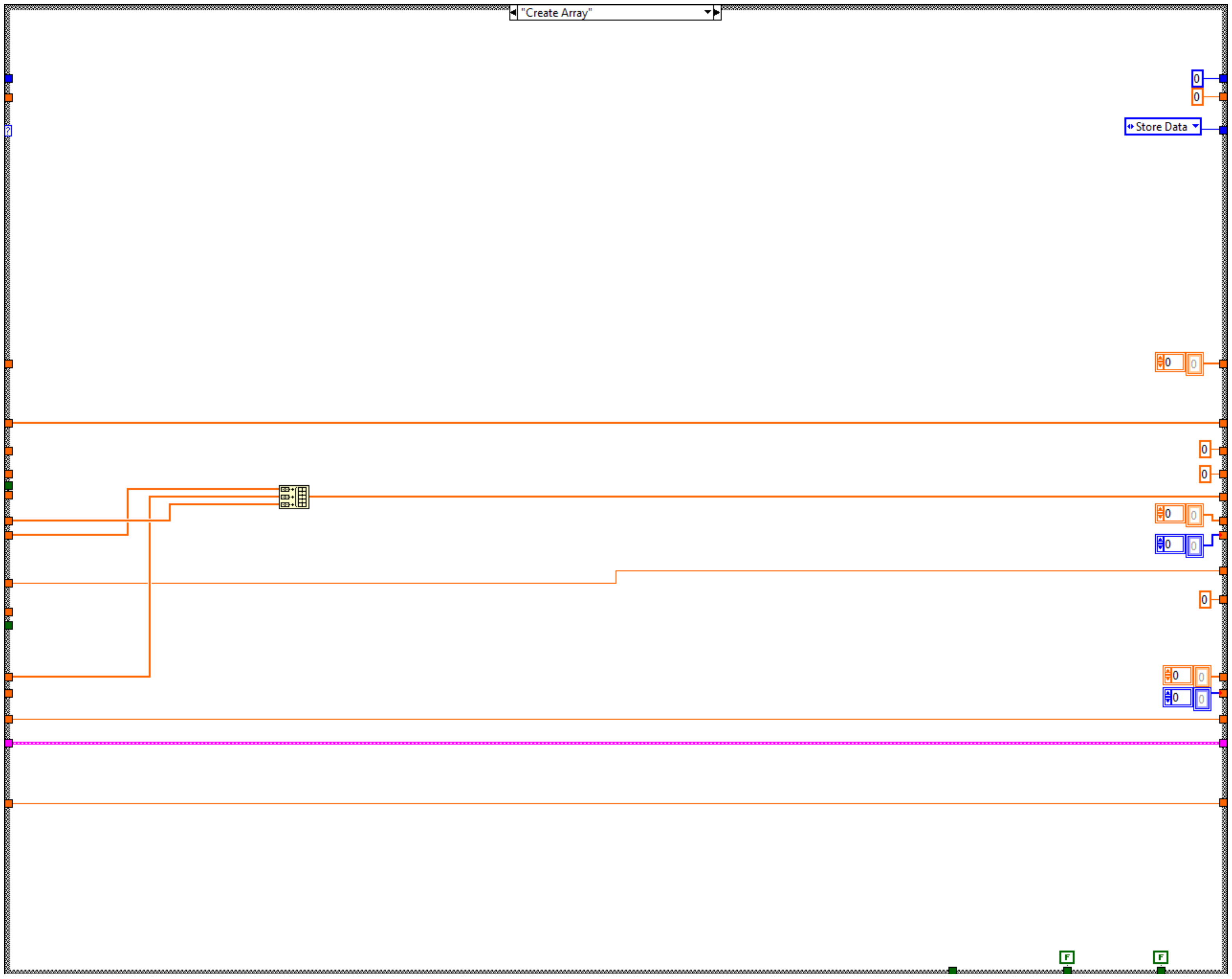


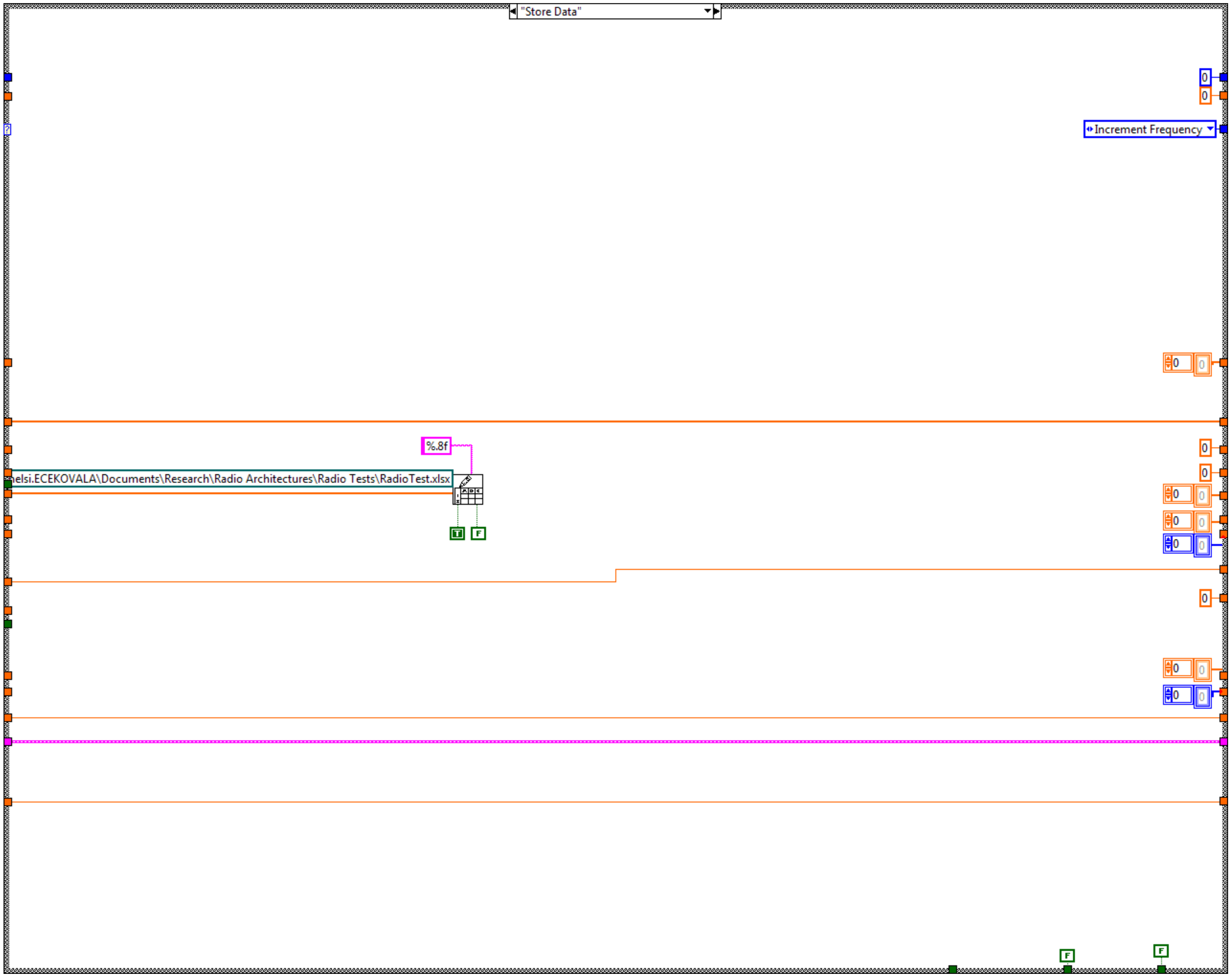
Insure consistent settings of: Exponential Averaging, # of packets, dB amplitude and produce spectrum


.."Decrement Amplitude", Default
Get New Spectra
AGESGD FREQ
AGESGD AMPL
AGESGD MOD On/Off
AGESGD RF On/Off
AGESGD ERROR?
AGESGD Err Msg
AGESGD Close
-135


"Get New Spectra"
Decrement Amplitude
DAQ Assistant3
data
task out
number of samp
rate
stop (T)
timeout (s)
error in
error out
Spectral Measurements 3
Signals
FFT - (Peak)
Phase
error in (no erro
error out
Restart Averagin
Insure consistent settings of: Exponential Averaging, # of packets, dB amplitude and produce spectrum
Updated FFT
Updated FFT Difference
unreserve
10
Unimodal
arithmetic
Updated Noise Average
False
False


False
1
0
F
0
True
T
0
False
1
0
F
0
True
T
0


"Create Array"
Store Data
0
F


"Store Data"
Increment Frequency
%.8f
elsi.ECEKOVALA\Documents\Research\Radio Architectures\Radio Tests\RadioTest.xlsx
T
F

Appendix D - Octave Code for Spurious Response Analysis

```
function spurs = findSpurs(s, minPower, var, maxm, maxn, IF, ftune, hilo)
% Created:                    March 2012
% Author:             Chelsi Kovala
% Description:        This function was written to analyze data recorded by the
%                     XCVR_Spur_Scan.vi written in NI LabVIEW. This function takes
%                     a set of frequencies which produced spurious responses and the
%                     the first IF of the receiver, and checks to see if mixer spurs
%                     could explain the spur by choosing all combinations of n and m
%                     and checking to see if the equation fIF=nfRF+-mfLO is satisfied.
%                     This all assumes a superheterodyne receiver.
% File Format:        File must be a text file with two columns of numbers,
%                     [frequency amplitude] with no headings to be read in correctly.
% Parameters:         s                text file to be read in
%                     minPower         events with power greater than this won't be considered
%                     var              how far the calculated value may differ from the
%                                      expected value in MHz, e.g. .1 = may differ by 100 kHz
%                     maxm             the maximum m coefficient to consider
%                     maxn             the maximum n coefficient to consider
%                     IF               the intermediate frequency to consider
%                     ftune            the frequency being received
%                     hilo             1 if highside injection is used, 0 if lowside injection
%                                      is used

file=fopen(s);
C = textscan(file,'%f %f');
a1=cell2mat(C(:,1));
a2=cell2mat(C(:,2));
NewA=[a1 a2];
[l dontcare]=size(NewA);
temp=1;
A=0;
for i=1:l
   if(NewA(i,2)<minPower)
      A(temp,1)=NewA(i,1);
      temp=temp+1;
   else
   end
end

[l1 dontcare]=size(A);
fIF1=IF(1);
if(hilo==1)
   flo1=fIF1+ftune;
else
   flo1=ftune-fIF1;
end
s1count=1;
s2count=1;
count=1;
spurDiff1st=zeros(1,3);
spurSum1st=zeros(1,3);
temp3=-1;
for i=1:l1
   for m=0:maxm
      for n=0:maxn
         temp1=(m*flo1-n*A(i,1));
         temp2=(n*A(i,1)+m*flo1);
         if ((abs(temp1)<fIF1+var) && (abs(temp1)>fIF1-var))
            if temp1<0
               spurDiff1st(s1count,:)=[A(i,1) -m n];
            else
               spurDiff1st(s1count,:)=[A(i,1) m -n];
            end
            s1count=s1count+1;
         end
         if ((temp2<fIF1+var) && (temp2>fIF1-var))
```

```
                spurSum1st(s2count,:)=[A(i,1) m n];
                s2count=s2count+1;
              else
                firstOrd(count,1)=abs(temp1);
                firstOrd(count,2)=temp2;
                count=count+1;
              end
            end
          end
        end
        spurDiff1st
spurSum1st
```