University of New Mexico
UNM Digital Repository

Electrical and Computer Engineering ETDs | Engineering ETDs

2-8-2011

Agile Testing Methods for IC Radios

Luis Martinez

Follow this and additional works at: https://digitalrepository.unm.edu/ece_etds

Recommended Citation
Martinez, Luis. "Agile Testing Methods for IC Radios." (2011). https://digitalrepository.unm.edu/ece_etds/173

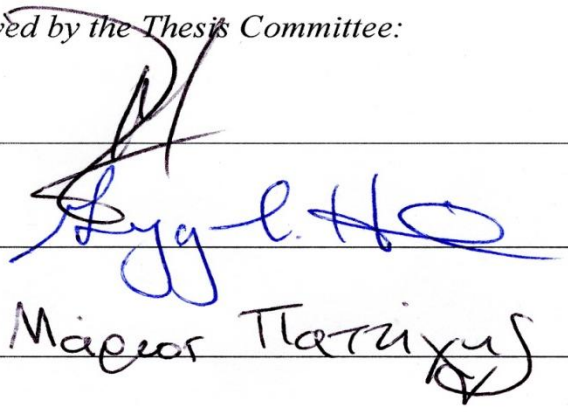
This Thesis is brought to you for free and open access by the Engineering ETDs at UNM Digital Repository. It has been accepted for inclusion in Electrical and Computer Engineering ETDs by an authorized administrator of UNM Digital Repository. For more information, please contact disc@unm.edu.

Luis Martinez
Candidate

Electrical and Computer Engineering
Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:



, Chairperson

AGILE TESTING METHODS FOR IC RADIOS

BY

LUIS MARTINEZ

BACHELORS OF SCIENCE IN COMPUTER ENGINEERING,
UNIVERSITY OF NEW MEXICO, 2006

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science in
Computer Engineering

The University of New Mexico
Albuquerque, New Mexico

December 2010

© 2010, Luis Martinez

ACKNOWLEDGEMENTS:

I herby acknowledge the great support and help of Dr. Jacob Rael of Broadcom Corp for his guidance, support, and instruction regarding IC radio design and testing. Through the long process of this project he showed great patience and provided valuable insight and ideas. I would also like to thank Guillermo Loyola or Broadcom Corp for his contributions to the JTAG portion of this thesis and his excellent instruction on interfacing RF chips and control machines. Finally, I would like to thank my advisor Dr. Ramiro Jordan for his patience and guidance throughout my graduate education and his help organizing and formatting this thesis.

AGILE TESTING METHODS FOR IC RADIOS

BY

LUIS MARTINEZ

ABSTRACT OF THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science in
Computer Engineering

The University of New Mexico
Albuquerque, New Mexico

December 2010

AGILE TESTING METHODS FOR IC RADIOS

BY

LUIS MARTINEZ

B.S., Computer Engineering, University of New Mexico, 2006

M.S., Computer Engineering, University of New Mexico, 2010

ABSTRACT

This thesis concentrates on the development of a new testing procedure for the radio used in Broadcom's WLAN and cellular chips. Verification and hardware testing for the radio is completed at the end of the development cycle by the fabrication of test chips which contain only radio specific hardware. Utilizing test chips allow for component specific design verification, performance analysis and debugging of the Integrated Circuit (IC) radio before large scale production. In this thesis we present a new testing method that is intended to improve the development time, lower costs, and provide more agility within the IC radio testing process. This thesis takes an in depth look at the development and deployment of these new testing methods. It provides an examination of the hardware requirements for testing IC radios and the software developed to automate test development and control testing procedures. The main components developed for this project are the Chip Communications and Control Module and the Phase-Locked Loop controller. Respectively, these components were designed to control the operation of a test chip and configure the mechanism that locks the chip's RF signal to a desired frequency. Previous testing methods include the use of Automated Testing Equipment (ATE) and small scale test platforms built around LabVIEW. These testing methods can be costly, impractical, require longer development cycles, and are not contusive to agile methodologies. The new procedures attempt to provide a better test platform for WLAN radios by addressing these issues and matching or exceeding the requirements of the previous test platforms.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction:

A common trait among test engineers is an insatiable obsession with control. Verification of test results requires patience, awareness and a strict regimen. Every variable must be accounted for and every scenario must be attended to. These goals, however, are in direct conflict with other extremely important aspects a test engineer must consider; speed and efficiency. In the design and application of integrated circuits (ICs), testing is oftentimes a bottleneck in the cycle of development. Although the ability to have governance of each and every variable in a controlled test environment is desirable, the practicality of doing so becomes increasingly difficult as the scale of an IC project grows. Ideally, testing will be fast, efficient, robust and cost effective.

Testing semiconductor circuits takes place in all stages of development. Before fabrication, circuit design is completed through simulation. During fabrication, parametric probe testing is used to check circuit continuity and interconnectivity issues during fabrication. Post fabrication, multiprobe testing is applied to a processed silicon wafer before the chips are cast into individual die. Multiprobe tests provide a variety of functional testing and are designed to quickly identify problems that may arise during fabrication. Final testing occurs as chips are separated from the wafer and packaged as individual units. In this stage, chips are run as they would be used in real world situations and to find issues that may have escaped previous testing.

For final testing, an IC may have many thousand individual programmable signals, each with a significant role to play in the function of the circuit. Because of the complexity involved in testing packaged ICs and a general lack of programming knowledge, there are many tools that offer a layer of abstraction. Tools such as LabVIEW and ChipScope are used for this exact purpose and they have made significant improvements in IC design and testing. Other innovations such as Automated Testing Environments (ATE) allow large scale testing of integrated circuits with little to no human involvement whatsoever. But what happens when testing? What happens when situations arise that are not helped, but hindered by the layer of abstraction which is intended to help the testing process? How do you bring costs down during final testing, and still provide adequate testing tools? This thesis will discuss the design and implementation of a text based testing method that is specifically applied to radio ICs. This test design provides a layer of abstraction, while offering other benefits such as better scalability, ease of development and version control. Completing the tasks require the utilization of many engineering disciplines. This includes software programming, IC design, IC testing, circuit design, and system integration. Much attention is given to the capabilities and resources of the Linux operating system and the examination of commercial and public software. The methods designed for this project were architected and implemented specifically for Radio test chips. This method was developed in conjunction with Broadcom Corporation for the Radio group in Broadcom's wireless LAN division and provides many industry implications.

Background:

In digital communications, a Radio refers to a device that can transmit or receive signals through modulation of electromagnetic waves with frequencies below that of visible light [9]. A radio will tune to a specified frequency and either broadcast or receive signals at that frequency. Radio ICs are integral components to a large array of wireless technologies. Wi-Fi, Wireless LAN and Bluetooth are just three examples of technologies that use radio frequency as a means of communication. Testing radio ICs proves to be an interesting challenge because of the scale of the device. Simulation and virtual testing is an important aspect of IC design, but it does not address real world situations that often arise during manufacture. As described in the article *Virtual Tests Comes of Age*, "while test simulation software is expected to reduce test time, it will not eliminate the need for real test" [10]. Their designs are not large enough to necessitate the use of Automated Testing Equipment (ATE), but large enough to require a sophisticated testing environment. Radios are developed as a System on Chip (SoC), which is to say it will be part of a chip that performs a number of different functions. After a radio is designed and simulated, it is fabricated into a test chip. A radio test chip is an intermediary implementation that is fabricated with the intention of verifying a design and testing for bugs on a platform that is completely isolated from a larger implementation, such as a WLAN chip. Obviously, the intention behind the fabrication of a test chip is to verify the radio's functionality before it is integrated into a larger design.

ATE greatly increases the ability to quickly and accurately test printed circuit boards (PCBs), ICs, and a number of related electronic components. In electronics ATE is

especially useful for large scale functional testing [7]. This technique involves applying a set of inputs to a device, then comparing the reported outputs to a set of expected outputs. Functional testing is ideal for production chips because it allows a test engineer to quickly test a high volume of units, commonly referred to as Units Under Test (UUT), to quickly identify and sort problematic units. The goal in testing a production chip with ATE is simply to determine where on the chip a defect, if any, is present. Once this is determined, the chip is sent back to a group of specialists in the area. Therefore, if ATE discovered a problem with the radio, it would be reported back to the radio group. For test chips, however, ATE is impractical and expensive.

Cost and scalability are the largest drawbacks for using ATE to test radio ICs. In an article for the EE times, Steve Carlson states, "the cost of testing complex system-on-chip designs will soon surpass the cost of manufacturing them [8]." One reason for this is scalability. In the same article, Carlson describes some of the costs, "for complex, high-speed SoC devices, testers cost between $2,000 and $9,000 per pin [8]." This is a common problem in IC design as it is often necessary to add new pins as design improvements and specification continually changes. Using the cost per pin described in Steve Carlson's article, if just ten chips are produced with fifty pins per chip the cost of ATE equipment would run between one million and four million dollars. This is before factoring the costs of setup, calibration and maintenance for the ATE equipment. For large companies like Broadcom it is not uncommon to design and fabricate eight to twelve test chips per year. Although it is always desirable to automate as much of any procedure as possible, it is just too expensive to continually develop thorough ATE for a

test chip. Also, at this stage in testing ICs it is extremely beneficial to have more interaction with the UUT than is commonly practiced using ATE. In order to save costs and allow for a more hands on interaction with the chip, a small scale test platform is needed.

Current Techniques:

Testing a radio requires the cooperation of a large group of people and often a diverse range of expertise. This group must decide on a test platform that ideally is easy to use, scalable, supports parallel development, and is highly customizable. The two major components for designing a design platform for a test chip are the physical interface and a software controller. The PCB board, which from this point on will be referred to as a test-board, is designed to allow interaction with the test chip. Most times, reading and writing to the chip is facilitated by a Joint Test Action Group (JTAG) interface. This interface will be expanded upon in the next Chapter. Software, such as National Instrument's LabVIEW or Agilent's Vee, is then used to create modules that will facilitate reading and writing to the chip and the programming of the chip's registers. Broadcom employs the use of LabVIEW for testing IC Radios. LabVIEW Modules are used for a variety of tasks, but the two most important are setting the radio's registers and controlling the Phase-Locked Loop (PLL). Setting the radio registers allows the test engineer to control the functional state of the radio. Specific configurations of the registers can determine if the radio is transmitting or receiving a signal, what frequency the radio is dialed to, and many other functions of the radio. The PLL will be covered in depth in Chapter 4. It is a device that is used by the radio in order to generate a stable frequency in which the radio 'locks' to and subsequently transmits or receives signals at.

The LabVIEW module for the PLL controller is the most complex and most important software module for testing IC radios. If the PLL controller does not work properly there is an extremely limited amount of testing that can be applied to the radio.

Although they can be very useful, there are many drawbacks to using commercial software like LabVIEW. Because LabVIEW is proprietary software, it costs money for each machine it is installed on. Also, because it is proprietary, there are no third party standards committees that manage LabVIEW such as there are using common programming languages [11]. Support for non Windows platforms is poor [11]. This means that although compatible with Linux and Apple LabVIEW is mostly used as a Windows application. LabVIEW ties the creation of user interfaces into a development cycle. It uses a graphical interface that allows non-programmers to build programs by dragging and dropping virtual representations of common electronic equipment and protocols. Hence, it can be inferred that a non-programmer can develop object oriented LabVIEW code, also known as G-code. Furthermore, it is fairly easy for a test engineer with little to no exposure to LabVIEW to create very simple LabVIEW modules. For complex models and large-scale projects, however, it becomes more important that a LabVIEW programmer posses an extensive knowledge of special LabVIEW syntax and tools. This requirement is one of the reasons that LabVIEW is not the optimal option for radio test chips.

Developing an interface for a radio test chip is a larger scale project and requires extensive knowledge of LabVIEW syntax. This requires some level of expertise and with

so many people using their expertise in other areas it becomes increasingly difficult for a large group of people to develop effective LabVIEW modules. As such, the responsibility of developing LabVIEW code is given to very few individuals. Instead of having a test procedure that would allow many individuals to utilize their expertise to develop procedures for the interaction software, there is deficit of available resources. Also, LabVIEW on a Windows operating system is somewhat more difficult when using repository systems. Without the use of a repository, there is no simple way to ensure that a LabVIEW module is up to date or that the same version of a module is being used by a large group of people.

Perhaps the largest drawback for using LabVIEW in the test platform of IC radios is the apparent difficulty to quickly modify or expand a LabVIEW module. As the speed of the development cycle continues to increase, so does the benefit for agile testing methods. Requirements for testing applications can change very quickly and, therefore, modifications and expansions to the test platform must also be completed quickly. Too often, when using LabVIEW, a module must be fully recreated due to a minor design change. For example, a recent update of a Broadcom radio included an increase in register width from eight to nine bits. This seemingly simple change required the entire test platform to be rewritten to accommodate the extra bit. Finally, because LabVIEW is a graphical based programming language, it is difficult to automate the creation of LabVIEW code. This is especially prevalent for simple yet tedious tasks, such as correctly programming a large number of registers to a predetermined value.

At Broadcom, there are alternative tools that can be utilized to test production chips. Dr. Guillermo Loyola of Broadcom Corporation wrote an in house built test suite, EpiDiag. It allows for interface with chips through both JTAG and PCI interfaces. EpiDiag is used with the Linux operating system and has many of the characteristics that would benefit the current test chip procedures. It is compatible with CVS and highly customizable. An individual can modify his EpiDiag workspace by writing some simple procedures using the TCL language. EpiDiag is often used to test production chips and other ICs that do not require extensive register level programming. To this point, it had never been applied to test chips, which were largely tested using LabVIEW platforms.

Evolving Methodology:

IC design is a constantly evolving discipline, and testing should be a part of that evolution, not a relic that hinders progress. For this reason, it was paramount that a new testing procedure be developed to address the issues of the previous test platform. The new testing procedure should be scalable, allow for more automation, allow easy use with a centralized repository system, be highly customizable, allow for parallel development, and allow for more versatility in testing. Dr. Jacob Rael of Broadcom Corporation proposed using TCL procedures similar to what would be found in the EpiDiag test tool, while using Python to automate some of the more repetitive tasks. The test platform would be ideal for the Linux environment Broadcom employs for IC design and simulation and be more compatible with the current CVS repository. A Linux based test platform would also be compatible with EpiDiag, and thus include an existing JTAG controller and TCL interpreter. The desire to incorporate these features into the IC radio

test platform, as well as mounting frustration using LabVIEW based methods, was the motivation for this project.

The first requirement to consider for the new test platform was the software. The software had to be able to perform all tasks completed using LabVIEW. The minimum requirements were to facilitate data I/O, interface with the radio test chip, and control the radio's PLL. In order to improve the test platform, the software was also required to be more agile in development, easier to manipulate, allow for more automation and be more contusive to parallel development. Linux was used as the primary operating system for the test system, which will be referred to as simply the Linux test environment. Using Linux, many of the necessary tools are already in place. Also, the distributed nature of the file system is beneficial to agile code development. Using tools already incorporated in Broadcom's distributed Linux OS, we could write Python code to encode/decode register data from the radio design into the testing environment, use TCL to develop a facilitator for the data flow, and use EpiDiag to interface the PC hardware and the chip. The software hierarchy is shown in Figure 1.

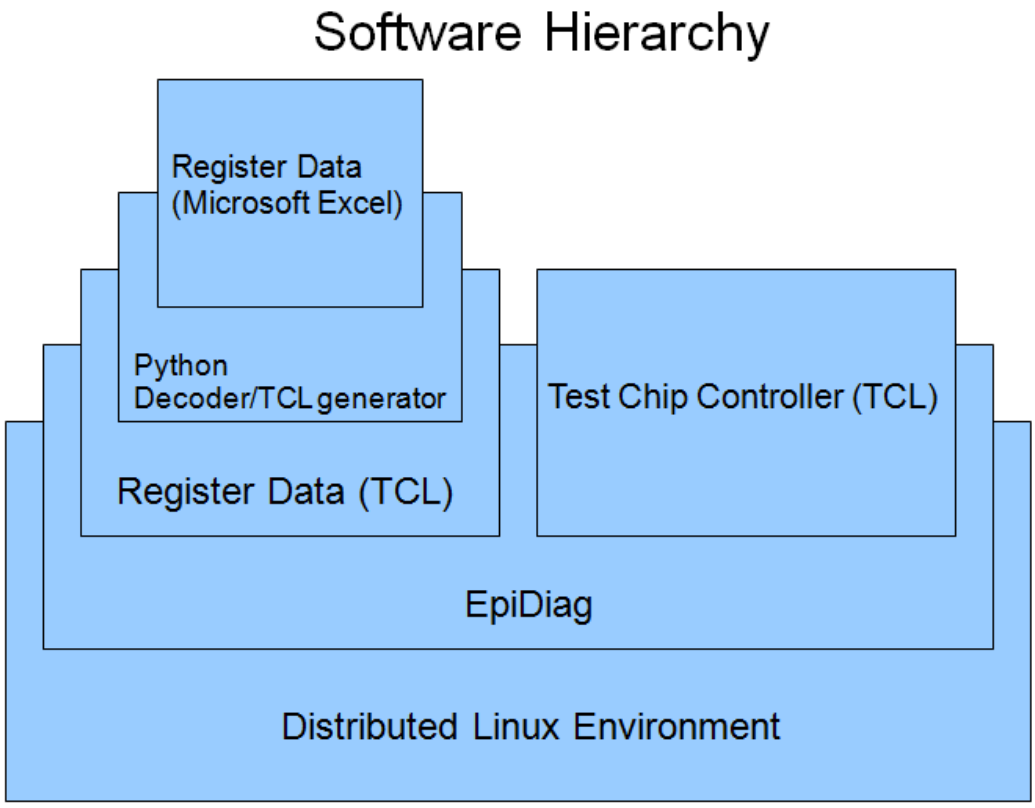


Figure 1 - A visual representation of the software hierarchy of the test platform

The second important aspect to consider was the hardware requirements. For this project, it was necessary to interface the chip with the controlling software. The standard procedure is to connect the control PC to a test-board that houses the chip via JTAG. This method was not altered, however certain changes for the JTAG circuit needed to be applied. This is discussed further in Chapter 2. The hardware connectivity is demonstrated in Figure 2.

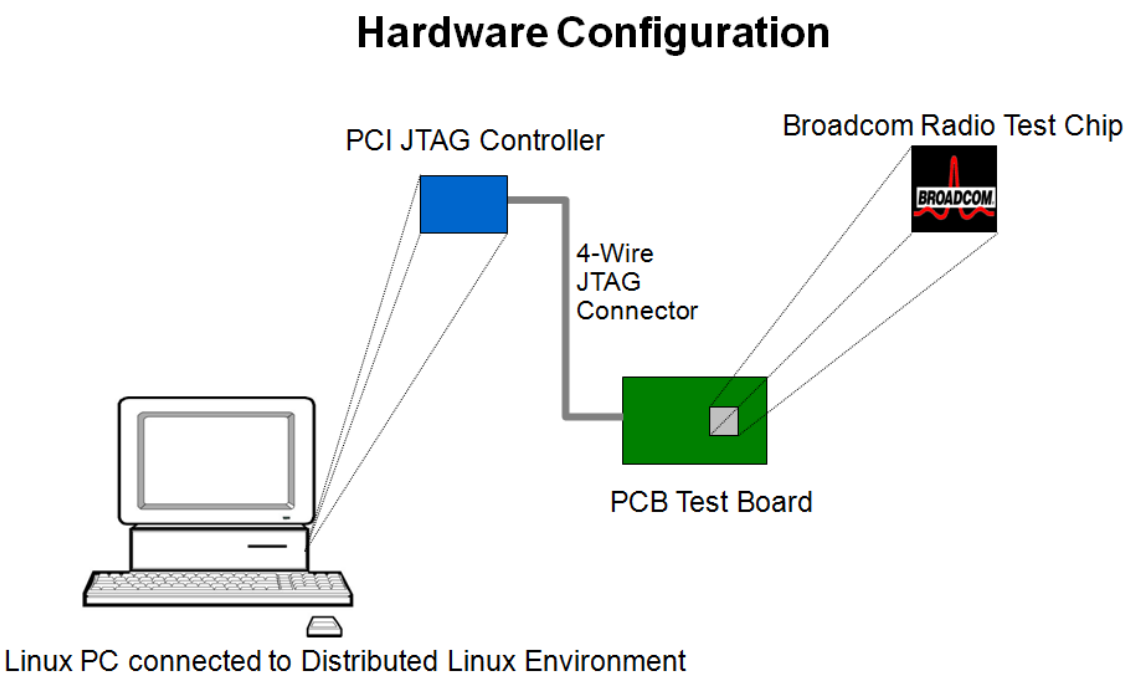


Figure 2 – An illustration demonstrating the hardware configuration

This thesis will discuss all aspects of this new testing method. Chapter 2 will focus on the development of a JTAG interface that can be easily transferred from one test chip to another. Chapter 3 will focus on the application of the register data to the test chip. Chapter 4 will discuss a PLL and give an overview of the PLL controller that is written in the TCL language. Chapter 5 will discuss the "bring-up" process-which is the moment these tools are applied for the first time to a new test chip, in order to lock a PLL to a specific frequency. Chapter 6 will discuss the development of new test procedures and parallel development. Chapter 7 will discuss the application of this test suite to the Microsoft Windows operating system, and finally, Chapter 8 will discuss future work and conclusions.

Summary:

This Chapter is used as an introduction to IC radio testing and some of the problems encountered in current testing methods. In the introduction some of the basic concepts of IC testing are discussed such as when testing takes place in the development cycle and why it is such an important issue in IC design. There is also a brief introduction to ATE and LabVIEW and how they fit in the testing at Broadcom Corp. Near the end of the introduction there is a small discussion of the issues in IC radio testing using current techniques and an introduction to the testing methods presented in this thesis.

The next section of Chapter 1 discusses some of the background of IC radio testing at Broadcom. ATE testing, while very beneficial for large scale production chips, is much too expensive for test chips and small scale testing. Small scale testing involves the use of proprietary software such as LabVIEW, which comes with its own drawbacks such as development time, parallel development, modifications, expandability automation. The final section gives a brief overview of the Linux Test platform which was developed in an effort to remedy some of the shortcomings of LabVIEW based test platforms. The software and hardware hierarchies are introduced as well as the basic requirements of the test platform. Finally, the last section describes some of the benefits of the new test platform when compared to ATE and LabVIEW based test platforms. These include agility, modifiability, expandability, easier automation, and cost.

Chapter 2

Introduction:

In order to properly test a chip a physical interface is required. There are a variety of physical interfaces that are capable of relaying control signals to and from a chip yet all serve the similar functions of applying power to the chip and providing a data I/O between the chip and a control system. This Chapter will focus on the JTAG interface, which is the transport control mechanism that facilitates the interaction between the chip and the control system for this project. JTAG is the common name for the IEEE 1149.1 standard for test access points as a method to test populated circuit boards and evolved into a commonly used method to debug integrated circuits [1]. In this Chapter there will be a brief introduction to the JTAG interface followed by descriptions of the JTAG circuit used for the project's test-board, how the JTAG controls are handled in software, and the verification of JTAG functionality for the project.

JTAG Interface:

As previously discussed, ATE is a powerful tool that greatly improves testing large quantities of chips; however it is nearly impossible to write ATE software that can anticipate each and every problem that may occur. Many Functional tests are done in the lab under the strict observation of a test engineer. Each chip has a test-board that allows direct interface with each input and output pin. While some chips use PCI or PCIe interfaces, the vast majority of all test boards are controlled using a JTAG interface. PCI and PCIe test boards do not utilize a JTAG sockets or JTAG busses, but still employ JTAG signals. The radio test chips that were used for this project were designed to use a

JTAG socket and bus. The JTAG interface is a special four or five-wire pin interface that is added to a chip. The connector pins are [1]:

- Test Data In (TDI)
- Test Data Out (TDO)
- Test Clock (TCK)
- Test Mode Select (TMS)
- Test Reset (TRST)

JTAG is a serial interface meaning there is only one data line available for input or output. The operating frequency of TCK is typically between 10-100MHz. Depending on if it is a read operation or write operation, one bit of data is transferred per clock cycle on the TDI or TDO wire.

JTAG support is available in EpiDiag and the tool can be used as a JTAG wrapper. A JTAG card is used to control JTAG signal flow and connects to a PC via a PCI slot. At Broadcom, JTAG cards utilize a 20MHz crystal oscillator. The PCI JTAG card has a five-wire JTAG pin interface and can facilitate the JTAG clock. This card is referred to as the JTAG master. EpiDiag's role is primarily to facilitate the clock of the PCI JTAG card and to encode/decode the JTAG signals. Until this point, EpiDiag had never been tested as being capable of interfacing a JTAG card with a test chip.

JTAG Circuit Design:

The JTAG circuit that was utilized on the test-board was designed to be simple and reliable. It was based on a circuit previously used for Broadcom test boards. Because EpiDiag had never been used to interface with a test chip, it was desireable to create a

JTAG circuit with as little latency as possible. Figure 3 displays a JTAG pin socket with each wire connected to a 10k pull-up resistor. The JTAG interface between the test board and the JTAG signals at the pin socket operate at 3.3V.

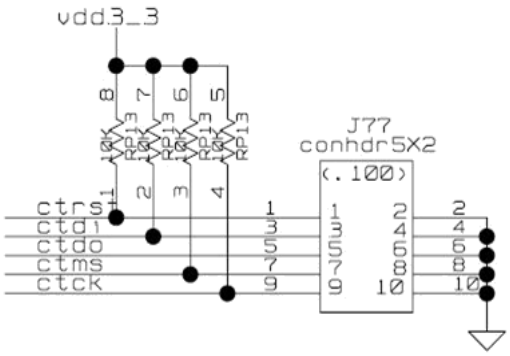


Figure 3 – The JTAG socket

Figure 4 displays a bus buffer, which converts the 3.3V input signal to 1.2V. This is the operating voltage at the chip level. The signals for TCK, TDI, TMS, and TRST are input signals and are down-converted. The TDO signal is up-converted through the buffer from 1.2V to 3.3V. The buffer for this circuit is the, the Texas Instruments Quadruple Bus Buffer Gate [11]. This buffer was chosen due to its low latency for voltage conversion.

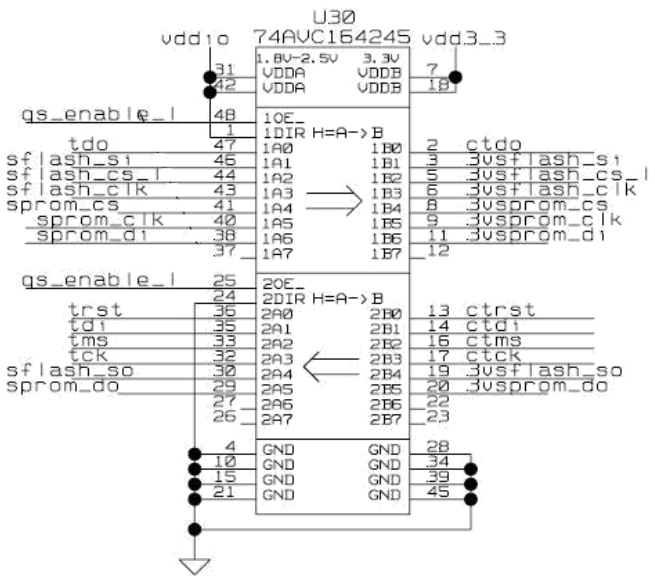


Figure 4 – The JTAG signal mapping

Handling Read/Write:

Reading and writing to a test-board is a large task due to the large number of registers that require setting for the radio test chip. Although most of this process is automated, there are procedures contained in EpiDiag that allow more direct control of the JTAG read and write functions. Unfortunately, this functionality had never been applied to test chips. Before this test method could be implemented, it was necessary to verify the ability to directly read and write to a JTAG register on the test chip using EpiDiag.

In order to read and write to a JTAG register, the JTAG master must first be configured to operate with the test chip's JTAG interface. This can be done using the TCL language and be incorporated into the overall test procedure. A test chip generally has registers with addresses between eight and ten bits in length, which eight bit data payloads. There

is a special radio ID register that identifies the radio ID and version number. This register has a data payload of thirty-two bits. The ‘jtag config’ command is a special EpiDiag syntax that is used to adjust properties of the JTAG interface. The following code shows this initial setup of the address, the data registers and the clock divider on the JTAG to its highest possible value. Increasing the value of the clock divider will slow the frequency of the JTAG signals.

```
set addr_len 10
set data_len 8
set id_len 32
jtag config clkd 30
jtag config ir $addr_len
jtag config dr $id_len
```

Writing to the radio is a straightforward process. The command for writing in EpiDiag is ‘jtag writereg’ and you need only to supply the destination address and the value to be stored at that address. The address must be in hexadecimal format, but the value can be hexadecimal or binary. Reading from the JTAG however, is somewhat tricky. Similar to the write command, the command to read from a register is ‘jtag readreg’, however the address for the readback path is offset. The offset itself is of varying magnitude. The general idea is that the offset is related to the total number of JTAG registers. If there are between 0x0 and 0x99 registers, the offset would to be 0x100 in hexadecimal. For example, if the offset is 0x100 and a value is written to address 0x4 then to read that value stored in at address 0x04, would read back from register 0x104. Similarly, if there are between 0x100 and 0x199 registers the offset is 0x200. An address stored in register 0x4 would be read back from register 0x204. For this project, the offset is 0x200. The code to readback from a register is shown below. The special case in the ‘if’ condition

refers to the radio ID register. When this is detected, the data length of the read register must be placed at thirty-two bits.

```
if {$addr != 0x01} {
    set addr [expr $addr + 0x200]
    set readval [jtag readreg $addr]
 return $readval
}
```

Testing the JTAG:

Testing of the JTAG consisted simply of reading and writing to a specific register using a distinct value such as 0x5a. The initial results were inconsistent, as some registers would read back the incorrect values while others appeared to be fully functional. This process was recreated in LabVIEW in an attempt to recreate the readback issue, however the LabVIEW test was able to prove the JTAG interface for the test chip and test-board was functioning properly. The next step was to trace the voltages of the signals from the board to the chip using a multi-meter to confirm the proper voltages from the JTAG socket to the chip. To enable measurements, the JTAG signals were set to continuously transmit on a loop. Using this technique, the proper voltages for each signal were traced before and after the buffer. The next test was to inspect the waveform of each signal with an oscilloscope, the results of which can be seen in Figure 5 along with a description of the signals.

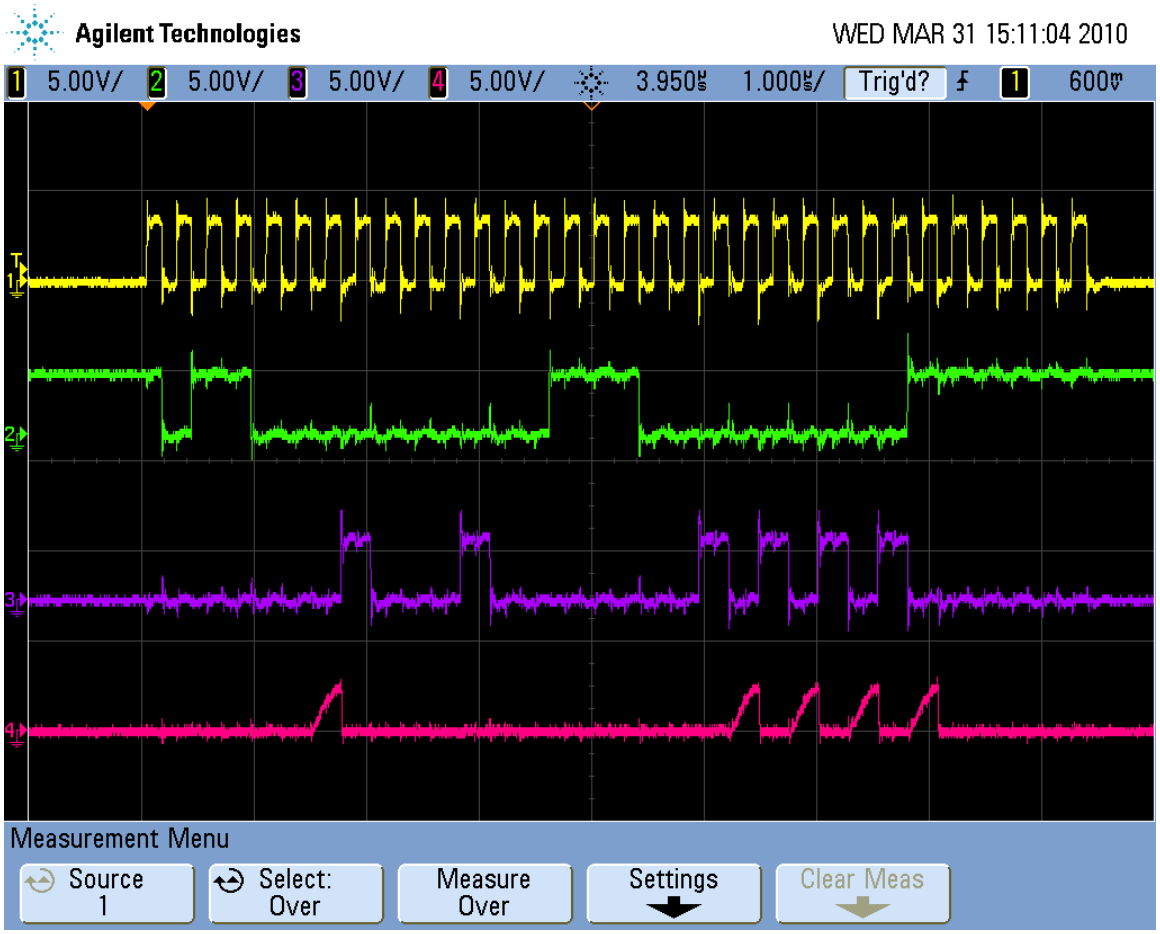


Figure 5 – Oscilloscope probe displaying the JTAG signals

- Channel 1/Yellow- TCLK
- Channel 2/Green – TMS
- Channel 3/Purple – TDI
- Channel 4/Pink- TDO

When measured at the JTAG socket, it is clear the TDO signal is weaker than the others as there is some signal loss apparent on the TDO wire. The solution to this problem was to slow down the clock of the JTAG master enough to handle the relatively weak signal, the crystal of the JTAG master was swapped from 20 MHz to 12 MHz, and the clock

divider of the JTAG master was set to its highest possible value. This resulted in a significantly slower JTAG. The signals ran at roughly 4 MHz as opposed to the usual 20 MHz. By slowing the JTAG signals, the readback function was now able to work reliably. Although the read/write for the JTAG is much slower than it would be using LabVIEW, for testing the radio the difference in speed is negligible.

Summary:

This Chapter focused on the development of the JTAG interface of the test platform and discussed the problems encountered and how they were resolved. Next, a brief overview of the JTAG interface was given as well as a description of the TMS, TDI, TDO and TCLK signals. A description of the physical components of the JTAG interface was given. This portion described how the JTAG signals are relayed from a lab computer running on Linux, to a PCI JTAG card, to a PCB test board and finally to the test chip itself. The physical JTAG circuit that was designed for this project was then illustrated. At this point there was a description of the buffer that was used to up and down convert the JTAG signals from 3.3V to 1.2V and vice-versa.

The second half of this Chapter dealt with configuring and utilizing EpiDiag, a proprietary Broadcom utility, to control the JTAG interface. There were also code examples that displayed how EpiDiag could be configured to work with a wide variety of JTAG interfaces and how to read and write to registers. Special attention was given to the differences in read and write addressing when using JTAG. Finally, there was a description of how the JTAG circuit was tested and debugged as well as a description of the challenges in this process.

Chapter 3

Introduction:

A radio functions as a state machine. By design, there are many programmable registers that can be set with binary strings. Changing the values of the programmable registers will alter the function of the radio. A major challenge in testing IC radios is correctly setting the programmable registers to achieve the desired radio state. Such is the topic of this Chapter. The first section describes the need for simulating the baseband settings for a test chip. The second section gives a description of the Microsoft Excel file that contains the state information of the radio. The next two sections how the Python programming language is used to parse the Excel file. The penultimate section of Chapter 3 describes how correct hexadecimal values are calculated for each register for a given radio state. The final section describes how TCL code that will be used to set a register state is auto generated in Python.

Applying Baseband Settings:

The baseband of a signal describes a range of frequencies from zero to the highest possible frequency for that signal. Technologies that transmit information using a baseband carrier include Ethernet and Composite Video. Wireless transmission at low frequencies will incur a lot of distortion and cannot be transmitted at the baseband frequency. In wireless communications, the baseband is modulated to a much higher frequency in order to allow for transmission at a higher radio frequency [12]. A consequence of the shifting is that the RF bandwidth of a signal is usually twice that of its

baseband bandwidth [12]. When a radio frequency is received by an antenna, it will be demodulated back to the baseband, thus the baseband in RF communications can be considered as the range of frequencies for a signal before and after frequency modulation.

A major difference between a production WLAN chip and a radio test chip is that there is no baseband signal configuration present on a radio test chip. The presence of a baseband signal configuration allows a production chip to function in a specific radio mode. When a radio mode is selected, the programmable registers on the chip are loaded with specific register values that configure the chip to operate in the selected mode. For example, the chip may be configured to transmit at 2.442GHz. For test chips, the baseband configuration must be applied to programmable registers on the chip manually. This configuration is developed during the design of the chip and includes the same values that will be loaded onto production chips. For instance, the radio that was tested has a set of existing settings for transmission in G band (frequency range of 2.4GHz to 2.5GHz), receiving in G band, transmission in a band (frequency range of 4.7GHz to 6GHz), receiving in a band, and many other modes of operation. The documentation for the baseband signals is stored in Microsoft Excel files. With the LabVIEW model, the baseband signals would be manually added to new LabVIEW module for each test chip. This practice was tedious and left room for errors. This Chapter will discuss methods to generate TCL procedures using Python, translate Microsoft Excel Data into TCL code, and how to format register data using TCL.

Excel File Description:

During the design and simulation of a radio, Microsoft Excel becomes a powerful tool. It allows a design team to organize data in a very readable and useful manner. When designing a radio, the design team creates a JTAG register map that maps each JTAG register to a specific address. The JTAG register map also lists which fields or ports are present in each register. The term ‘port’ is simply a reference to a connection pin from the register on the chip to an external I/O signal. A port can refer to a 1 bit wire or multi-bit bus. In the JTAG register map, the port for each register is assigned a starting bit, stopping bit, reset value, preferred value and a brief description.

Microsoft Excel is also used to store and organize radio configuration data. A radio configuration file is simply a mapping of the JTAG ports of a radio to specific hexadecimal values. When these bits are instantiated on the radio, the radio will be configured to a specific state. Common states for the radio include transmit, receive and configuration. In order to centralize the data for this project, a single file was created that was called the JTAG Master. The JTAG Master is an Excel file organized by worksheet. The first Excel worksheet was reserved for the JTAG register map, while each subsequent worksheet was devoted to configuration data for a specific radio state. Organizing this data into a single file has the benefits of having a single point of reference for register information and allowing version control using a repository such as CVS.

The first portion of this document to be discussed in detail is the register map or JTAG register description. The register map is simply a document that provides a detailed description for each programmable register used by the radio. Registers are organized by address and, with the exception of the radio ID register, most have a data payload of eight bits. A subset of the register data is the field name for each signal. The field name is an English abstraction that is used to describe a digital signal in the radio. A field name has a corresponding start bit, end bit and reset value. Table 1 gives a brief description of the important categories in the JTAG description.

Element	Description
register_name	A name or placeholder for each register. If none is given the default is REG[address]
Address	The hexadecimal address for each register
field_name	A description of bits associated with specific digital signals on the chip. Will also provide a width in big-endian notation.
start_bit	The least significant bit for a corresponding field_name and register
end_bit	The most significant bit for a corresponding field_name and register
reset_value	The default value for a field_name if none is given in the baseband description.
Documentation	A small description of the field_name, including any special notes for programming.
RTL_Name	The RTL name for a field_name used in simulation. Is equivalent to field_name.

Table 1 – Example of the register information from the Register Map Excel File

The radio state is a mapping between the field names for each programmable register in the radio and their corresponding binary and hexadecimal values. Each radio state has unique set of values that are programmed to the chip. The radio states for transmit and receive in 'G' band, for example, have vastly different register values. There is a one to one mapping between a field name and a given value. This value is represented in both hexadecimal and binary format. Table 2 gives an example for a field name, the corresponding hexadecimal value, and corresponding binary value for a register that is configured to transmit in 'G' band.

Transmit in 'G' band		
field_name	Hex Value	Bin Value
i_wrf_jtag_bandgap_B<3:0>	0x6	0110

Table 2 – Table displaying an entry in the Radio State Excel file

It is important to differentiate between the different radio states and to assign the correct binary value to the correct field name in the correct register. This process is outlined in the next section.

Python Automation:

The Python programming language is an extremely versatile scripting language that can be used for an impressive array of tasks. Python can be used for simple calculations and scripting or for massive object-oriented programs. For this project, Python was an

invaluable tool used for a variety of tasks, but the single most important was the translation of the JTAG registers from the Excel file into a TCL procedure. With the previous test platform, a test engineer would develop a LabVIEW module for each register state. For each LabVIEW module he would manually configure each register setting to match the data given in the Excel file. This is a long and tedious process and prone to mistakes. With the Linux test platform, a Python script was developed to automate this process. The script could be used to parse the Excel file, decode the register value, and store the value in a data structure that maps it to the correct hexadecimal register data to its register address. Automating this process using Python scripting saves large amounts of time and greatly reduces errors.

Once again, the purpose of the Python code is to parse the Excel file, decode the data and then auto-generate a TCL script that will have procedures that can be sourced to the test chip. There are unique tools and challenges for each portion, which are outlined below.

- A special tool is needed to parse a Microsoft Excel file using Python (this is not completely true as it is possible to create custom code to do just that, but that would be a thesis in itself). This is made possible with the use of a Python module named XLRD [3]. XLRD makes it possible for Python to decode data Microsoft Excel documents. Using XLRD, each cell of an Excel document can be saved into a Python data structure.

- Python does not natively support binary values. Character strings can be used to represent a binary value, but these are not mutable objects. There are tools

available that can treat a binary string as a mutable object. To do so, the MyHDL module is imported into the Python script. The MyHDL module is used to enable functions commonly associated with hardware description languages including classes for signal generation and simulation [2]. The intbv class allows binary strings to be interpreted as integers, which are mutable objects.

Decoding the Excel File:

In order to create a baseband we must first decode and store all relevant information from the JTAG description of the Excel file. For this we need to import a Python module called xlrd, which will interface Python with Microsoft Excel and make it possible to read from an Excel document. Using commands from xlrd, we are able to open an Excel document with the following commands:

```
wb = open_workbook(register_settings_xls)
wb.sheet_names()
sh_final = wb.sheet_by_name(u'JTAG_final')
```

Using the preceding code, it is possible to open an excel document, and then load all data from the Excel workbook titled ‘JTAG_final’ into a Python object called sh_final. This object is similar to a two-dimensional array, and is indexed by row and column. Accessing the object sh_final.cell_value(0,0) will return the value placed into cell (0,A) from the Excel file. The Excel values are placed into Python data structures called dictionaries and lists [5]. A Python list is similar to an array or queue. It is an iterable object that has elements indexed by position. Unlike C or C++ arrays, Python lists are not typed, and may contain integers, strings, floats and any other Python object in a single list. A Python dictionary is a hashable object that is indexed by an immutable object (integers, floating point values, strings, etc.) called a key. Each key in a dictionary is

unique and mapped to an associative ‘definition’, which can be any type of Python object such as integers, strings, lists, or even other dictionaries.

The JTAG register data is placed into a Python dictionary called, registers. This dictionary is indexed by the addresses of the JTAG registers. The definition for each address key is separate dictionary that is indexed by elements of the register, listed in column 1 of Table1. In this dictionary, the keys have, as their definitions, a list of the values for each element at that address. The data structure is illustrated in Table 2.

Key: address	Definition: Dictionary with corresponding keys and addresses		
0x06	Key: field name parameter	Definition: List of field names in register	
	field_name	Index	Element
		0	NOOP<2:0>
		1	wrf_i_jtag_bb_AACI_TRIM<3:0>
		2	wrf_i_jtag_bb_AACI_OFF
	Key: start bit parameter	Definition: List of start bits for field names in register	
	start_bit	Index	Element
		0	7
		1	4
		2	0
	Key: end bit parameter	Definition: List of end bits for field names in register	
	end_bit	Index	Element
		0	5
		1	1
		2	0
	Key: reset parameter	Definition: list of reset values for field names in register.	
	reset_value	Index	Element
		0	0
		1	0
		2	1

Table 3 – A table displaying the distribution of bits in a register

The code below demonstrates how to decode the Excel object and organize the data into Python data structures.

```
jtag_reg_keys = sh_final.row_values(0)[2:]
for rownum in range(1,sh_final.nrows):
    if sh_final.cell_value(rownum, 0)
     or sh_final.cell_value(rownum,0) == 0:
        addr_name = hex2int(sh_final.row_values(rownum)[0])
        key_vals = sh_final.row_values(rownum)[2:]

       tmp = key_vals #create list equal to tmp
        for x in range(len(jtag_reg_keys)):
            tmp[x] = []
            registers[addr_name] = dict(zip(jtag_reg_keys, tmp))
            addresses[addr_name] = dict(zip(jtag_reg_keys, tmp))
    else:
        if sh_final.cell_value(rownum, 2)[0:6] != 'idcode':
            key_vals = sh_final.row_values(rownum)
            key_vals = key_vals[2:]
            mytmp = dict(zip(jtag_reg_keys, key_vals))
            for cell in mytmp.keys():
                registers[addr_name][cell].append(mytmp[cell])
```

A similar but much simpler method is used to load each radio state into a Python dictionary. The radio state is a simple one to one mapping between a field name and a binary value. Accordingly, only a single Python dictionary is needed utilizing a registers' field name as the dictionary key and the corresponding binary value as the dictionary definition. The code below demonstrates this process.

```
sh_radio_state = wb.sheet_by_name(xls)
for rownum in range(sh_radio_state.nrows):
    radio_state_keys.append(sh_radio_state.cell_value(rownum, 0))
    radio_state_bvals.append(sh_radio_state.cell_value(rownum, 2))

bitvals = dict(zip(radio_state_keys, radio_state_bvals))
```

Calculating the Register Values:

Now that the register map and state values have been loaded into Python, the proper register values must be calculated for each radio state. The writing to a JTAG register via the EpiDiag tool requires the register value to be in hexadecimal format. Calculating this value is not as simple as to add up the value for each field name in a register. This is because a field name is assigned by position in the register. Therefore, if there were eight fields in a register of alternating value between 0 and 1, the register values in binary would be 01010101, which translate to 0x55 in hexadecimal format.

The register values are calculated by determining the binary value for each field name in the register map Excel file. This is accomplished by assigning the binary value of the field name from the radio state Excel file. The field name will match a similar field name found in the register map Excel file. This is simple to do in Python by using the 'has_key()' method to quickly search for the existence of a key in a dictionary. The following code demonstrates this comparison by parsing each field name in the Python dictionary *addresses*, which is the dictionary containing the register map information. A second Python dictionary called *bitvals* is derived from the radio state Excel file. It contains the mapping of the field names to their binary values. It is here the has_key() method is used.

```
for x in addresses[addr]['field_name']:
    if bitvals.has_key(x):
```

If there is a field name in the register map that is not represented in the radio state, that field is given its reset value. If the field name is represented in the *bitvals* dictionary, then the binary value taken from the *bitvals* dictionary is stored in the field. After the binary

values for each fieldname in a register are concatenated, it can be converted into the hexadecimal representation. To do this, the binary string is first converted to an intbv() object. The intbv() method allows the representation of a binary string as an integer value. Because the registers are represented with 8-bit values the intbv() object may repesent any value between 0 and 255. Because this object is interpreted as an integer, it is then converted to hexadecimal format and the register value is completed. The intbv object and final hex value are demonstrated below.

```
hexbv = hex(intbv(bv,0,256))
hexbv = hexbv.strip('L')
```

Auto Generation TCL Procedures:

Writing test procedures in any language is a tedious process. For integrated circuits, this is especially true because most test procedures involve correctly setting a massive amount of registers. Writing long repetitive code by hand takes large amounts of time and is prone to mistakes. Automating these types of tasks saves time, mistakes and most importantly, boredom. After the success of automating code for the baseband registers it was only natural that similar processes would be implemented for other long testing procedures. The test procedures to control the integrated digital to analog converter (IDAC) and the FM tuning registers are especially long. Luckily, the data to control the IDAC and tuning registers is stored in Excel files similar to those that housed the baseband data. Auto-generating the TCL procedures to control the IDAC and the FM Tuning registers was simply a matter of modifying the procedure used to generate the baseband register procedure. Please refer to Appendix C and Appendix D for the code used to generate the IDAC and FM registers respectively.

Summary:

Chapter 3 focused on how the baseband register data is decoded from Microsoft Excel data into TCL procedures and how Python can be used to automate the process. Applying the baseband register data will configure the radio for a variety of operational states including those intended for transmission, reception, and calibration. The Excel data is categorized in two parts, the JTAG register map and the Radio Configuration Data. The register map and contains a full description of all programmable registers available in the radio including the address of the register, the ports within the register, and the size and position of each port. The radio configuration data assigns the ports of the radio to hexadecimal values. When the ports of the radio are assigned to these values, the radio is programmed to function in a given state. For example, the TxG state has ports that will enable the radio to transmit data in G band. The second half of the Chapter describes a Python utility named tc_reg_gen.py. This utility was written to complete three specific tasks:

1. Assigning the value of each port described in the radio configuration file to the correct register position in the register map.
2. Calculating an 8-bit hexadecimal value for each register based on the values read from the configuration file.
3. Creating TCL procedures that could transfer the register data to ports on the chip.

Auto generation of large blocks of code helps to save time, resources, and reduces bugs or other mistakes that commonly occur when writing large portions of code.

Chapter 4

Introduction:

In Chapter 2 it was shown that it is possible to read and write to a radio register via JTAG interface and using EpiDiag as a controller, so long as EpiDiag is configured correctly. Chapter 3 demonstrated a method to convert existing register data from a Microsoft Excel file into a TCL procedure. By loading the TCL procedure into EpiDiag, the register data could be written to the radio registers on the chip. The focus of Chapter 4 is the consolidation of these two concepts and various other additional utilities into a single piece of software. The purpose of the software is to provide a means of facilitating all communication and control of the chip. This Chapter will describe the function of the Chip Control and Communication Module (CCCM) while placing special emphasis on some of the more important procedures in the module.

The Chip Control and Communications Module:

The main objective of the CCCM is to provide a toolbox in which all facets of control and communication with the chip can be called, accessed, and modified. The CCCM is designed as a living code that is constantly evolving and changing. Portions of code can be added by anybody for any specific reason. This is important to the concept of agility in testing. The code is stored in a CVS repository and stable versions are periodically checked in. Anybody with access to the CVS server can import a local copy of the CCCM then edit, customize, or add procedures that suit their specific needs. If a portion of code is useful and can be used by many people, it can be added to the repository version of the code.

The CCCM is easily modified and very mutable, but that is not to imply it did not require a comprehensive framework and a complete set of built in utilities. The core functionality of the CCCM is to facilitate data I/O to the chip and the core procedures of the CCCM are encapsulated from the more specialized procedures on the chip. This aids the agility in coding the CCCM as specialized procedures exist independently and do not disrupt the core functionality of the module.

Manipulating Register Data in TCL:

A large concern for this project was how to manipulate register data using the TCL language. Reading and writing to a register using TCL is extremely easy, however altering specific bits within a register presents a certain set of challenges. Specifically, we needed to answer the following questions:

1. How do we convert the integer or hexadecimal data that is read back from a register into a binary representation?
2. How do we ensure the size of the binary string remains consistent?
3. How do we change a specific bit or range of bits within our binary string without affecting the other bits in the string?
4. What is the best way to convert the newly formed binary string back into a hexadecimal representation so it can be written back to a register?

Although Broadcom did have a TCL method to address these issues, they were designed specifically for other applications and were not easily integrated into this test environment. For this reason, the TCL function *format_register* was created for use with this project. The following is a brief description of how format_register is used and some

of the more important portions of code that is used in the function. The complete format_register code can be found in Appendix B.

When the function is called we pass in the initial register data in hex, the starting bit that will be altered, the ending bit that will be altered, and finally the value that will be inserted. The return value is an 8-bit hexadecimal number that can be written back to the register. An example of how to use the function is as follows.

```
set foo [format_reg $init $insert 3 1]
```

In the preceding example if init was set to be 0xff and insert was set to 0x0, then the value returned and set to foo would be 0xf1.

Using TCL to convert the value read back from a register into a binary string is, in theory, extremely simple. TCL has a built in function *binary_scan* that can be used to return a binary string from a given input value, but there is a heavy reliance on proper formatting of both the input value and return string. The single largest hurdle in using this function is correctly formatting all data. The read function will return data that is stored at a specific address and is interpreted in TCL as a hexadecimal string as opposed to a hexadecimal value [5]. This simply means if we set a variable in TCL it will be interpreted as a string until the variable is parsed using another TCL function. For most built in TCL functions this is not an issue, however, *binary scan* will return a ‘type’ error when it is passed the hex string read from a register. This can be addressed by simply formatting the hex string as an integer and then reformatting as a hex string. This effective, yet far from elegant fix is shown below:

```
#set variable to register value
set init_val [read_radio 0x3]
set init_val [expr int($init_hex_str)]
#convert back to hex
set init_hex_val [format %x $init_val]
```

Once the input data is properly formatted, we can use the *binary scan* function to convert the reformatted hex string into a binary string. Concurrently, we can control the length of the binary string because the *binary scan* function utilizes many string formatting techniques, including denoting the length of a string. For this project, the registers are always 8-bits wide; hence the binary string is also set to be 8 characters wide. Controlling the width of the binary string is an important task because it helps to control data integrity. The code below demonstrates the use of *binary scan*.

```
if { $init_val > 15 } {
#   puts "initial hex val $init_hex_val"
    #convert to binary number
    binary scan [binary format H2 $init_hex_val] B8 init_bin_val
   } else {
      set init_msb 0000
      binary scan [binary format H2 $init_hex_val] B8 init_bin_val
      set init_bin_val [concat $init_msb$init_bin_val]
   }
```

At this point the register data has been properly formatted as a binary string. Similarly, we convert the insertion value using the exact same method. With both the register data and the insertion data now represented as binary strings, we can use the start bit and the end bit to properly slice register's binary string. In TCL, the binary string is indexed as an array starting at 0 from the first character in the string. By contrast the register's byte order is reversed and first character of the binary string would correspond to bit 7 of the register. Because we know the register size, we can retrieve the correct bit from the TCL

array by subtracting both the start and the stop bit from the width of the resister. The most significant bits of the initial binary string are retrieved by slicing from the beginning string to the register width subtracted by the start bit, while the least significant bits are taken by slicing from the register width subtracted by the end bit to the end of the binary string. Concatenating the MSBs with the insertion string with the LSBs creates the new binary string.

Before the new value can be written back into the register, it must be converted back into a hexadecimal value. Formatting strings using functions from the standard TCL library proved to be difficult to implement and test. In order to save time and speed up the coding process the string is again broken into 4 bit segments and mapped directly to corresponding hex values. Finally, the register data is at its final state, and can be written back to the register.

Summary:

This Chapter is used as an introduction the CCCM or the Chip Control and Communication Module. The CCCM is the primary facilitator of data in the Linux test environment. The first section of this Chapter is an introduction to the CCCM and an overview of the benefits of having agile code for a project of this nature. The second portion of this Chapter is an overview of some of the challenges involved in controlling I/O data and properly formatting the binary code for the registers on the chip. Some of the more difficult challenges involved altering specific binary values in a register without altering the surrounding values. For this, the format_register function was developed with

the specific requirement of manipulating binary data for a register, while keeping data integrity. The rest of the Chapter concerns how this was accomplished in detail.

Chapter 5

Introduction:

Phase-locked loops (PLL) are primarily used in communication systems for a variety of applications, such as recovering the clock from digital data signals, performing frequency and phase modulation and demodulation, and to synthesize exact frequencies for receiver tuning. In wireless communications, a PLL is implemented as a digital logic device and incorporated into a microchip. The PLL functions as an oscillator whose frequency is locked onto some frequency component of an input signal [6]. In other words, if a PLL is given a reference frequency of 2.4GHz, the PLL will increase or decrease its own output frequency until it too is 2.4GHz. A PLL performs a similar function to the phase of an input frequency and will also match the phase of a controlled oscillator to the phase of the input signal.

A simpler explanation of PLL is to view it as an example of a negative feedback system [6]. A PLL will compare the frequencies of two signals using a phase detector (PD). The first signal is called a reference frequency and is derived from a very stable source, often a crystal oscillator. The second input is given by feedback from the output of a controlled oscillator. If the signal frequencies do not match each other, an error signal is produced. This error signal is proportional to the difference between the two frequencies. The error signal is then passed through a low-pass filter (LPF). The filtered signal is used to drive a voltage-controlled oscillator (VCO), which creates the output frequency. The output frequency is passed through a frequency divider and then back to one of the inputs of the phase detector. This produces a negative feedback loop. If the output frequency diverges

away from the input signal, the error signal will increase. This will eventually drive the output frequency in the opposite direction, in order to reduce the error; therefore it is easy to illustrate how the output is “locked” to the reference frequency.

Most electronic systems provide reference frequencies that operate at tens to hundreds of Megahertz. For this project, the reference frequency is a crystal oscillator (XTAL) that has a frequency of 20MHz. The details regarding this will be discussed in the next Chapter. Because the reference frequency for most PLLs are in the tens to hundreds of Megahertz, and the operating frequency of the PLL is in the Gigahertz range, the frequency from the XTAL is divided to a desirable input range before it is applied as the reference frequency. This is best illustrated by modern processors, which operate with clocks from the high hundreds of Megahertz to a few Gigahertz, yet have relatively low speeds for reference crystals. These clocks are provided using clock generating Palls. This process is known as clock generation. In WLAN chips, the primary goal of the PLL design is clock generation.

The PLL in Depth:

Figure 6, shown below, is a visual representation of the PLL.

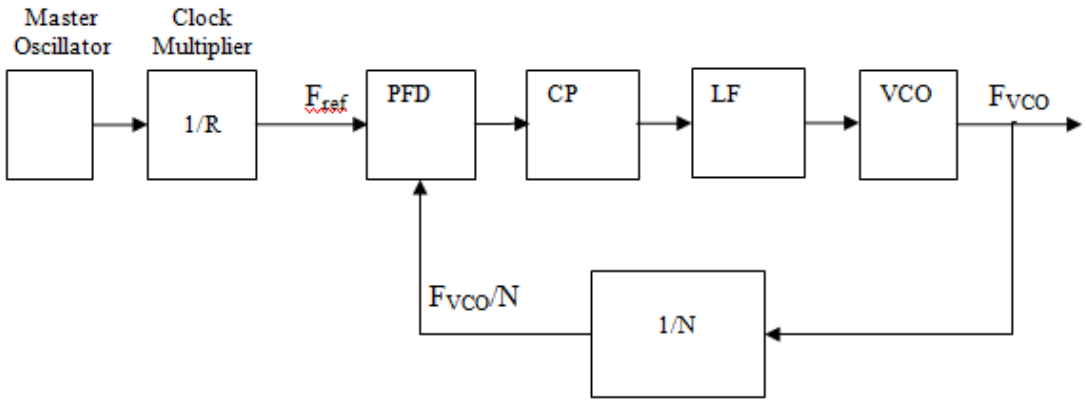


Figure 6 – A visual diagram of the PLL

The crystal oscillator generates a clock that acts as a reference clock for the PLL, which goes to the phase frequency detector (PFD); which also has an input from the VCO. Depending upon the output of the PFD, the charge pump (CP) induces a current into the loop filter that is designed to work at a specific bandwidth. The basic function is that of a current to voltage converter that works in a specific bandwidth. The VCO is the main block of the PLL. The VCO takes the input from LPF, and depending upon the type of VCO, generates a clock that is feedback through though a divider circuit. This divides the F_{vco} by the user-defined value, which in turn is a second input of the PFD.
Further details for each portion of the PLL are provided below.

Crystal Oscillator – The purpose of the crystal oscillator is to generate a reference frequency. For this project, we used a 20MHz crystal. Accordingly, the initial reference frequency is also 20MHz.

PFD – A Phase Frequency Detector takes two signals as inputs. The crystal oscillator provides the reference frequency, F_{ref}. The VCO frequency (F_{vco}) is provided as feedback through the Voltage Controlled Oscillator. The PFD output is used to control the VCO in such a way that the phase difference between the two inputs is held constant [6]. For this reason, the PLL is referred to as a negative feedback system. An *n*-state PFD may be extended to as many states desired, but a three state PFD is the most widely used because it is simple and can act as both a phase and frequency detector. [Insert reference] A state diagram for the circuit is shown in Figure 7.

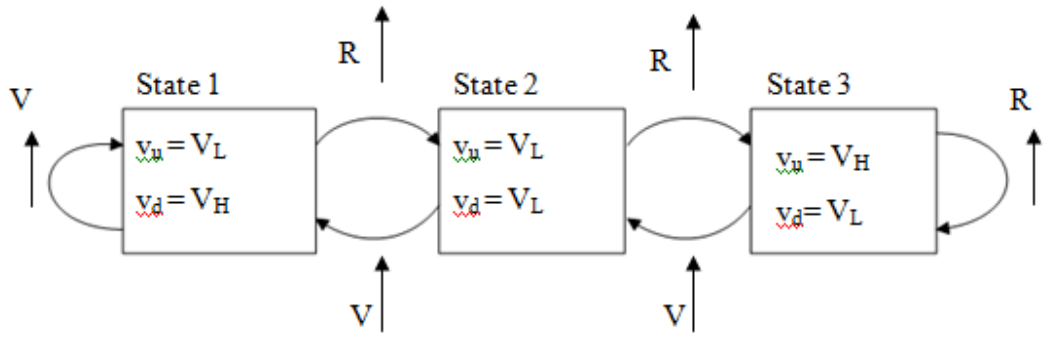


Figure 7 - The PLL state diagram

To expand upon the state diagram, we have our VCO frequency (V) and our reference frequency (R) these are digital frequencies represented by square wave clock cycles. The up arrow denotes a clock event. When R is triggered, we move to a higher State, while V moves toward a lower State. Suppose we are initially in State 1, if there is a clock events toggle for the reference frequency, we will move to State 2. If the VCO output triggers the next clock event we move back to State 1. As long as the VCO and reference frequencies toggle between each other, we will alternate between States 1 and 2. If the

VCO is consistently falling behind the reference in phase, eventually there will be two reference events before a single VCO event and we will be in State 3.

The PFD output is denoted by v_u and v_d, which are translated as ‘up’ and ‘down’ respectively. V_H and V_L correspond to logical voltage high and logical voltage low and act as a switch for signals, which are then passed to the charge pump. From the state diagram we can infer in State 1 our reference frequency is running slower than our VCO frequency and must be decreased. Similarly, we can clearly see that both signals are in phase in State 2. The charge pump uses these values in the order they arrive.

Charge Pump – The charge pump has input dependant on the PFD. It will receive the up or down signal, which will control switches to steer current in or out of a capacitor. The purpose is to cause charge or discharge the capacitor. The time the switch is turned on is directly proportional to the phase difference of the VCO and reference signals. The voltage from the capacitor will be passed through a low pass filter and ultimately tune the VCO. The current value is increased when the ‘up’ signal is received by the charge pump, similarly, when the ‘down’ signal is received the current value is subtracted. A pseudo implementation of the charge pump is as follows

If up>down then:

    Icp = x (where x is the current induced from the current sources in the charge pump)

Else if down<up:

    Icp = -x

Else:

Icp = 0 (where the output of the charge pump is drained to ground)

Loop Filter – A loop filter is used in order to attenuate the PFD signal at high frequencies, while still allowing for dc unity gain. If an attenuator is used instead of a loop filter it is possible to effectively reduce the ac gain, K. The drawback to this method is a reduction in DC gain, and therefore the dc voltage V_{co} the PFD can provide to the VCO is now limited. This greatly reduces the frequency range of the PLL and therefore, using a loop filter is the preferred method [6]. Adding a capacitor to an attenuator forms a simple loop filter. This is represented in Figure 8.With a large capacitor, the DC path to ground is blocked and the dc component is not attenuated [6].

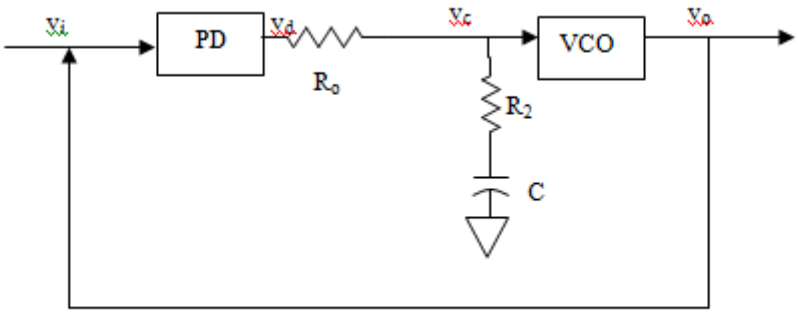


Figure 8 – The loop filter

Equation 5.1 represents the transfer function for the loop filter.

$$F(s) = K_h \frac{s+\omega_2}{s+\omega_1} \tag{5.1}$$

Where

$$K_h = \frac{R_2}{R_o + R_2} \tag{5.2}$$

with

$$\omega_1 = \frac{1}{(R_o + R_2)C} \tag{5.3}$$

and

$$\omega_2 = \frac{1}{R_2 C} \tag{5.4}$$

The signal flow for the loop filter in the PLL is demonstrated in Figure 9. This signal flow accounts for the gain of the loop filter, F(s) [6].

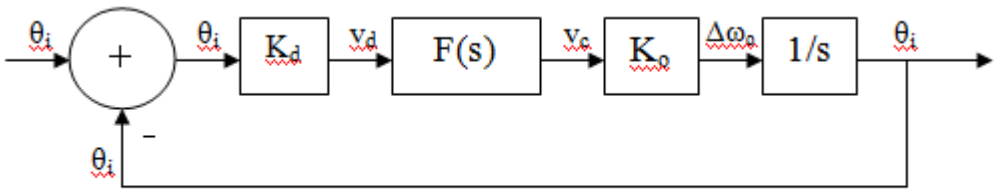


Figure 9 - The PLL signal flow

Voltage-Controlled Oscillator – A VCO is simply defined as an electronic oscillator. A voltage input is supplied to the VCO, which in turn controls the oscillation frequency [6]. As stated before, the phase detector generates a voltage v_d proportional to the phase difference of its two input signals; the vco θ_o and the reference θ_i. This difference is applied as a control voltage v_c which is passed to the VCO to adjust the VCO frequency ω_o. Through negative feedback, this causes $\omega_o = \omega_i$, thus both the phase and frequency are locked [6]. The output frequency deviation can be defined as $\Delta\omega o$, where

$$\Delta \omega_o = \omega_o - \omega_i \tag{5.5}$$

$\Delta \omega_o$ is a measure of how far ω_o is from its average lock. The VCO gain is defined by K_o, where

$$K_o = d\omega_o / dv_c = d\Delta\omega_o / dv_c \tag{5.6}$$

Using equation 5.5 and 5.6, we can model the frequency deviation by

$$\Delta\omega_o = K_o(v_c - V_c) \tag{5.7}$$

This model can be illustrated using the signal flow shown in Figure 10.

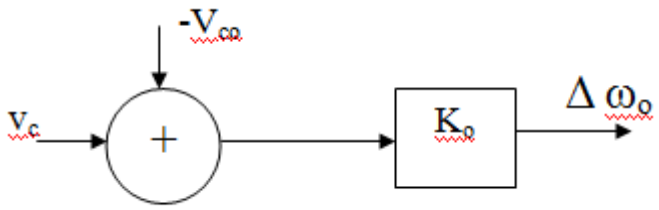


Figure 10 – The signal flow of the VCO

For this thesis, specifically, we are referring to a voltage-controlled crystal oscillator. This type of VCO is beneficial because we have an accurate external reference that we are adjusting our output frequency to match, which is especially useful when designing clock generators. This is exactly the goal of the VCO used in the radio. A clock generator is an electronic oscillator that is designed to provide signal timing to synchronize operations in digital circuits. Some design parameters to consider when designing a VCO are tuning voltage, range, center frequency, and jitter. Jitter must be minimized in radio transmitters and receivers.

Divider – Any number of dividers may be used to design a PLL, with the only limitation being which is best for the type of PLL being used. The output for this PLL employs a very high frequency range, so it makes sense to divide at a very high value and bring the signal into phase. At this point, a choice must be made to use either an integer divider or a fractional divider. For the purposes of this project, a multi-modulus divider (MMD) is the best option because this will allow the PLL to tune to a specific band. Thus, the divider used for this PLL is fractional.

The PLL Controller:

Properly controlling the registers of the PLL is the most essential task involved in testing of WLAN chips. The function of the PLL Controller is to configure the PLL to lock to a specific frequency. If the PLL can successfully lock to a specific frequency, the WLAN chip may be configured to transmit a signal or receive a signal on the specified frequency or 'channel'. More importantly, a locked PLL displays the overall design of the chip is more or less, 'working'. Controlling the PLL is simply a matter of properly setting a large block of control registers. These registers can be accessed via JTAG interface. The goal in creating the PLL controller was to prove that it could be written quickly, was easily modified, could lock the PLL to multiple frequencies, and most importantly could effectively control the PLL as well or better than previous methods using LabVIEW.

As discussed in Chapter 2 and 3, it is possible to interface with the chip via TCL code and a JTAG interface. The PLL is a part of the chip and thus requires the initial setup discussed in the CCCM in Chapter 3. For this reason, the PLL controller was not written as its own module, but rather as a procedure within the CCCM named rf_pll. The rf_pll procedure will lock the PLL to a frequency that is determined via user input or function call from a separate TCL procedure. The XTAL value passed to the function corresponds to the value of the reference crystal in MHz. If the reference crystal were 20MHz, the XTAL variable would have a corresponding value of 20. This value also determines the reference frequency for the PFD. The input frequency is limited to the IEEE 802.11 standard for A, B and G bands or more specifically between the ranges of 2.4GHz-2.5GHz and 5GHz – 6GHz. The input frequency is used as a reference for the local

oscillator (LO) and will determine the band the radio is operating in which will determine the initial VCO frequency. This is not the final frequency for the VCO, as that is determined by the PLL loop, but is used for calibration purposes. The VCO frequency is 2/3 the input frequency for A band and 3/2 the input frequency for B/G band. The Sigma-Delta divider for the VCO is calculated by dividing the VCO frequency by the reference frequency. This is demonstrated in the following code:

```
proc rf_pll { xtal channel } {
    if { $channel > 2400 && $channel < 2500} {
        set band g
    } elseif { $channel > 4700 && $channel < 6000 } {
        set band a
    }
    if {$band == "a"} {
        set Vco_freq [expr $channel*2.0/3 }
    else {
        set Vco_freq [expr $channel*1.5]}
    #Sigma-Delta Divider
    set pll_sg_val_int [expr floor($Vco_freq/$PFD_ref_freq)]
    set pll_sg_val_int [expr round($pll_sg_val_int)]
```

The most difficult task in correctly setting the PLL involves calculating then properly setting the register values for the loop filter and the charge pump of the PLL. Using the reference values, it is possible to properly calculate the register values that are used to configure the PLL. With extensive use of the format_register function discussed in Chapter 3, the register settings that were calculated for PLL can be applied. In test chips it is advantageous to not have predefined register values in order to properly optimize register settings and the PLL functionality. In production chips, the register values are defined in the baseband and programmed directly. The following reference values are used to calculate register settings for the loop filter, charge pump, and VCO.

- Low End VCO Frequency = 3000
- High End VCO Frequency = 4200

- KVCO Reference (VCO gain) = 30
- Low End KVCO = 27
- High End KVCO = 68
- N Divider Reference = 90
- N Divider desired value = 140
- Loop Bandwidth Reference in KHz = 270
- Loop Bandwidth Desired = 260

By utilizing these initial settings, we are able to derive and apply the correct register settings for each component of the PLL. The code for the PLL controller is displayed in Appendix D.

Summary:

In this Chapter, the focus is on the Phase-Locked Loop or the PLL for short. The PLL is the central component of the WLAN radio and all functionality of the radio is dependant of the PLL. A PLL is a negative feedback system that will increase or decrease the oscillation of its output frequency to match that of an input frequency. For digital communications, this means the PLL can be used as a means of setting or locking a specific frequency channel in order to transmit or receive a signal to that specified channel. There are two sections to this Chapter. In the first section, an in depth look is given to the PLL circuit and the function of each component. A brief summary of the components of the PLL is as follows.

- The crystal oscillator is used to generate a reference frequency.

- The PFD is a device which compares the output frequency with the reference frequency
- The charge pump will increase or decrease a voltage dependant on input from the PFD
- The Loop Filter attenuates the signal
- The VCO will increase or decrease the output frequency depending on the voltage from the charge pump.

The second portion of this Chapter regards how the PLL is controlled in software. Special attention is given toward the setup of the PLL and changing programmable resistors that can simulate a reference frequency.

Chapter 5

Introduction:

In the previous Chapters, we discussed the basics of the project and the methodology that was put in use. This Chapter will focus on the implementation of the methodology. The first section discusses the bring-up process. In wireless communications, this is the process of placing a fabricated chip onto a test board and testing for basic functionality on the chip. Next is an overview of the results of the bring-up process as a result of using the methodology discussed in the previous Chapter. Finally, there is a short discussion of some of the unexpected problems encountered during the bring-up process.

The Bring-Up Process:

The goal of the bring-up process is to confirm that the IC design of the radio is functional. The bring-up process refers to the initial testing of a radio test chip immediately after the chips fabrication. This is especially valuable for new and untested designs because it provides confirmation of the IC design on a physical medium, may discover complications in fabrication, and is a quick method of discovering critical bugs or glitches in the design. The bring-up process is also an important part of the fabrication design and the design of the test board. Errors in the fabrication are seen on the chip as unresponsive or ‘dead’ spots that are unresponsive when it can be proved that proper voltages are applied. For this project, two separate fabrications were used and tested in the bring-up process. For the test board design, the bring-up process will confirm if the test board itself is suitable for testing. This means the test board can be used to apply

proper power to the chip and successfully routes signals to/from the chip and some I/O interface.

For an IC radio test chip, the primary goal of the bring-up process is to transmit a tone and lock the PLL to set of frequently used frequency channels. Unfortunately, it is rare to lock the PLL in the first attempt. When this happens, it is advantageous to have a test platform that displays agility and adaptability. For instance, if it is determined that the problem with a chip's functionality is an incorrectly written register, this may indicate a bug in the software used to configure the chip. It was proven over and over again, that altering the Python code for the register generator or the CCCM could be completed in minutes, while debugging a LabVIEW module could take hours or even days. The hardware setup for the bring-up process is similar to that shown in Figure 2 in Chapter 1. Shown below is a list of the additional hardware required for use the bring-up process:

- Spectrum Analyzer to display power vs. frequency of RF signals.
- Oscilloscope to test clock and crystal frequencies.
- Frequency Generator which is used to apply an RF signal to the test board when measuring the receive signals.
- DC Voltage Source to power the test board.

Figure 11 displays a real life implementation of the hardware setup.

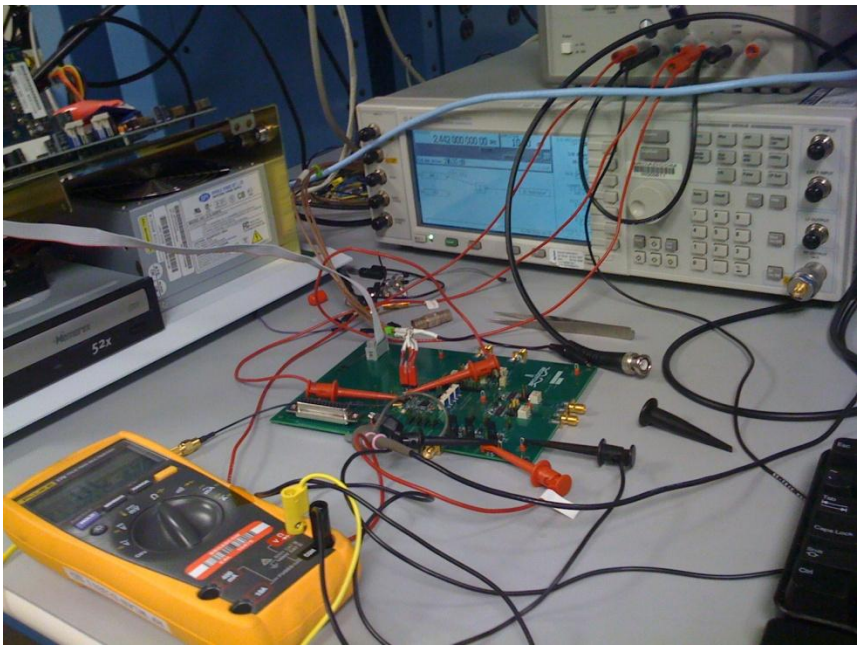


Figure 11 – Picture of the laboratory bench setup

After all hardware is in place and properly hooked up, the time comes to execute a test plan. The first step is to make sure all hardware and peripherals are properly working and calibrated. Once this is completed a ‘smoke’ test is performed to ensure the test board is working properly. To perform a smoke test, a dummy chip is placed onto a test board and then power is supplied. The currents from the power source to the chip are measured to ensure they are within an acceptable range. Once the test-board is proven to be functional, the PCI JTAG card on the DUT controller is checked by reading back the JTAG ID using EpiDiag commands similar to those discussed in Chapter 3. Now the entire hardware setup is connected together as the test-board is connected to the DUT controller via JTAG interface, in order to confirm read/write functionality between the DUT controller and the chip on the test-board. Finally, the register values that put the chip into transmit mode are loaded and the PLL controller is used in an attempt to lock the PLL.

Locking the PLL:

The first attempt to lock the PLL for the Radio Test Chip was not successful. After once again verifying the signal path between the DUT controller and the test chip, it was determined that a register was likely set incorrectly. During this time, the agility of this testing methodology was put to the test. We were able to simultaneously read back the PLL registers, compare to their expected value, and alter the register values. At the same time, we were able to discover a bug in the Python code that resulted in errors in about 1% of register values calculated from the Excel data. After the bug in the Python script was corrected, we were able to successfully lock the PLL. Figure 12 displays the locked signal at a frequency of 2442 Megahertz.

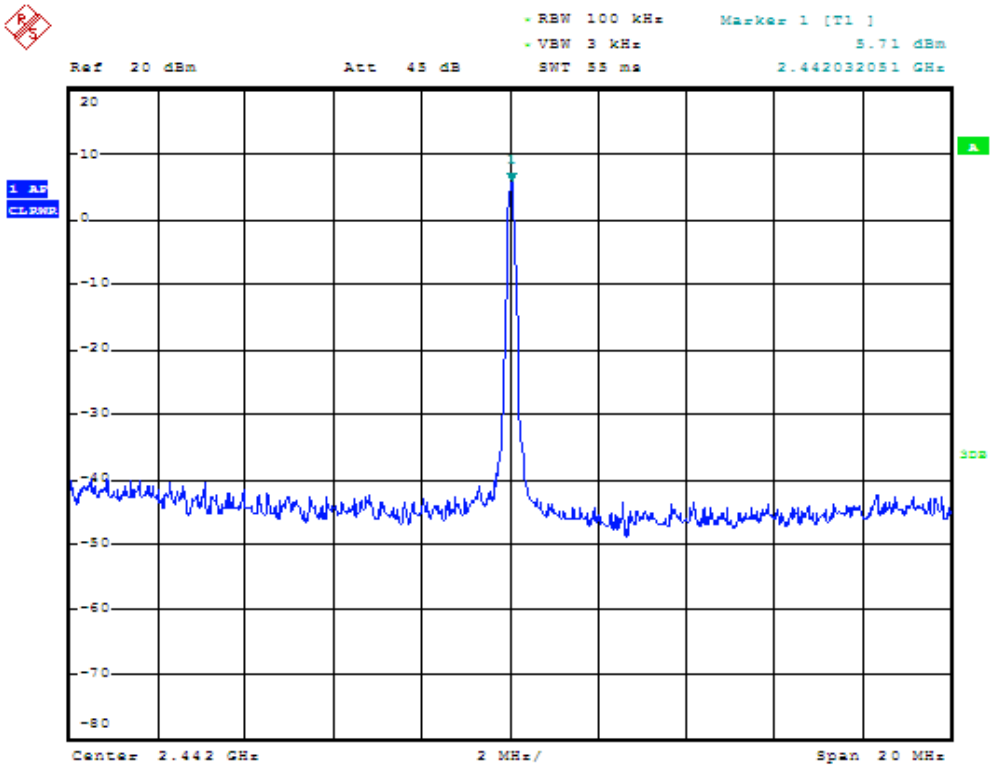


Figure 12 - A measurement of the PLL at the frequency 2.442GHz

The new test methodology proved to be a resounding success. The entire bring up process took roughly 3.5 hours to complete from the time the chip was placed on the test board to the time the PLL was successfully locked. While this is not an ideal time, this includes debugging of both the chip and an unexpected problem with the Python script. Even with the unexpected setback, the bring-up process was significantly faster than the previous methods using the LabVIEW based system.

Unexpected Problems:

There is no amount of planning that can account for every problem that may be encountered during the bring-up process. No matter how well a chip is designed and no matter how capable a test platform is to find and debug an error, there are some problems that cannot be anticipated. Certain errors are simply outside of the expertise of a development team. When this happens, outside help may be required. During the bring-up for the Radio Test Chip, one such occurrence was encountered.

When setting the JTAG registers for a group of test chips, it was noticed that the values could not be set for a group of registers that were used to set Rx gain. In the layout of the chip, these registers were physically right next to each other. After verifying the read/write capability to other registers on the chip, the chip itself was tested for ground at the input pads. The package description for the test chip is shown in Appendix E. Grounds were detected at nine separate pads on the chip. If the grounded pads were the result of a poor design, it would be disastrous for the design team. If the grounds were a result from a fabrication issue, the problem could be fixed and testing could be resumed. An X-ray of the chip was conducted in order to determine if there was a mistake in

fabrication. Figure 13 displays an X-ray of the test chip divided into regions marked 01-07.

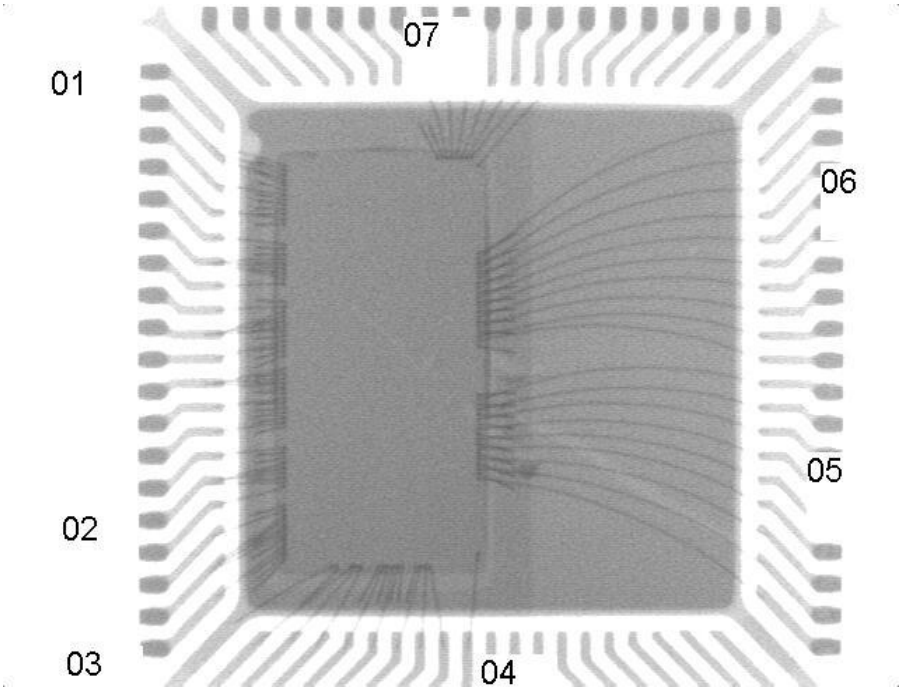


Figure 13 – X-Ray Image of the radio test chip (TOP)

Region 05 corresponds to the area of the chip with grounded pads. Figure 14 displays a side view of this region and Figure 15 displays a close-up of Region 05.

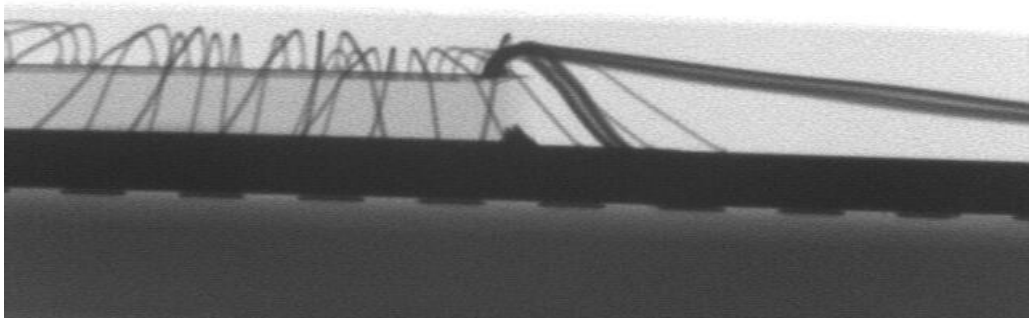


Figure 14 – X-Ray Image of the radio test chip (SIDE)

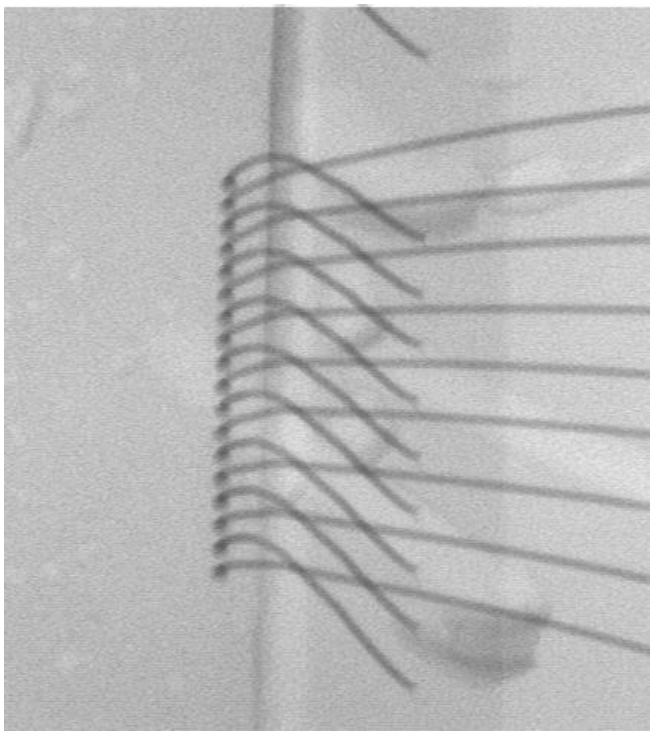


Figure 15 – Close up of defective region on radio test chip

These two views display an obvious error in the fabrication of the chip. When a chip is fabricated, it is normal to create a noticeable difference in the heights of the ground wires and the bond wires. Figure 14 displays a view of Regions 04 and 05 closely together and it is clear the bond wires and ground wires are the same height. Consequently, Figure 15 shows the ground wires and the bond wires are in fact touching each other. This is clear evidence of a problem during the fabrication of the chip, as well as an example of an unexpected error that would is not accounted for in the testing method described in this thesis.

Summary:

This Chapter focuses on the physical implementation of the Linux test environment. The first section discusses the bring-up process. This refers to the first time a test chip is powered on after fabrication. The purpose of the bring-up process is to confirm the chip is being powered correctly and the programmable registers on the chip are set to the correct binary values. From this point, it should be possible to confirm the test chip is able to perform its core functionality. For this particular study, this is confirmed when the PLL is able to lock to a specific channel. As described in this Chapter, the PLL for our test chip was successfully locked using the Linux test environment. Finally, the last portion of this Chapter focuses on some unforeseen errors that were encountered during the bring-up process and how these issues were resolved.

Chapter 6

Introduction:

The proceeding Chapters discuss the core requirements for a functional Linux based test environment. The focus of this Chapter will be the addition of test procedures that are not required for the core functionality of the test environment, but are specific to WLAN radio testing. The first section discusses how additional test procedures are added to the CCCM. The next section deals with the advantages of working in parallel. The final section of this Chapter discusses using Python to automate the writing of long testing procedures into TCL.

Additional Test Procedures:

Once the PLL reliably locks to a given channel, many other areas of the radio need to be tested in order to ensure it is functioning according to acceptable standards. All supplementary tests for IC radios are dependent on the PLL and accordingly, it must be functional before any other test procedures can be utilized. With a functional PLL, most new test procedures require little more than writing specific values to the appropriate registers.

Working in Parallel:

A large benefit to writing test procedures in TCL as opposed to LabVIEW is the ability to work in parallel with other engineers. This is primarily due to the ability to use a CVS repository. CVS allows version control of the TCL code, which means centralized versions are archived in the repository and local copies can be edited without affecting

the archived versions. Therefore, anybody contributing to testing can write and edit the code as they wish, when they wish, so long as they are careful to periodically update from the repository.

Another aspect that makes working with a TCL much more suited to parallel work is the ease of use. Many test cases are very simple state machines that only require setting a few registers to specific values. These procedures such as this are short and easy to write. Consider the procedure that will power down all power amplifiers (PA) in the radio. Doing so requires the power up bit for each power amplifier to be set to 0. These fields can be found by referring to the JTAG description discussed in Chapter 2. The code that turns off each PA is shown below.

```
proc pa_pd {} {
# Radio procedure to power down all power amplifiers.
    radio 0x5e 0x3e
    radio 0xe3 0x3e
    radio 0x79 0x00
    radio 0xfe 0x00
}
```

In the code above, each register has a PA field corresponding to least significant bit of the register. For this reason, the least significant bit of each value written to its corresponding register is 0. Table 4 describes other test procedures contributed by others for this project.

Procedure	Description
RCAL	Calibrate resistors in LPF of PLL
RCCAL	Calibrate the internal RC oscillator
CLPO	Turn on the CLPO
GPIO	Turn on/off GPIO pins
IQCAL	Calibrate phase and amplitude for the external power amplifier
TSSI_PAD	Enable TSSI

Table 4 – table of supplemental test procedures

Python Automation for Large Procedures:

Many of the additional test procedures simply require setting a distinct set of registers to an exact value. Most additional cases require very few registers to set and are easily written by hand. Other test procedures require many states and can require thousands setting thousands of register values. Larger test procedures such as these benefit from Python automation. This process once again exploits existing Excel documents that contain information regarding register and field name states. By modifying the Python code, which maps registers to their corresponding baseband values, it is easy to automate the development of TCL code for procedures that require large amounts of data. One such example is the procedure used to set the tuning registers of the PLL. Each channel in the RF bandwidth can be tuned by setting a set of registers to a value given by the tuning table. Appendix F displays the Python code used to automate this process.

Summary:

Adding additional test procedures to the CCCM can be accomplished quickly. It was important to the overall design and to adhere to the tenants of agility that test procedures would be encapsulated the CCCM. This is to say, a test procedure can be added to the CCCM and not affect the core functionality of the CCCM. This allows a large group of people to work simultaneously on the CCCM without disrupting the shared resources. TCL code for longer test procedures can be automated, in this case using Python scripting.

Chapter 7

Introduction:

Thus far, all work referenced in this thesis has been conducted using a UNIX based operating system, specifically Linux. However, many people, test engineers included, are more comfortable using other operating systems such as Microsoft Windows. In order to accommodate a broader range of users, certain aspects of the test platform were modified to work with Microsoft Windows. This Chapter discusses the challenges involved in allowing the test platform to function using the Microsoft Windows operating system. It should be noted that this setup was only implemented and tested using Windows XP with professional Service Pack 3, and has not been verified for Windows Vista, Windows 7 or any previous Windows version.

Compiling EpiDiag for Windows:

EpiDiag was coded using the C language, which allows it to be compiled directly for use with Windows. In order to compile the source code, required only an interface with the EpiDiag Makefile, and a C compiler that is compatible with Microsoft Windows. A suitable compiler can be found with Microsoft Visual C++ Express, which is an integrated development environment (IDE) used to develop and debug C, C++ and C++/CLI code for Microsoft applications. For the purposes of this project, only the C compiler is used and no actual code is developed using Microsoft Visual C++.

The EpiDiag Makefile requires an interpreter similar to that found native in Linux or UNIX. Installing Cygwin with the GNU Make package provides this functionality.

Cygwin provides a UNIX-like environment and command line interface that is natively integrated into Windows. Thus, it allows the use of UNIX tools and applications in a Windows environment. Including the GNU Make package ensures the ability to build a program using a Makefile.

Using EpiDiag with Windows:

After EpiDiag is successfully compiled as Windows executable file it can be run using the Windows operating system. Running the EpiDiag executable requires a command line interpreter. The EpiDiag executable file was run using the DOS command prompt and the Cygwin command prompt. The commands to operate EpiDiag on Windows are similar to those used for Linux. The hardware requirements for the test chip are the same as those outlined in Chapters 2 and 3. In order to interface EpiDiag with Windows, the machine requires a PC with at least one PCI slot and a Broadcom PCI JTAG card. In fact, from this point all procedures for IC radio testing are the exact same as those outlined for the Linux Test Platform. The only difference is the operating system is now Windows. The hardware requirements, however, severely limit the benefits of using this practice.

Summary:

This Chapter gives a brief overview of utilizing EpiDiag with a windows operating system. Because EpiDiag is written in C, it can be compiled as a windows executable file by using the Visual Studio Express compiler in conjunction with a Cygwin, which will handle the EpiDiag MakeFile. When EpiDiag is successfully compiled for windows, its functionality and hardware requirements are essentially the same as with Linux. Because of this, using EpiDiag with Windows is impractical.

Chapter 8

Introduction:

The focus of this Chapter is to examine the conclusions drawn from the implementation of the test platform. The first portion will consist of a quick overview of the pros and cons of the Linux based test platform. The second portion will then give a comparison to the previous test platform using LabVIEW and discuss whether or not it is a favorable alternative. Finally, there will be a quick discussion regarding the future work of the project and the influence given to other Broadcom test platforms.

The Performance of the Test Platform:

The Linux based test platform met and exceeded most expectations. It provided an agile test environment that was much more contusive toward work with multiple people. The Linux based test environment was capable of performing all tests previously conducted using the LabVIEW based test environment. It allowed for more specialized procedures to be written and is more easily automated. The tools used in the Linux test environment require less time to learn as most test procedures can be written with minimal TCL coding knowledge. Also, as discussed in Chapter 5, the bring-up process was completed with dramatic efficiency.

At the conclusion of this study, the Linux based test environment was so successful, that many of the techniques used were adopted into the testing of production chips for Broadcom's WLAN radios. In fact, he PLL controller from the CCCM and the Python Excel decoder were both directly incorporated into test procedures used for production

chips. Since the abandonment of LabVIEW based test environments, there has been a dramatic improvement in both the time it takes to create a test environment as well as the efficiency of testing for production chips.

Comparison to Pervious Platforms:

The Linux based test environment compared favorably to both the use of ATE and LabVIEW based testing environments. The following table compares the use of the Linux based test environment as opposed to ATE and LabVIEW.

Linux Test Environment Opposed to ATE	
Advantages	Disadvantages
Much more cost effective Easily Customizable Can introduce specialized procedures Fast development cycle Can be applied to characterization, functional, and parametric testing Does not require extensive programming Ideal for laboratory testing of smaller sampling size	Loss of measurement accuracy Intended ICs with low pin counts Not intended for large scale testing No current methods of data interpretation

Table 5 – Table describing advantages of Linux Test Environment opposed to ATE

Linux Test Environment Opposed to LabVIEW Test Environment	
Advantages	Disadvantages
More cost effective Easier customization Faster development cycle Does not require extensive programming or knowledge of LabVIEW Faster Bring-Up process Agile nature allows more flexibility Automation for writing long test procedures	Loss of measurement accuracy Does not include analytical tools Requires PCI slot and JTAG card

Table 6 – Comparison of Linux Test Environment and LabVIEW Test Environment

Future Work:

At the completion of this study, the Linux Test Platform had shown only a fraction of its possible capabilities. As discussed in Chapter 6, it is possible for anyone to add additional code to the CCCM without altering its overall functionality. The agile nature of the CCCM allows for any number of possible extensions and specialized testing procedures to be added to the CCCM. One important extension that will be introduced in future versions of the CCCM is data collection and storage. The function of this extension is to collect data from lab equipment specific to each test, then to store the data in a centralized database according to test type, date, and version.

Test automation is another key component that will benefit the Linux Test Platform in the future. Test automation would be most beneficial in a situation requiring a large array of test procedures to be completed on a chip, as it would greatly reduce human error and the time it would take to conduct the procedures. Test automation can be completed using tools that are readily available. A subroutine to control specific procedures within the CCCM can be added to the CCCM itself. A UNIX shell script can be used in order to launch EpiDiag and the CCCM. And finally, a UNIX cron job can be used to launch the shell script at a specific time or date. This is just one example of many possible automation techniques.

Finally, USB support and functionality will be introduced at some point in the future. This will allow for a much greater flexibility for the test platform by eliminating the hardware restriction that requires the use of a PCI JTAG card. USB functionality would allow a larger variety of machines that could be used to control the DUT, as there would no longer be a need for a PCI slot. During the study an effort was made to incorporate USB functionality, but it was halted by hardware restrictions.

Conclusion:

The development of an agile testing environment for IC radios required the utilization of many engineering disciplines. This includes software programming, IC design, IC testing, circuit design, and system integration. Much attention is given to the capabilities and resources of the Linux operating system and the examination of commercial and public software. The methods designed for this project were architected and implemented specifically for radio test chips and is now in use in industry.

The Linux based test environment was easily able to complete all the tasks that would normally be completed using programs such as LabVIEW. The testing methods described in this study demonstrated a number of improvements over more conventional testing methods. Changes to test procedures could be completed quickly, the development cycle was much faster, and problems and bugs in the test code were dealt with in a much more timely fashion. This study proved that by using cost effective and relatively simple code, testing for WLAN IC radios could be robust, agile, and comprehensive.

References:

[1] JTAG Tutorial [Online]. Available: http://www.corelis.com/education/JTAG_Tutorial.htm

[2] Jan Decaluwe (2004) MyHDL: a Python-Based Hardware Description Language [Online]. Available: http://www.Linuxjournal.com/article/7542

[3] Chris Withers & John Machin (2009) Working with Excel Files in Python

[4] Mike Lutz & David Ascher (2004) Learning Python

[5] Brent B. Welch (1997) Practical Programming in TCL and Tk Second Edition

[6] Dan H. Wolaver (1991) Phase-Locked Loop Circuit Design

[7] John D. McGregor & David A. Sykes (2001) A Practical Guide to Testing Object Oriented Software

[8] Steve Carison, "ATE Struggles To Keep Pace With VLSI," *EE Times*, 13 December 2001 [Online]; available from http://www.eetimes.com: accessed June 2009.

[9] Chad Fasca and Dylan McGrath, "Virtual Test Comes of Age," *Evaluating Engineering* July 1998 [Online]; available from http: / /www.virtualtestcom: accessed June 2009.

[10] http://focus.ti.com/lit/ds/symlink/sn74ahct125-q1.pdf

[11] "LabVIEW from Wikipedia, the Free Encyclopedia", Wikipidea www.wikipedia.com

[12]

Appendix A: tc_reg_gen.py

```
#!/usr/bin/env Python

import sys, string, os
sys.path.append("/projects/radio/gallery/bin")
sys.path.append("/projects/radio/lib/Python")
from xlrd import open_workbook
from myhdl import intbv
from myhdl import bin
import re

#-----------------------------------------------------
# Declare MS Excel File Variables    ****************
#-----------------------------------------------------
#*****************************************************
#
def rxtx_dictionary(xls):
    """
    Create an Rx or Tx dictionary to be compared to the full
    register list
    """
    print 'Loading', xls, 'workbook\n'

    rxtx_keys = []
    rxtx_hvals = []
    rxtx_bvals = []

    sh_rxtx = wb.sheet_by_name(xls)
#sh_rxtx = wb.sheet_by_name(u'2057_jtag')


    for rownum in range(sh_rxtx.nrows):
        rxtx_keys.append(sh_rxtx.cell_value(rownum, 0))
        rxtx_hvals.append(sh_rxtx.cell_value(rownum, 1))
        rxtx_bvals.append(sh_rxtx.cell_value(rownum, 2))

    return dict(zip(rxtx_keys, rxtx_bvals))


#-----------------------------------------------------
# Declare MS Excel File Variables    ****************
#-----------------------------------------------------
#*****************************************************
#
def hex2int(in_val):
    """
    convert hex to 16 bit integers
    """
    if isinstance(in_val, (float)):
        out = int(str(int(in_val)),16)
    elif isinstance(in_val, (unicode)):
        out = int(in_val,16)
```

```
    else:
        print "error"
    return out


#-----------------------------------------------------
# Declare MS Excel File Variables    ****************
#-----------------------------------------------------
#*****************************************************
def int2binary(n):
    """
    Convert integer into binary. This is for reset value only
    """
    bStr = ''
    if n < 0: raise ValueError, "Must be a positive integer"
#     if n == 0: return '0'
#     while n > 0:
#         bStr = str(n % 2) + bStr
#         n = n >> 1
    bStr = bin(n)
    return bStr

#-----------------------------------------------------
# Declare MS Excel File Variables    ****************
#-----------------------------------------------------
#*****************************************************

def full_TCL():
    """
    Create file that will produce TCL procedures for both Rx and Tx
modes
    """
    workbook = []
    for i in wb.sheet_names():
        workbook.append(i)

#    file2write = raw_input('Please specify output file:')
    file2write = sys.argv[2]
    tclfile = open(file2write,'w')
    tclfile.write('#2057 register write\n')

    for i in workbook[1:]:
#        print 'Loading', i,'file'
        txrx_bv = rxtx_dictionary(i)
        tclfile.write('# ')
        tclfile.write(i)
        tclfile.write('\n')
        tclfile.write('proc ')
        tclfile.write(i)
        tclfile.write(' { } {\n')

        for reg in registers:
            tclfile.write('set reg(RF_')
            tclfile.write(reg)
```

```
            tclfile.write(') ')
            tcl_bv = lookup_bv(reg, txrx_bv)
            tclfile.write(tcl_bv)
            tclfile.write('\n')

        tclfile.write('}\n')
        print 'finished',i,'\n'
    tclfile.close

#-------------------------------------------------
#          ****************************************
#-------------------------------------------------
#************************************************
#
def lookup_bv(addr, bitvals):
    """
    Calculate the bitvalue for a given address. Return in hex
    """
    bv = ''
    index = 0

    for x in registers[addr]['field_name']:


        if bitvals.has_key(x):
            if x.find('<') > 0:
                span = x.split('<')[1].replace('>','').split(':')
                bit_length = int(span[0])-int(span[1])+1
                #bit_length

                index += 1
                tmp_str = str(int(bitvals[x]))
                tmp_intb2 = int(tmp_str,2)
                tmp2 = bin(tmp_intb2,bit_length)
                bv += str(tmp2)
            else:
                index += 1
                bv += str(int(bitvals[x]))
        else:
            print addr, registers[addr]['reset_value'][index]
            bv = int(registers[addr]['reset_value'][index])
            bv = int2binary(bv)
            index += 1

    hexbv = hex(intbv(bv,0,256))
    hexbv = hexbv.strip('L')
    return hexbv


#-------------------------------------------------
#          ****************************************
#-------------------------------------------------
#************************************************
#
```

```
def create_keys():
    """
    create keys for Python dictionaries
    """
    full_list = sh_final.row_values(0)
    reg_data = full_list[2:]
    return reg_data


#-----------------------------------------------------
# Create Dictionary for All registers    ************
# using 2057_final.xls       ************************
#-----------------------------------------------------
#*****************************************************
#

# register_settings_xls = raw_input('Please specify MS Excel file:
')
register_settings_xls = sys.argv[1]
print 'Loading', register_settings_xls

#-----------------------------------------------------
# Declare MS Excel File Variables    ****************
#-----------------------------------------------------
#*****************************************************
#
wb = open_workbook(register_settings_xls)
wb.sheet_names()

sh_final = wb.sheet_by_name(u'JTAG_final')


#-----------------------------------------------------
# Declare MS Excel File Variables    ****************
#-----------------------------------------------------
#*****************************************************
#
registers = {}
bitvals = {}
addresses = {}
register_keys = []
reg_values_keys = []
key_vals = 0


#load in the rx/tx commands and binary values into dictionary bitval
reg_values_keys = create_keys()

#this is to fill the register names and creat a dictionary for each
register.
for rownum in range(1,sh_final.nrows):
    if sh_final.cell_value(rownum, 0) == 'PORTS':
        break
    elif sh_final.cell_value(rownum, 1) == 'idcode':
```

```
        continue

    elif sh_final.cell_value(rownum, 0) or
sh_final.cell_value(rownum,0) == 0:
        # skip over the idcode
        reg_name = sh_final.row_values(rownum)[1]
        addr_name = hex2int(sh_final.row_values(rownum)[0])

        key_vals = sh_final.row_values(rownum) #gets regs and
addresses
        key_vals = key_vals[2:] #eliminate whitespace at beginning
of row

        tmp = key_vals #create list equal to tmp
        # initialize the 2D list
        for x in range(len(reg_values_keys)):
            tmp[x] = []

            registers[reg_name] = dict(zip(reg_values_keys, tmp))
            addresses[addr_name] = dict(zip(reg_values_keys, tmp))
    else:
        if sh_final.cell_value(rownum, 2)[0:6] != 'idcode':
            key_vals = sh_final.row_values(rownum) #gets regs and
addresses
            key_vals = key_vals[2:] #eliminate whitespace at
beginning of row
            mytmp = dict(zip(reg_values_keys, key_vals))
            for cell in mytmp.keys():
                registers[reg_name][cell].append(mytmp[cell])

#-----------------------------------------------------
# Main loop of script    ***************************
#-----------------------------------------------------
#*****************************************************
#
#options menu

#full_TCL()
```

Appendix B: CCCM Initialization and Utilities

```
# Initialize JTAG and echo radio_id.
set addr_len 10
set data_len 8
set id_len 32

s rgphy/tc2064_jtag.tcl

#jtag config clkd 14
jtag config clkd 30
```

```
jtag config ir $addr_len
jtag config dr $id_len
set id [jtag readreg 0x01]
puts "Chip ID: $id"
jtag config dr $data_len
global double
set double 1

proc hexpr {args} {
# Utility function to converts argument into hex values
      set r [uplevel expr $args]
      format 0x%x [expr $r & 0xffffffff]
}

proc usdelay { {usec 16} } {
# Utility function to delay, like a sleep command.
      global reg def

      # Use "clock" function to time delay (one clock click is
1usec)
      # (Note that while the "clock" function is fairly accurate,
      #  TCL overhead spoils overall accuracy, limiting the
minimum delay.
      #  On a 2.0GHz cpu, the overhead is about 25us to 30us.
12/11/05)
      # These timings can be measured via "checktimer.tcl"
      # ------------------------------------------------------
---------
      set t0 [clock clicks]
      for {set ix 0} {$ix < 100000} {incr ix} {
            set t1 [clock clicks]
            if { $t1 < $t0 } { set t0 $t1 }
            set ntics [expr $t1 - $t0 ]
            if { $ntics > $usec } { break }
      }
      puts "usdelay($ix)($usec): Elapsed time $ntics tics"
}

proc write_radio { addr data } {
# Radio procedure - allows you to write directly to a jtag
register
#*****************************************************
# Argument list
#*****************************************************
# addr: hex address to write to
# data: data stored at specified address.
    jtag writereg $addr $data
}

proc read_radio { addr } {
# Radio procedure - allows you to read directly from a jtag
register
```

```
#*****************************************************
# Argument list
#*****************************************************
# addr: the address data is read from. this address is
# offset by a factor of 0x100 to designate it as a read
# address
    if {$addr != 0x01} {
        set addr [expr $addr + 0x200]
        set readval [jtag readreg $addr]
     return $readval
    } else {
        set data_len 32
     jtag config dr $data_len
     # jtag config
        set readval [jtag readreg $addr]
     # puts "tp01 $readval"
        set data_len 8
     jtag config dr $data_len
     # jtag config
     return $readval
    }
}

proc radio {args} {
# Radio procedure to both read and write to register based on the
number
# of command line arguments
#*****************************************************
# Argument list
#*****************************************************
# args: creates a list of command line arguements
    set addr [lindex $args 0]
    switch [llength $args] {
        1 {
            read_radio $addr
        }
        2 {
            set val [lindex $args 1]
            write_radio $addr $val
        }
    }
}

proc bin_table { inval } {
# Utility function to convert from bin back to hex.
#*****************************************************
# Argument list
#*****************************************************
# inval: the 4 bit binary number that will be
# returned as hex 0 - f
    switch -exact -- $inval {
        0000 { set out_val 0 }
```

```
        0001 { set out_val 1 }
        0010 { set out_val 2 }
        0011 { set out_val 3 }
        0100 { set out_val 4 }
        0101 { set out_val 5 }
        0110 { set out_val 6 }
        0111 { set out_val 7 }
        1000 { set out_val 8 }
        1001 { set out_val 9 }
        1010 { set out_val a }
        1011 { set out_val b }
        1100 { set out_val c }
        1101 { set out_val d }
        1110 { set out_val e }
        1111 { set out_val f }
        default {puts "binary data cannot be calculated";
            puts "";
            error }
    }
    return $out_val
}


proc doubler { doubler_val } {
    global double

    set double $doubler_val
    set doubler_init [radio 0x9f]
    set doubler_reg_val [format_reg $doubler_init $doubler_val 2
2]
    write_radio 0x9f $doubler_reg_val
}


proc force_caps { val } {
    radio 0x4e [expr 0x02 | (0x01 & $val)]
    radio 0x4c [expr 0xff & $val]
    radio 0x56 0x0

}

proc unforce_caps { } {
    radio 0x4e 0x0
    radio 0x4c 0x0
    radio 0x56 0x0

}

proc reg_dump {{filename "none"} {permissions "w"}} {
# Radio procedure to dump the contents of all 2057 registers into
a log file
# that is specified by the user.
```

```
#*****************************************************
# Argument list
#*****************************************************
# filename: User defined file name
# permissions: A set value to allow read/write to logfile
#*****************************************************

    set addr_high 0xf0

    set logfile [open $filename $permissions]

    puts $logfile "# tc2064_regdump"
    puts $logfile "proc tc2064_regdump"
    puts $logfile "{"

    for {set addr 0} {$addr < $addr_high} {incr addr 1} {

        set current_value [radio $addr]
        # set hex_addr $addr
        set hex_addr [hexpr $addr]


        puts $logfile "\tradio 0x$addr 0x$current_value"
        set current_value [hexpr [radio $addr]]
        set hex_addr [hexpr $addr]
        puts $logfile "radio $hex_addr $current_value"


    }
    puts $logfile "}"
    close $logfile
}
```

Appendix C: CCCM Format Register Function

```
proc format_reg { initial_val insert_val insert_start insert_stop
} {
# Utillity function to change parts of a register. This procedure
will
# take an initial 8-bit hex number and an insertion value. The
these
# values will be converted to binary and the insertion value will
# overwrite selected bits in the initial, designated by
insert_start and
# insert_stop.
#*****************************************************
```

```
# Argument list
#*************************************************
# initial_value: the starting value of the register
# insert_value: the value to be inserted
# insert_start: the lower bound where instert_val is placed into
# initial_val
# insert_stop: the upper_bound where insert_val is placed into
initial_val
#*************************************************
    set init_hex_str $initial_val
    set ins_hex_str $insert_val
    set insert_start_loc $insert_start
    set insert_stop_loc $insert_stop
    #puts "value inserted $insert_val"

    #FIXME must convert hex string to int, then back to hex
    #find a way around this
    #convert init val to int
    set init_val [expr int($init_hex_str)]

    #convert back to hex
    set init_hex_val [format %x $init_val]
    #set init_hex_val [hexpr $init_val]
    if { $init_val > 15 } {


        #   puts "initial hex val $init_hex_val"
        #convert to binary number
        binary scan [binary format H2 $init_hex_val] B8
init_bin_val
    } else {
        set init_msb 0000
        binary scan [binary format H2 $init_hex_val] B4
init_bin_val
        set init_bin_val [concat $init_msb$init_bin_val]
    }

    #FIXME Is a repeat of convert to bin string, but for
insert_val
    #convert init val to int
    set ins_val [expr int($ins_hex_str)]
    #convert back to hex
    set ins_hex_val [format %x $ins_val]

    #NOTE: cannot use hexpr here
    #set ins_hex_val [hexpr $ins_val]

    #convert to binary number
    if { $ins_val > 15 } {

        #convert to binary number
```

```
        binary scan [binary format H2 $ins_hex_val] B8
ins_bin_val
    } else {
        set ins_msb 0000
        binary scan [binary format H2 $ins_hex_val] B4
ins_bin_val
        set ins_bin_val [concat $ins_msb$ins_bin_val]
    }

    set ins_size [expr $insert_start - $insert_stop]

    #get only relevant data
    set get_ins [expr 7 - $ins_size]
    set ins_bin_val [string range $ins_bin_val $get_ins 7]


    ######################################################
    # Slice up the initial binary value
    ######################################################

    #Tcl is big endian, so switch start and stop
    if {$insert_start == 7 } {
        if {$insert_stop == 0 } {
            set insert_stop 7
            set new_val $ins_bin_val
        } else {
            set insert_stop [expr 7-$insert_stop + 1]
            set LSBs [string range $init_bin_val $insert_stop 7]
            set new_val [concat $ins_bin_val$LSBs]
        }
    } else {
        set insert_start [expr 7-$insert_start - 1]
        set MSBs [string range $init_bin_val 0 $insert_start]



        if {$insert_stop == 0 } {
            set insert_stop 7
            set new_val [concat $MSBs$ins_bin_val]
        } else {
            set insert_stop [expr 7-$insert_stop + 1]
            set LSBs [string range $init_bin_val $insert_stop 7]
            set new_val [concat $MSBs$ins_bin_val$LSBs]
        }
    }


    ######################################################
    # Convert cut in half 8-bit binary string
    # Convert to hex using table
    set p1_new_val [string range $new_val 0 3]
    set p2_new_val [string range $new_val 4 7]
```

```
    set p1_new_val [bin_table $p1_new_val]
    set p2_new_val [bin_table $p2_new_val]

    set new_val [concat 0x$p1_new_val$p2_new_val]

    return $new_val
}
```

Appendix D: CCCM RF_PLL Function

```
proc rf_pll { xtal channel } {
# Radio procedure is designed to characterize the PLL
# for TC2057 by writing directly to the registers through
# JTAG wire. It will return the band to the init file.
# information for jtag writing can be found in
# synth_xtal_loggen.xls in rgphy directory
#*************************************************
# Argument list
#*************************************************
# xtal: the reference crystal freq utilized. Generally 20 or 40.
# channel: desired frequency channel for the pll
#*************************************************
    global band
    global double
# TODO: remove this hack which turns on synthesizer
    radio 0x44 0x1c
#_______________________________________________

    set loop_bw_ref 270.0
    set loop_bw_des 260.0
    set xtal_freq $xtal
    #set double [doubler]
    #set doubler_val [doubler]
    if { $double == 0 } {
        set PFD_ref_freq [expr $xtal * 2.0]
    } else {
        set PFD_ref_freq $xtal
    }
    puts "PFD_ref_freq $PFD_ref_freq"

    if { $channel > 2400 && $channel < 2500} {
        set band g
    } elseif { $channel > 4700 && $channel < 6000 } {
        set band a
    }

    if {$band == "a"} {
```

```
        set Vco_freq [expr $channel*2.0/3]
    } else {
        set Vco_freq [expr $channel*1.5]
    }


    puts "We have channel $channel"
    puts "We are in Band $band"
    puts "We have FVco $Vco_freq"

    #divider
    set pll_sg_val_int [expr floor($Vco_freq/$PFD_ref_freq)]
    set pll_sg_val_int [expr round($pll_sg_val_int)]
    puts "div_int $pll_sg_val_int"


    #*************************************************
    # Loop Filter

    set Ndiv_ratio [expr $pll_sg_val_int/90.0]
    puts "Ndiv_ratio $Ndiv_ratio"

    set le_Vco_freq 3000.0
    set le_KVco 27.0
    set he_Vco_freq 4200.0
    set he_KVco 68.0
    set KVco_ref 30.0
    set KVco_des [expr ($he_KVco - $le_KVco)/($he_Vco_freq -
$le_Vco_freq)*($Vco_freq - $le_Vco_freq) + $le_KVco]

    set KVco_ratio [expr $KVco_des/$KVco_ref]


    set loop_bw_ratio [expr $loop_bw_des/$loop_bw_ref]
    puts "loop_bw_ratio $loop_bw_ratio"

    # PLL_lf_r1 (ohm) reg042, wrf_i_jtag_pll_lf_3<7:3>)
    set lf_r1_ref 2640.0
    set lf_r1_des 8000.0

    set lf_r1_init [radio 0x42]
    set lf_r1_int_val [expr round(($lf_r1_des - 680.0)/490.0)]
    puts "lf_r1_int_val $lf_r1_int_val"

    set lf_r1_val [hexpr $lf_r1_int_val]

    set lf_r1 [format_reg $lf_r1_init $lf_r1_val 7 3]
    write_radio 0x42 $lf_r1

    set lf_r1_exact [expr 680.0 + $lf_r1_int_val * 490.0]
    set lf_r1_ratio [expr $lf_r1_exact/$lf_r1_ref]
    puts "lf_r1_ratio $lf_r1_ratio"
```

```
    # PLL_lf_r2 (ohm), (42, wrf_i_jtag_pll_lf_3<2:0>,43,
wrf_i_jtag_pll_lf_4<6:5>)
    set lf_r2_ref 1660
    set lf_r2_des [expr $lf_r1_ratio * $lf_r2_ref]

    set lf_r2_val [expr round(($lf_r2_des - 680)/490)]
    puts "lf_r2_val $lf_r2_val"
    set lf_r2_val [hexpr $lf_r2_val]

    set lf_r2_msb_init [radio 0x42]
    set lf_r2_lsb_init [radio 0x43]
    set lf_r2_msb_val [hexpr (0x1c & $lf_r2_val)>>2]
    set lf_r2_lsb_val [hexpr (0x3 & $lf_r2_val)]

    set lf_r2_msb [format_reg $lf_r2_msb_init $lf_r2_msb_val 2 0]
    set lf_r2_lsb [format_reg $lf_r2_lsb_init $lf_r2_lsb_val 6 5]

    write_radio 0x42 $lf_r2_msb
    write_radio 0x43 $lf_r2_lsb


    # PLL_lf_r3 (ohm) (43, wrf_i_jtag_pll_lf_4<4:0>)
    set lf_r3_ref 1660
    set lf_r3_des [expr $lf_r1_ratio * $lf_r3_ref]
    set lf_r3_val [expr round(($lf_r3_des - 680)/490)]
    puts "lf_r3_val $lf_r3_val"
    set lf_r3_val [hexpr $lf_r3_val]


    set lf_r3_init [radio 0x43]
    set lf_r3 [format_reg $lf_r3_init $lf_r3_val 4 0]

    write_radio 0x43 $lf_r3

    # PLL_lf_c1 (pF) (40, wrf_i_jtag_pll_lf_1<7:4>)
    set lf_c1_ref 1046.5
    set lf_c1_des [expr $lf_c1_ref/$lf_r1_ratio/$loop_bw_ratio]

    set lf_c1_val [expr round(($lf_c1_des - 170)/12.8)]
    puts "lf_c1_val $lf_c1_val"
    set lf_c1_val [hexpr $lf_c1_val]

    set lf_c1_init [radio 0x40]
    set lf_c1 [format_reg $lf_c1_init $lf_c1_val 7 4]

    write_radio 0x40 $lf_c1


    # PLL_lf_c2 (pF) (40, wrf_i_jtag_pll_lf_1<3:0>)
    set lf_c2_ref 61.7
    set lf_c2_des [expr $lf_c2_ref/$lf_r1_ratio/$loop_bw_ratio]
```

```
    set lf_c2_val [expr round(($lf_c2_des - 13)/0.73)]
    puts "lf_c2_val $lf_c2_val"

    set lf_c2_val [hexpr $lf_c2_val]

    set lf_c2_init [radio 0x40]
    set lf_c2 [format_reg $lf_c2_init $lf_c2_val 3 0]

    write_radio 0x40 $lf_c2

    # PLL_lf_c3 (pF) (41, wrf_i_jtag_pll_lf_2<7:4>)
    set lf_c3_ref 27
    set lf_c3_des [expr $lf_c3_ref/$lf_r1_ratio/$loop_bw_ratio]

    set lf_c3_val [expr round(($lf_c3_des - 5)/0.46)]
    puts "lf_c3_val $lf_c3_val"

    set lf_c3_val [hexpr $lf_c3_val]

    set lf_c3_init [radio 0x41]
    set lf_c3 [format_reg $lf_c3_init $lf_c3_val 7 4]

    write_radio 0x41 $lf_c3

    # PLL_lf_c4 (pF) (41, wrf_i_jtag_pll_lf_2<3:0>)
    set lf_c4_ref 26.4
    set lf_c4_des [expr $lf_c4_ref/$lf_r1_ratio/$loop_bw_ratio]

    set lf_c4_val [expr round(($lf_c4_des - 4.2)/0.46)]
    puts "lf_c4_val $lf_c4_val"


    set lf_c4_val [hexpr $lf_c4_val]

    set lf_c4_init [radio 0x41]
    set lf_c4 [format_reg $lf_c4_init $lf_c4_val 3 0]

    write_radio 0x41 $lf_c4


    #"PLL_cp_current (uA) (3C, wrf_i_jtag_pll_cp_2<5:0>)"
      set cp_curr_init [radio 0x3c]
    set cp_curr_ref 1250
        set cp_curr_des [expr
$cp_curr_ref*$loop_bw_ratio*$Ndiv_ratio/$KVco_ratio/$lf_r1_ratio]

    #"PLL_Kpd_scale2 (3C, wrf_i_jtag_pll_cp_2<6>)"
      if { $cp_curr_des > 750 } {
        set Kpd_scale2_val 1
    } else {
        set Kpd_scale2_val 0
```

```
    }
    puts "Kpd_scale2_val $Kpd_scale2_val"
    set cp_curr_val [expr
round($cp_curr_des/12.5/($Kpd_scale2_val + 1)-4)]
    puts "cp_curr_val $cp_curr_val"
    set cp_curr_val [hexpr $cp_curr_val]

    set Kpd_scale2 [format_reg $cp_curr_init $Kpd_scale2_val 6 6]
    write_radio 0x3C $Kpd_scale2

    set cp_curr_init [radio 0x3c]
    set cp_curr [format_reg $cp_curr_init $cp_curr_val 5 0]
    write_radio 0x3C $cp_curr

    set cp_curr_cal [expr 12.5*($cp_curr_val +
4)*($Kpd_scale2_val+1)]
    puts "cp_curr_cal $cp_curr_cal"

    #"PLL_ioff_scale2 (3D, wrf_i_jtag_pll_cp_3<6>)"
    set ioff_scale2 0

    #"PLL_offset_current (uA) (3D, wrf_i_jtag_pll_cp_3<5:0>"
      #FIXME: may need tweaking
    set offset_des [expr 4*$cp_curr_cal*$PFD_ref_freq/$Vco_freq]
    puts "offset_des $offset_des"

    set offset_val [expr ceil($offset_des/1.5-2)]
    puts "offset_val $offset_val"
    if {$offset_val < 0 } {
        set offset_val 0
    }
    set offset_val [expr int($offset_val)]

    set offset_val [hexpr $offset_val]
    set offset_init [radio 0x3d]

    set offset_curr [format_reg $offset_init $offset_val 5 0]
    write_radio 0x3d $offset_curr


    #"PLL_cal_xt_endiv (57, wrf_i_jtag_pll_xtal_1<5>)"
    set cal_xt_endiv_init [radio 0x57]
    if {$xtal_freq > 26 } {
        set cal_xt_endiv_val 1
    } else {
        set cal_xt_endiv_val 0
    }

    set cal_xt_endiv [format_reg $cal_xt_endiv_init
$cal_xt_endiv_val 5 5]
    puts "xt_endiv $cal_xt_endiv"
    write_radio 0x57 $cal_xt_endiv
```

```
    #"PLL_cal_xt_sdiv (57, wrf_i_jtag_pll_xtal_1<4>"
    set cal_xt_sdiv_init [radio 0x57]

    if {$xtal_freq > 52 } {
        set cal_xt_sdiv_val 1
    } else {
        set cal_xt_sdiv_val 0
    }

    set cal_xt_sdiv [format_reg $cal_xt_sdiv_init
$cal_xt_sdiv_val 4 4]
    puts "xt_sdiv $cal_xt_sdiv"
    write_radio 0x57 $cal_xt_sdiv



#*************************************************************
******

    # calculate cal xtal freq
    if {$cal_xt_endiv_val == 0} {
        set cal_xtal_freq $xtal_freq
    } elseif { $cal_xt_sdiv_val == 0 } {
        set cal_xtal_freq $[expr $xtal_freq/2]
    } else {
        set cal_xtal_freq $[expr $xtal_freq/4]
    }
    #"PLL_sel_short (4a, wrf_i_jtag_pll_vco_1<1>"
    set sel_short_init [radio 0x4a]

    if {$PFD_ref_freq > 45 } {
        set sel_short_val 1
    } else {
        set sel_short_val 0
    }

    set sel_short [format_reg $sel_short_init $sel_short_val 1 1]
    write_radio 0x4a $sel_short

    #-----------------------------------------------------
    #-----------------------------------------------------


    #*****************************************************
    # VCO Cal
    #*****************************************************
    # Reference freq
    set pll_enable_TO_wanted 1.6
    #PLL_delayBeforeOpenLoop (4f, wrf_i_jtag_pll_vcocal_3<7:0>
    set pll_delay_BOL_wanted 5
```

```
    set pll_delay_BOL_val [expr
round($pll_delay_BOL_wanted/$pll_enable_TO_wanted -1 )]
    #puts "pll_delay_BOL_val $pll_delay_BOL_val"

    set pll_delay_BOL_val [hexpr $pll_delay_BOL_val]

    #set pll_delay_BOL_init [radio 0x4f]
    write_radio 0x4f $pll_delay_BOL_val


    #PLL_enableTimeOut, set Tunit x 2
    # (52, wrf_i_jtag_pll_vcocal_6<2:0>) (53,
wrf_i_jtag_pll_vcocal<6:5>
    set enable_TO_init [radio 0x53]

    set pll_enable_TO_val [expr
round($cal_xtal_freq*$pll_enable_TO_wanted/2 - 1 )]
    #puts "pll_enable_TO_val $pll_enable_TO_val"
    set pll_enable_TO_int_val [expr int($pll_enable_TO_val)]
    set pll_enable_TO_val [hexpr $pll_enable_TO_int_val]
    set enable_TO [format_reg $enable_TO_init $pll_enable_TO_val
6 5]

    write_radio 0x53 enable_TO

    #PLL_cal_ref_timeout (51, wrf_i_jtag_pll_vcocal_5<7:0>
    set cal_ref_TO_wanted 8
    set cal_ref_TO_val [expr
$cal_xtal_freq.0*$cal_ref_TO_wanted.0/($pll_enable_TO_int_val+1)-
1]
    set cal_ref_TO_int_val [expr round($cal_ref_TO_val)]
    #puts "cal_ref_TO_val $cal_ref_TO_int_val"
    set cal_ref_TO [hexpr $cal_ref_TO_int_val]


    write_radio 0x51 $cal_ref_TO

    #PLL_cal_set_count
    #(53, wrf_i_jtag_pll_vcocal_7<3:0>) (54,
wrf_i_jtag_pll_vcocal_8<7:0>
    #puts "cal_xtal_freq $cal_xtal_freq"
    set cal_ref_TO_cal [expr ($cal_ref_TO_int_val +
1.0)/$cal_xtal_freq*($pll_enable_TO_int_val +1)]
    #puts "cal_ref_TO_cal $cal_ref_TO_cal"
    set cal_set_count_val [expr round($cal_ref_TO_cal *
$Vco_freq/16-1)]
    #puts "cal_set_count_val $cal_set_count_val"
    set cal_set_count_val [hexpr $cal_set_count_val]
    set cal_set_count_MSB [expr ($cal_set_count_val & 0xff00)>>8]
    set cal_set_count_LSB [expr $cal_set_count_val & 0xff]

    set cal_set_count_reg53 [radio 0x53]
```

```
    set cal_set_count_MSB [format_reg $cal_set_count_reg53
$cal_set_count_MSB 3 0]
    set cal_set_count_MSB [format_reg $cal_set_count_MSB 0x01 4
4]

    write_radio 0x53 $cal_set_count_MSB
    write_radio 0x54 $cal_set_count_LSB

    #divider, integer bits
    #(45, wrf_i_jtag_pll_sg_1<4:0>) (46, wrf_i_jtag_pll_sg_2<7:4>
    set pll_sg_msb_init [radio 0x45]
    set pll_sg_lsb_init [radio 0x46]



    set pll_sg_val [hexpr $pll_sg_val_int]


    set pll_sg_msb [expr ($pll_sg_val & 0xff)>>4]
    set pll_sg_lsb [expr $pll_sg_val & 0xf]

    set pll_sg_msb [format_reg $pll_sg_msb_init $pll_sg_msb 4 0]
    set pll_sg_lsb [format_reg $pll_sg_lsb_init $pll_sg_lsb 7 4]

    write_radio 0x45 $pll_sg_msb
    write_radio 0x46 $pll_sg_lsb


    #divider, frac bits
    # 46, wrf_i_jtag_pll_sg_2<3:0>
    # 47, wrf_i_jtag_pll_sg_3<7:0>
    # 48, wrf_i_jtag_pll_sg_4<7:0>
    set sg_frac_2_init [radio 0x46]

    set sg_frac_val_a [expr ($Vco_freq/$PFD_ref_freq -
$pll_sg_val_int)]

    set sg_frac_val_b [expr abs($sg_frac_val_a)]

    set sg_frac_val_c [expr pow(2, -20)]

    set sg_frac_val [expr round($sg_frac_val_b/$sg_frac_val_c)]
    puts "div_frac $sg_frac_val"

    set sg_frac_val [hexpr $sg_frac_val]

    set sg_frac_val_2 [expr ($sg_frac_val & 0xff0000)>>16]

    set sg_frac_val_3 [expr ($sg_frac_val & 0x0ff00)>>8]
    puts "fractal hex $sg_frac_val"

    set sg_frac_val_4 [expr ($sg_frac_val & 0x000ff)]
```

```
    set sg_frac_val_2 [format_reg $sg_frac_2_init $sg_frac_val_2
3 0]

    write_radio 0x46 $sg_frac_val_2
    write_radio 0x47 $sg_frac_val_3
    write_radio 0x48 $sg_frac_val_4



    #*************************************************


    vco_cal
}

#proc pll_check_lock {} {
    # Radio proceedure to check if PLL is locked.
    #set val [radio 0x33]
    #if {$val == 0x05} {
    #    puts "PLL Locked."
    #} elseif {$val == 0x02} {
    #    puts "PLL *NOT* Locked."
    #} else {
    #    puts "PLL reg 0x33 returns $val, an unknown value"
    #}
#}

proc vco_cal {} {
# Radio procedure to trigger a VCO calibration.

    #correction made to VCO LDO register.
    radio 0x9b 0x36

    radio 0x56 0x0

    radio 0x56 0x2

    radio 0x56 0x3

    radio 0x49 0xc

    radio 0x49 0xd
 #   set val [radio 0x2b]
 #   radio 0x2b [hexpr $val & 0xfe]

 #   set val [radio 0x2e]
 #   radio 0x2e [hexpr $val & 0xfb]

 #   set val [radio 0x2e]
```

```
 #   radio 0x2e [hexpr $val | 0x04]

 #   set val [radio 0x2b]
 #   radio 0x2b [hexpr $val | 0x01]
}
```