MQP-CS-DXF-NV22

NVIDIA Regression Testing Utilities

A **Major Qualifying Project** Report

Submitted to the Faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science.

Submitted by:

Shaoming Feng

Tory Jaskoviak

March 6, 2012

Some confidential material has been removed from

this report at the request of our sponsor.

Abstract

Upgrades to the Android system on a Tegra chip may cause unintended hardware issues, including complications with clock generator and multiplexer components. Regression testing is required to ensure these impediments do not occur. Currently, manual testing of such components is time consuming and complex. We were tasked with developing utilities to automate this process. At the conclusion of our project, we were able to develop two tools that significantly reduced time spent on regression testing. The clock tree sanity tool can efficiently compare clock trees and return issues with clock components. Similarly, the pin multiplexer tool can read, write, and debug register values with great ease of use.

Acknowledgements

The team would like to thank the following people for their invaluable help with the project by providing information and guidance.

Professor David Finkel

Karthik Ramakrishnan

Daniel Solomon

Wen Yi

Contents

List of Figures

Executive Summary

The Tegra family is a series of chips NVIDIA developed for mobile devices. Currently, the Tegra 3 is predominantly used in Android devices. Due to strict hardware conditions, certain Tegra components must run within certain tolerance values. For instance, each clock generator has a maximum frequency. Occasionally, updates to the Tegra software on the Android system can cause such components to run outside of these tolerances. To prevent this from happening, software engineers perform stringent regression testing on many Tegra components, including clock generators and multiplexer pins. Manually testing these components is extremely inefficient and complex, as there are over 250 clocks and 250 pins. Thus, we were tasked with developing automated tools to speed up the regression test of clock generators and multiplexer pins.

The first utility, the clock tree sanity tool, accomplishes the task of regression testing clock generators (clocks). We began with a preliminary version of the tool developed by software engineers at NVIDIA. There were several issues with this version that made it susceptible to false negatives and complete failures. We were tasked with developing a more efficient and stable tool. The first issue was that clocks are prone to slight fluctuations, which must be ignored when sampling. Therefore, we decided the most effective solution was to take multiple samples over a given interval. We had to build the internal structure for maintaining multiple samples and calculating the mode of the samples in order to best represent one clock tree. Another issue we encountered is that the clocks were affected by power use. The state of the device is relevant to power use, as an idle state uses much less power than a graphic intensive application. Each of these states must be tested individually. We added functionality so that the tool could put the device in a different state before it began sampling. The last issue we had to overcome was extra power usage as a result of the USB connection with the test device. We decided to replace the USB connection with a RS-232 serial connection, which uses much less power. Then, we changed the commands to communicate over the new port. At the conclusion, we had developed a tool that is easily used and satisfies all of our initial requirements. The final clock tree sanity tool can sample and compare clocks efficiently in less than five seconds.

The second utility, the pin multiplexer tool, completes the task of regression testing multiplexer pins. We faced several key problems during the design and development process. The first problem was the format of user input. We had to design an easy and flexible input format for the user to generate. Our solution was a comma-separated value file format which was easily generated and supported by an existing Python module. The second problem was how to set the device to a user-defined power state, similar to the issue with the clock tree sanity tool. We solved this issue by researching android shell commands and application management and we developed the tool to set power state accordingly. The third problem was parsing the system output from a serial connection. As on Tegra 3 devices, both kernel space and user space messages are mixed together, and thus we needed to identify and separate them for later use. Furthermore, the shell output format differed between different Android versions. We had to account for different shell prompt positions and different characters used between these versions. In these circumstances, we carefully studied the Android system to learn its output format and we were able to handle these issues successfully. In conclusion, we built an easy to use, flexible,

and good quality tool that is able to reduce the time and difficulty needed during the pinmux testing process.

1.0 Introduction

The goal of our project was to develop regression testing utilities to automate the testing of clock generator and pin multiplexer components in the Tegra 3 chip. We will discuss clock generator and pin multiplexer components in more detail in Section 2.2. In this section we will introduce our project in the context of its problem background. We will highlight the pitfalls of manual testing, followed by the advantages of automated testing, and finally how automation would benefit our project in the fields of efficiency and accuracy.

Glenford Myers writes in his book, *The Art of Software Testing*, "Testing is the process of executing a program with the intent of finding errors." [1] The benefits of testing are apparent, as testing helps to minimize the amount of defects found in a product. Software defects become more expensive to fix as the project timeline progresses. Therefore, manual testing, the most common form of testing, helps to alleviate that cost. But this is not enough. According to a study released by the National Institute of Standards and Technology, in 2002 "software bugs, or errors, [were] so prevalent and so detrimental that they cost the U.S. economy an estimated $59.5 billion annually." [2] One reason for the high number and cost of software bugs in 2002 was the unreliability of manual testing. Manual testing is very time consuming and can be inaccurate. The solution to this issue is automated testing.

Automated testing became increasingly prevalent in the last few years. It can drastically increase accuracy and efficiency. In a case study performed by Three Rivers Technologies, automated testing was developed to cover 80% of a client's test plan. This resulted in the reduction of time spent testing from several weeks to four hours. Moreover, there was an increase in the number of defects found and the automation was robust enough to last several years. Overall, the automated tests saved tens of thousands of man hours at the cost of six months to develop the automation. The end product was better quality and the client saved money. [3] Automated testing can apply to any category of testing, including regression testing. Regression testing is the process of testing to ensure no new errors occur as a result of enhancements to existing software.

NVIDIA currently performs regression testing on most of its products. In the case of the Tegra 3 chip, regression testing is performed to ensure that hardware components work within their given tolerances. The manual testing of clock generator and pin multiplexer components is inefficient and complex. There are over 250 clocks and each clock has 10 attributes. Similarly, there are over 250 multiplexer pins. Given the wide benefits of automated testing, our project objective was to develop automated utilities to regression test the clock generators and pin multiplexer components of the Tegra 3 chip. Such automation should be able to reduce the time spent regression testing these components dramatically and increase the accuracy of the tests.

Our report is outlined in the following sections:

Chapter 2 discusses the background knowledge relevant to understanding our project. This includes information about clock generators, pin multiplexers, the Android OS, and Tegra processors. Afterwards, Chapter 3 states the methodology and requirements of our project, including main deliverables. There are two parts in Chapter 3; Chapter 3.1 discusses the clock tree sanity tool and Chapter 3.2 discusses the pin multiplexer tool. Finally, Chapter 4 discusses the results and conclusion of our project. Further information regarding our citations and various materials can be found in the references and appendices section at the end.

2.0 Background

2.1 NVIDIA

2.1.1 Overview

NVIDIA, founded in January 1993, is a semiconductor company that produces graphics processors, wireless communications processors, PC motherboard core logic chipsets, and digital media player software. The company's mission is to "build a great company and be the most important 3D graphics company in the world". It invented the graphics processing unit (GPU) in 1999, which led the world into the computer graphics computing era. NVIDIA holds more than 2100 patents worldwide and it is one of the most famous companies in Information Technology industry. [6] [8]

2.1.2 Main Brand

NVIDIA has many notable product families, including:

- **Riva TNT –** The "Real-time Interactive Video and Animation accelerator" was released in 1998, which became a milestone product for the NVIDIA era.
- **GeForce –** The world's first GPU belonged to the GeForce series. It was a graphics card series that covered all tiers of PC markets.
- **GoForce –** The GoForce series were chipsets used mainly in handheld devices. It was later replaced by the Tegra series.
- **nForce –** The nForce series was a motherboard chipset created for the AMD Athlon/Duron CPU.
- **Quardro –** The Quadro family was a professional graphics solution for high-end performance and quality. It aimed at accelerating Computer-Aided Design (CAD) and digital content creation.
- **Tesla –** NVIDIA's third brand of GPU was the Tesla family. It was the first dedicated general purpose GPU. It was designed for highly parallelized general processing necessary for supercomputing.
- **Tegra –** The Tegra family, launched in 2008, was a system-on-a-chip (SoC) built for mobile devices. The Tegra 3 became the first CPU with quad cores, which aided in increased performance and power management. [9]

2.1.3 Historical Events

- **1993** – NVIDIA is founded by Jen-Hsun Huang, Chris Malachowsky and Chris Priem
- **1995** – NVIDIA launches its first product, NV1
- **1997** – RIVA 128 is launched; one million units are sold in the first 4 months
- **1999** – NVIDIA invents the GPU
- **2000** – NVIDIA acquires the graphics pioneer 3dfx
- **2001** – NVIDIA enters the integrated graphics market with nForce
- **2004** – The Scalable Link Interface (SLI) is launched
- **2005** – NVIDIA develops the processor for Play Station 3
- **2006** – CUDA architecture is unveiled
- **2008** – NVIDIA launches the Tegra mobile processor
- **2010** – NVIDIA builds the world's fastest supercomputer [7]

2.2 Tegra Processors

2.2.1 Introduction

Tegra is a system-on-a-chip (SoC) CPU produced by NVIDIA. It is the world's first mobile super CPU with dual and quad cores. These cores allow it to handle multitasking fluently. The Ultra-low power GeForce GPU on Tegra could provide a console-quality gaming and high definition video playback user experience without compromising battery life. [11]

Our project was based on the hardware of Tegra 2 and Tegra 3. Tegra 2 contains a dual-core ARM Cortex A9 CPU with up to 1.2GHz frequency, while Tegra 3 is a quad-core CPU with up to 1.4GHz frequency for each core. Both chips use different CPU structures and the pins provide varied functionalities. Internal to NVIDIA, Tegra 2 and 3 are referred to as T20 and T30 respectively. [11]

2.2.2 Clock Distribution Network

A clock generator, also known as a clock, is a component of an integrated circuit that generates oscillating square frequency waves, as depicted in Figure 1. The crest of the wave is associated with a 1, while the trough of a wave is associated with a 0. This binary representation allows the circuit to easily understand the signals. The clock cycle, the time between the rising edges of adjacent periods, generally corresponds to one hardware instruction. Since each clock controls a different section of a circuit, the clock cycle directly controls the speed at which instructions are processed in the section. Generally multiple clocks are combined on an integrated circuit to form a clock distributed network, or clock tree.

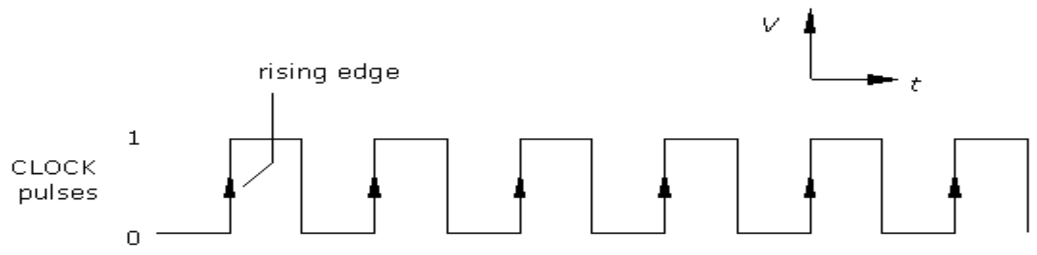

Figure 1: Clock Generator Signal [13]

A clock tree controls the frequency signals of each individual clock. According to an article written by Eby Friedman, a Professor of Electrical and Computer Engineering at University of Rochester, "clock signals are typically loaded with the greatest fanout, travel over the longest distances, and operate at the highest speeds of any signal, either control or data, within the entire system" (Friedman). For this reason, clocks generally account for about half of the power use of the circuit. To help alleviate this power usage, clock trees are set up in such a way to allow clock gating. Clock gating refers to the process in which individual clocks can be directly controlled and turned on or off. There is a positive correlation between performance and power use. Resource intensive applications require higher performance, and as a result, increased power use. The benefit of clock gating is apprent when the device is in a resouce non-intensive state, such as an idle state. In this case, a clock tree has the ability to turn off clocks in order to save power. Thus, it is crucial for the clock tree to balance performance needs with power management. Clock gating is particularly useful in mobile devices because the battery power source requires better power management. The Tegra 3 chip makes great use of clock gating. It employs about 250 different clocks that control all aspects of the chip.

2.2.3 Pin Multiplexer

Tegra 3 devices provide a large number of pins to communicate with other hardware. Among all of these pins, 244 input/output pins could be used as either general-purpose input/output (GPIO) or special function input/output (SFIO). A general-purpose pin is a generic pin whose behavior is controlled by software, while a special function pin is a pin whose function is pre-defined in the hardware. On a Tegra 3 chip, each SFIO pin has 4 different possible pre-defined functions.

To control these pins, Tegra 3 applies a multiplexing logic design. Figure 2 shows the logic gates associated with a pin.

CONFIDENTIAL MATERIAL

Figure 2: GPIO/SFIO Pin Multiplexing Architecture [10]

The gates represented in the figure control the function of a pin. By changing the value of the gate input, the output will change accordingly, thereby changing the function of the pin. Programmatically, we can change the input by writing values to the address of some registers in the operating system.

2.3 Android OS

Originally, Android Inc. developed the Android OS before Google acquired it. Android is a product of the Open Handset Alliance, which is a syndicate of open source mobile technologies and software. There are more than 80 corporations that belong to the Open Handset Alliance including Google and NVIDIA. [12]

Android is an open source system designed for mobile devices. Based on a Linux foundation, the Android platform provides the operating system, middleware, and key applications. Android is built upon C/C++ libraries and provides a hierarchical structure including the Linux kernel, Libraries, the Application Framework, and Applications. The Linux kernel contains all of the drivers and power management policies for the devices. The Libraries contain OpenGL ES for 3D graphics and SQLite for database support among other things. The Application Framework provides several managers for resources, notifications, and locations. Lastly, the Application level provides the high level applications including the phone and web browser. One key component of Android is the Dalvik Virtual Machine, which is a process virtual machine designed to efficiently handle process control and multitasking. [2]

Android has gone through several version iterations including Android 2.3 Gingerbread, Android 3.2 Honeycomb, and more recently, Android 4.0 Ice Cream Sandwich. Each release provided several enhancements, including changes to their shell command syntax. For this reason, we needed to adjust our software to be compatible with all versions.

2.4 Device Input/Output

There are several methods of communication available to send commands to the Android OS and the hardware. The Pinmux utility was required to support the following connection types:

Android Debug Bridge/ Universal Serial Bus

Android Debug Bridge (ADB) is a command line tool included in the Android Software Development Kit (SDK). It is used for communicating with an Android emulator instance or android devices. To use ADB, the device has to be connected to the development computer with a Universal Serial Bus (USB) connection. We were able to install packages, launch applications, and send various commands through ADB. [1]

PySerial / Recommended Standard 232 Port

Recommended Standard 232 (RS-232) Port is usually a 25 pin blue port. We were able to use this port to receive kernel messages directly from the connected Android device. We could also send commands to its shell and receive output. In order to perform these operations in an easier way, we applied a Python module called PySerial. This module encapsulates the access to the serial port. It provides backend support on multiple platforms. [14]

GFShell / Joint Test Action Group

Joint Test Action Group (JTAG) is commonly known as an IC debug port. The port is used for testing the circuit boards. Using JTAG, we were able to read and write values to the registers on the Tegra CPU without the aid of the operating system. GFShell is the corresponding Windows software used for controlling JTAG connections. [5]

3.0 Methodology

3.1 Clock Tree Sanity Tool

The Clock Tree Sanity Tool is used for regression testing and debugging. It is an NVIDIA-internal utility used by Quality Assurance and Software Engineers. At a basic level, the tool compares a base clock tree with a sample clock tree taken from a device and it outputs clock attributes that mismatch. The base clock tree is known as the Golden Base Tree and it is the ideal clock tree which is used as a reference. A warnings file may be input, containing all attributes that the user wants the script to ignore. If such clock attributes mismatch, the script throws a warning rather than a failure. A separate script is run to parse the warning file.

Before we began our project, NVIDIA engineers had developed a basic implementation of the tool; however it was unstable and missing key features. This implementation had the capability to accept and parse a warnings file. It was also capable of gathering one clock tree sample from the device using an ADB/USB connection. However, due to the sensitive nature of the clocks, extra power use as a result of the USB connection changed clock values and produced false negatives. Moreover, gathering one sample was error prone as the clocks are unstable and are subject to random fluctuations, which must be ignored. The script stored all clock trees as a tree list structure. This structure was impractical because it relied on the organization of the clock tree, which varies. The base implementation was able to traverse the base tree list and sample tree list and output fields that mismatched. Our project modified and extended this tool to fix such defects and implement new features.

3.1.1 Requirements

There were four main categories for the initial requirements including portability, flexibility, ease of use, and maintainability.

Portability: The tool needed to be portable and have the ability to run on two different systems. It needed to be able to run on Gerrit Virtual Submit (GVS), a build system. It also needed to run as a standalone tool in a Linux environment.

Flexibility: The tool needed to be developed in such a way to easily allow changes, including changes to input parameters, input files, and output structure to fit the needs of the user.

Ease of Use: The tool must be user-friendly, simple, and quick. The output must be clear and informative.

Maintainability: The tool must be extensible and accommodate for future changes. The source code must contain accurate and useful documentation and there must be a corresponding wiki and README.

From a high level standpoint, the existing implementation included a basic structure and some useful Python code. The warnings file and warnings parser only required minor alterations. In order to decrease false negatives, the script needed to gather multiple samples. The mode was calculated from these samples in order to ignore negligible clock fluctuations. We converted the connection type to a RS232 serial connection from an ADB/USB connection, which altered clock attributes. We converted the mechanism for storing the clock trees to an ordered dictionary tree rather than a tree list in order to increase stability. An ordered dictionary tree does not depend on organization of the tree, as objects are accessed via a key-value pair. Lastly, we added new functionality to allow the device to be tested under several use cases, including audio and video playback, as well a game state. Such use cases were required because the clock trees differ based on the processes running and the power use of the device.

3.1.3 Usage

The usage instructions for the tool are as follows:

clockTree_parser.py

```
usage: clockTree_parser.py [-h] [-warningFilename]
                              baseFilename interval repeat {os_idle, video, audio, game}

Clock Distributed Network Power Sanity Tool

positional arguments:
  baseFilename        golden base clock tree used as point of reference
  interval            time interval between which each sample is pulled from the device
  repeat              how many samples are pulled from the device (mode is calculated)
  {os_idle, video, audio}
                      state of the device based on use case

optional arguments:
  -h, --help          show this help message and exit
  -w warningFilename
                      file containing list of exceptions/warnings for certain clocks

Example:     ./clockTree_parser.py baseClockTree 0.1 20 os_idle –w warningsFile
             ./clockTree_parser.py -h
```

warnings_parser.py

```
usage: warnings_parser.py [-h] warningFilename

Warning File Parser

positional arguments:
  warningFilename     file containing the list of exceptions/warning for certain clocks

optional arguments:
  -h, --help          show this message and exit
```

Example: ./warnings_parser.py warningsFile
./warnings_parser -h

3.1.4 Input

The script allows two files to be input, including the Golden Base Tree file and the warnings file. The Golden Base Tree will be in the same tree format as exists on the device, but with ideal clocks. Figure 3 illustrates a portion of a clock tree:

Confidential Material

Figure 3: Sample Clock Tree

This clock tree includes the name of the clock along with several attributes including the clock state, ref, div, and rate. The asterisk symbol before the name states that the clock has not changed from its initialized value. An exclamation point symbol before the name alerts that the clock has run over its maximum frequency. The dollar sign symbol specifies that the clock is in a sleep state. The state field specifies whether the clock is currently on or off. Also, the rate states the frequency that the clock is currently running (this is only true for clocks that are in the on state). Each base attribute value is compared to the corresponding sample clock attribute value. However, in the case where the state is off, other values such as the ref, div, and rate are ignored.

The warnings file allows the user to specify clocks and clock attributes that should be ignored. Even when comparing the mode of several clock tree samples to the base, some clocks are still unstable and outliers cannot be ignored. In this case, the clock should be added to the warnings file. The warning file has a specific format that is parsed when the tool is run. The warnings file is an optional parameter. Figure 4 demonstrates an example warning file.

```
#[clock_name]
#state <ignore>  ---> ignore the state while error checking
#ref {+,-}<variance>  ---> specify the tolerance range for ref
#rate <min rate> <max rate> ---> specify the min/max rate
#<ignore>  ---> ignore the attribute specifications for this clock
#completely
#Note: isInitial, isMax, isAsleep, div, sharedRate, vddVolt van only be
#set to ignore

[pll_p]
state ignore
ref +-1

[cclk_lp]
rate 102000000 204000000

[i2c3]
Ignore
```

Figure 4: Sample Clock Tree Warning File

In this example, there are three clocks. In the first clock, the state is ignored and the ref attribute is given a tolerance of plus or minus one. The second clock specifies the maximum and minimum value for the rate attribute. The third clock is completely ignored, including all attributes. If any of these aforementioned attributes are mismatched, the script will output a warning instead of a failure.

3.1.5 Output

Output was required to be clean and informative. It specifies the failed clock name, the mismatched attribute along with sample value and base value, and the immediate tree hierarchy of the failed clock. Figure 5 depicts the desired output:

```
FAILURE: cpu
   Test Image Value: ref: [7]
   Expected Value: ref: [8]
clk_m
       pll_e
             pll_p
                   cclk_lp
                         cpu_lp
                                cpu

WARNING: i2c1
   Test Image Value: state: [‘on’]
   Expected Value: state: [‘off’]
clk_m
       i2c1
```

Figure 5: Sample Clock Tree Output

In this example, the first clock throws a failure and the second clock merely throws a warning. The name of the clock is given as well as the expected and sample value of the mismatched clock attribute.

3.1.6 Project Deliverables

Over the course of the development process, we completed several iterations with three main core deliverables. Core Deliverable 1 required enhancements to fix errors with the preliminary implementation. We upgraded the script to Python 2.7 and updated command line argument passing and validation. We converted the tree list structure into a more robust ordered dictionary tree structure for storing clock trees. Also, we consolidated, refactored, and documented existing code.

Core Deliverable 2 required the reduction of false negatives when comparing the base tree to the sample. We developed a method of gathering multiple clock tree samples and calculating the mode of such samples. This prevented false negatives due to negligible clock fluctuations. Other featured added include random intervals between samples and enhanced output.

Core Deliverable 3 required the addition of use cases and integration with GVS. The group added several use cases to the script including an idle state, audio and video playback, and a game state. In order to achieve GVS integration, the group needed to convert from an ADB/USB connection to a RS232 serial connection. This also prevented false negatives due to the effect of the USB connection on the clocks. Aside from core deliverables, a final iteration required the addition of README files, wiki pages, and code documentation.

3.2 Pinmux Tester Tool

Pinmux Tester is a tool used for debugging and testing Pinmux (PM) configurations. Its intended users would be the Hardware Team, Software Team, and Software Quality Assurance Team of NVIDIA.

At a basic level, the tool accepts PM configurations from a plaintext CSV file or interactive command line and then performs actions based on the information. A set of rules based on the target chip confirms the accuracy of the supplied PM configuration. The tool has the functionality to either compare a list of PMs with sample values taken from the device or write PM values directly to the device according to the configuration file. These two actions are performed in certain power state use cases, such as OSIdle, display-off, and audio playback.

3.2.1 Requirements

There are four main categories for the initial requirements including portability, flexibility, ease of use, and maintainability. The tool needs to be usable on Windows, Linux, or Mac. Also, the utility must support all three chips (T20, T30 and T33) and be extensible for future chips. Meanwhile, it must be flexible enough to use multiple means of communicating with the target devices: RS232 serial connection, android debug bridge (ADB), or JTAG via GFShell. The tool must be easy and quick to use. The output must be clear and appropriate with different verbosity levels specified by the user. For maintainability, the tool must be well documented, through code comments and a wiki. The documentation should provide enough insight to aid new engineers who continue to develop the utility.

3.2.2 High-Level Design

From a high level view, the tool was implemented in Python 2.7. User input is in the form of a Comma-Separate Value (CSV) file or interactive shell-like command line. Input data is checked based on the target device. An ADB or serial connection enters the system into desired power states. The reg_dump tool performs register read/writes.

3.2.3 Usage

The usage instructions for the tool are as follows:

```
Usage: PMTester [OPTIONS]…

 -f <file>                    Input file, in CSV format
```

-o <file>	Target log file; default is stdout
-v [0-2]	Specify verbosity; default is 1 0: Success/failure 1: Show which Pinmux names lead to errors 2: Show which Pinmux names lead to errors and reason: illegal values, failed writes, etc.
--debug	Show debug messages
-i	Interactive mode
-d	Double check writes. Performs a read after write to verify write was successful.
-r	Process and validate input, but do not perform any actions on target
-c [T20, T30, T33]	Select target chip
-a [gb, hc, ics]	Select Android version. Used for any OS-specific settings, such as app launch commands.
--serial [device]	Communication to target will be done over RS232. Default device: /dev/ttyUSB0
--adb [serial #]	Communication to target will be done over ADB. By default, no serial # is used (only 1 device connected).
--info [string]	Print Pinmux info for the selected chip and quit. If no string is supplied, information about all available Pinmux will be printed. Information includes register name, address (base and offset), reset value
-h, --help	Display this message and quit.

3.2.4 Input

Input is accepted as a CSV file or via command line during interactive mode. If ambiguity exists in partial register names, an error is reported.

Pinmux Name

The full name of a PM, for example, looks like "PINMUX_AUX_GPIO_PV3_0", which maps to a four bytes address 0x70003064. Acceptable input would be one of the following: full name, a unique partial name in between two underscores in the full name, full address with or without starting "0x" or last four digit of the hex address.

Pinmux Value

PM values are one byte on T30 and T33 chips, while the values are four bytes on T20 chips. Each bit of the values defines either a default value or user input. The name of the bits that need to be given from user input are PM, PUPD, TRISTATE, E_INPUT and LOCK. For example, if the PM value is 3 for a certain Pinmux, the first two digits of its value would be 0b11.

Power State

There are eight possible power states including:

- OSIdle: System idle state with the screen turned on

- OSidle_Display_Off : System idle state with screen turned off
- LP1: Specific hardware setting when OS is in suspension
- 3D: UnrealEngine 3 3D scene is playing
- VIDEO: mp4 video is playing
- AUDIO: mp3 music is playing
- WEB: Web browser is opened and a webpage is loaded
- RIPTIDE: Riptide game is running

CSV File

CSV files are parsed serially (no rearrangement of data will occur). Each input line translates into internal commands, which issue chronologically. CSV input comments start with '#'. In addition, special comment, such as '#!WAIT', will cause the utility to pause before continuing to issue the remaining commands. These special comments can be used if the user wishes to manually interact with the system before running the rest of the commands. The comment '#!CLEAN' causes the system to reset to the OSIdle mode and the '#!REBOOT' command restarts the system.

CSV fields include:

- PM name
- PM value (PM, PUPD, TRISTATE, E_INPUT, LOCK)
- Action (read/write/double check write)
- Power State
- Power State detail (video/audio playback files)

CSV file example:

Figure 6: Sample Pinmux CSV File

Corresponding actions:

1. When the program starts, it initializes itself, parses the file, validates, and opens a connection to target.
2. For the first Pinmux, HDMI_CEC, the program enters target into OSIdle. It reads PINMUX_AUX_HDMI_CEC_0 and updates PM, TRISTATE, E_INPUT, and LOCK bits to specified values. It writes a new value to PINMUX_AUX_HDMI_CEC_0. If applicable, it reads the same register and compares the result to the expected value.
3. For second Pinmux at 0x7000E41C, the program starts the media player and plays up_h264_1080_10M.mp4. It reads APBDEV_PMC_DPD_PADS_ORIDE_0 and compares the result to expected values.
4. The program pauses execution and waits for user input to continue at the "#!WAIT" command
5. For Pinmux, CLK_32K, the program enters target into OSIdle. Similar to step 1, it reads PINMUX_AUX_CLK_32K_OUT_0 and updates the specified values. If applicable, it reads the same register and compares the result to 0x00000014.

Interactive Mode

Interactive mode offers a quick and convenient way of running quick tests. The interface is similar to a command line debugger, such as GDB.

Supported command:

r <addr>	- Read
w <addr> <value>	- Write
d <addr> <value>	- Double check write
chip <T20|T30|T33>	- Change rules/settings
find <name|address>	- Looking for register address by partial name, e.g. find clk find 314c find 7000314c
state <usecase> [<file>]	- Change power state, supports following use case: OSIDLE, OSIDLE_DISPLAY_OFF, UE3, RIPTIDE, WEB, LP1 AUDIO (need file path), VIDEO (need file path)
shell <command>	- Execute command on android shell
q, quit	- Quit

3.2.5 Output

There are three output verbosity levels. Level 0 displays only a "success" or "failure". Level 1 displays PM names that lead to errors. Level 2 displays PM names that lead to errors and the reason for the error, such as illegal values or failed writes.

3.2.6 Project Deliverables

The project splits into several core deliverables, including:

Core Deliverable 1:

- CSV parser
- Command line interface

Core Deliverable 2:

- Parse internal data
- Set use cases on OS
- Apply register read/write commands to devices through reg_dump based on RS232 and USB connection

Optional Deliverable 1

- Allow PM read/writes to be done through GFShell rather than through reg_dump

Optional Deliverable 2:

- Interactive mode, which is a working front-end interactive command line interface that is able to parse user input into internal data

4.0 Results & Conclusion

Overall, our project was successful. We were able to meet all requirements and deliver both test utilities ahead of schedule.

Clock Tree Sanity Tool

The Clock Tree Sanity Tool project lifecycle consisted of three core deliverables along with an iteration of cleanup and documentation. Each core deliverable was followed by a short testing phase, which provided valuable feedback on the tool. This feedback was used to make slight modifications to requirements when necessary.

The first core deliverable required us to fix errors in the preliminary version of the tool. We focused on refactoring and consolidating. We upgraded to Python 2.7 and updated the input arguments and validation. We were able to merge several functions together, while removing functions that were unnecessary. In preparation for the second core deliverable, we updated the structure used to hold the clock tree samples to an ordered dictionary tree for greater robustness.

The second core deliverable involved more dramatic changes. A single clock tree sample proved to be ineffective as clocks are not completely stable and are known to have random fluctuations. Our solution was to add functionality to gather multiple samples. The samples were stored in a ‘master tree’. This master tree was an ordered dictionary tree that combined all samples into one tree structure. The master tree was converted into a mode tree, where the mode was calculated for each clock attribute. The mode best represented the values of the sample by removing outliers. The mode tree was compared to the golden base tree and mismatched attributes were printed as output. Feedback from the testing phase led us to update the output to increase readability.

The final core deliverable required us to add different use cases. The utility had to be upgraded to send device commands using an RS-232 connection. These commands included installing the use case applications, running the applications, and gathering clock samples from the device. Several new use cases were added including audio and video playback, as well as camera and game usage. These use case applications had been built previously by the Software Quality Assurance team for use with power regression testing. At the end of the iteration, the testing phase involved integration with the GVS build system.

Upon successful runs on the GVS build system, the final iteration called for code documentation, README file, and a wiki page. Throughout the project, we kept good practice of documenting code. Thus, we only needed to build a README file and a wiki page. We also had to update the warning file to include hotplug deviations, clocks that are so unstable that even the mode of multiple samples may be incorrect. Such clock attributes must be ignored and have warnings thrown instead of errors in the event of a mismatch. Following these changes, we submitted the utility for final code review.

Pinmux Utility

The Pinmux Utility consisted of two main parts: the front-end (core deliverable 1) and the back-end (core deliverable 2). The front-end deliverable required processing and saving user input as internal data while the back-end deliverable required the procedure of operation on Android system and devices. As the two parts were closely related to each other, we decided to apply an agile development process. We began by building a minimal workable module for both parts and then by adding the required functionality iteratively. Using this method, we were able to test the complete utility whenever a small piece of work was finished.

The front-end deliverable required us to handle program parameters (flags), parse a CSV input file, and save the parsed information as internal data. In Python, program parameters are commonly handled by the "optparse" module. However, after researching, we found that this module had been recently upgraded and renamed "argparse". The functionality of the "argparse" module was greatly improved to include better parameter validation and parameter management. The disadvantage of this upgrade was its incompatibility with Python versions lower than 2.7. Therefore, we were required to upgrade our test environment to Python 2.7. In addition, to parse the CSV file, we chose to use another Python module, "CSVReader". Considering the fact that the CSV input file might be very long, we parsed the file one line at a time in order to reduce the memory used. We tested the front-end and it worked well.

The back-end deliverable was far more complicated. We needed to set the Android OS into different power states and also operate the registers via different connection types. The strategy we used was to modularize each of the above functions. We tried to minimize the data sharing between these modules in order to reduce their influence on one another. Thus, any changes to a module would not be able to break another module accidentally. Each connection type had its own module and each power state had its own module. At the end of the project, we carefully tested all the modules to ensure compatibility with the front-end.

We performed several demos, as well as a final presentation, to our team. Throughout the project, we also communicated with other teams in the company and gained a lot of assistance and professional feedback. Based on the feedback, we improved our utility greatly. Finally, the code was peer reviewed and we submitted it to the company repository.

References

[1] Android.com. Android Debug Bridge

[2] Google Inc. Android Developers

[3] Myers, Glenford J. The Art of Software Testing 1979

[4] National Institute of Standards and Technology. Software Errors Cost U.S. Economy $59.5 Billion Annually 2002

[5] Nelson Chow. GFShell Introduction 2012

[6] NVIDIA. History

[7] NVIDIA. NVIDIA History - A Timeline of Innovation

[8] NVIDIA. NVIDIA Overview - Vision, innovation and market leadership

[9] NVIDIA. Products

[10] NVIDIA. TECHNICAL REFERENCE MANUALNVIDIA TEGRA 3HD Mobile Processors 2012

[11] NVIDIA. Tegra 2 And Tegra 3 Super Processors - Do more with multi-core processor performance 2012

[12] Open Handset Alliance. Android Overview

[13] Phillips, W. D. Design Electronics 1999

[14] pySerial. pySerial Overview

[15] Three Rivers Technologies. Automated Testing Case Study

Appendices

Appendix A: Clock Tree Sanity Tool Usage

clockTree_parser.py

```
usage: clockTree_parser.py [-h] [-warningFilename]
                              baseFilename interval repeat {os_idle, video, audio, game}

Clock Distributed Network Power Sanity Tool

positional arguments:
  baseFilename         golden base clock tree used as point of reference
  interval             time interval between which each sample is pulled from the device
  repeat               how many samples are pulled from the device (mode is calculated)
  {os_idle, video, audio}
                       state of the device based on use case

optional arguments:
  -h, --help           show this help message and exit
  -w warningFilename
                       file containing list of exceptions/warnings for certain clocks

Example:       ./clockTree_parser.py baseClockTree 0.1 20 os_idle –w warningsFile
               ./clockTree_parser.py -h
```

warnings_parser.py

```
usage: warnings_parser.py [-h] warningFilename

Warning File Parser

positional arguments:
  warningFilename      file containing the list of exceptions/warning for certain clocks

optional arguments:
  -h, --help           show this message and exit

Example:       ./warnings_parser.py warningsFile
               ./warnings_parser -h
```

Appendix B: Clock Tree Sanity Tool Warnings File

```
#[clock_name]
#state <ignore>  ---> ignore the state while error checking
#ref {+,-}<variance>  ---> specify the tolerance range for ref
#rate <min rate> <max rate> ---> specify the min/max rate
#<ignore>  ---> ignore the attribute specifications for this clock #completely
#Note: isInitial, isMax, isAsleep, div, sharedRate, vddVolt can only be set to ignore

[pll_p]
state ignore
ref +-1

[cclk_lp]
rate 102000000 204000000

[i2c3]
Ignore
```

Appendix C: Clock Tree Sanity Tool Output

```
FAILURE: cpu
  Test Image Value: ref: [7]
  Expected Value: ref: [8]
clk_m
     pll_e
          pll_p
               cclk_lp
                    cpu_lp
                         cpu

WARNING: i2c1
  Test Image Value: state: ['on']
  Expected Value: state: ['off']
clk_m
     i2c1

Missing Clock: cclk_m
```

Appendix D: Clock Tree Sanity Tool Schedule

Timeline	**Project Stage**	**Details**
Day 1-3	Set up environment	• Sync and build code • Acquire permissions, mailing lists, equipment
Day 4-5	Preliminary research into existing tool	• Motivation for this project • Clock Trees ○ What are they ○ How are they used ○ Use case scenarios • Design concepts
Day 6-10	**Core Deliverable 1:** Fix initial errors in the current implementation and reduce false negatives	• Understand design implementation • Fix issues and enhance current design implementation
Day 10-11	Core Deliverable Test/Feedback	• Test tool locally and on GVS build system • Make changes according to feedback from results
Day 12-18	**Core Deliverable 2:** Gather multiple samples and take mode of samples. Reduce false negatives.	• Gather multiple samples based on interval and repeat input values ○ Add randomization to interval value • Take mode of samples • Compare mode to golden base tree
Day 19-21	Core Deliverable Test/Feedback	• Test tool locally and on GVS build system • Make changes according to feedback from results
Day 22-26	**Core Deliverable 3:** Add different use cases and convert from ADB/USB to UART serial connection.	• Add use cases: idle, audio, video, game • Convert from ADB/USB to UART

Day 27-28	Core Deliverable Test/Feedback	• Test tool locally and on GVS build system • Make changes according to feedback from results
Day 29-34	Add code documentation, README, and a wiki page. Consider hotplug deviations, update warnings file, update base trees	• Cleanup code, add documentation • Add README and wiki page • Consider hotplug deviations o Update warnings file • Update base trees for all use cases
Days 35	**Finalize**	• Submit for code review • Final testing • Develop presentation/demo
Days 36-38	Spare Time	• Spare time for schedule slips or other assignments

Appendix E: Pinmux Tester Tool Usage

```
usage: RegChecker.py [options]...

  -h, --help        show this help message and exit

  -f <File>         Input file, in CSV format

  -o <File>         Target logfile; default is stdout

  -v [0-2]          Specify verbosity; default is 1

                          0: Succeed/failed

                          1: Show which Pinmux names lead to errors

                          2: Show which Pinmux names lead to errors and reasons


  --debug           Debug mode

  --config [<File>]  Import configuration file, default is t30_allregs_config

  -i                Interactive mode

  -d                Double check writes. Performs a read after write to verify write was successful

  -r                Process and validate input, but do not perform any actions on target

  --install         Install UE3/Riptide/Media Player application automatically

  -a [gb,hc,ics]    Select Android version. Used for any OS-specific settings, such as app launch
commands

  -c [T20,T30,T33]   Select target chip

  -g                read/write PM through GFShell

  --serial [device]  Communication to target will be done over RS232.

                    Default device: /dev/ttyUSB0

  --adb [serial #]   Communication to target will be done over ADB.

                    By default, no serial # is used (only 1 device connected).
```

Appendix F: Pinmux Tester Tool Interactive Mode Usage

ALL <addr> and <value> are hex number that starts with 0x

```
r <addr>            - Read
w <addr> <value>    - Write
d <addr> <value>    - Double check write
find <name|address> - Looking for register address by partial name, e.g.
                      find clk
                      find 314c
                      find 7000314c
state <usecase> [<file>]
                    - Change power state, supports following use case:
                      OSIDLE, OSIDLE_DISPLAY_OFF,
                      UE3, RIPTIDE, WEB, LP1
                      AUDIO (need file path), VIDEO (need file path)
shell <command>     - Execute command on android shell
q, quit             - Quit
```

Appendix G: Pinmux Tester Tool CSV File Example

```
@PinmuxName,PM,PUPD,TRISTATE,E_INPUT,OD, LOCK,IO_RESET,Action,Usecase,Info

#!CLEAN

#!wait
SPDIF_OUT,2,3,1,1,1,1,1,R,AUDIO,/data/222.mp3
SPDIF_OUT,2,0,0,0,,0,,R,,
0x70003354,3,0,0,0,,0,,R,,
0x70003354,4,0,0,0,,0,,R,,
0x70003354,1,0,0,0,,0,,R,,

@PinmuxName,PM,PUPD,TRISTATE,E_INPUT,OD, LOCK,IO_RESET,Action,Usecase,Info
#!wait
GPIO_PBB6,0,0,0,0,,0,,R,Video,/data/111.mp4
SDMMC3_DAT5,0,0,0,0,,0,,R,,

@PinmuxName,PM,PUPD,TRISTATE,E_INPUT,OD, LOCK,IO_RESET,Action,Usecase,Info
#!wait
0x70003060,0,0,1,0,,0,,W,osidle_display_off,
GPIO_PBB6,0,0,0,0,,0,,R,,
SDMMC3_DAT5,0,0,1,,,1,,W,RIPTIDE,
0x70003060,0,0,1,,,1,,R,UE3,

#!wait
SPDIF_OUT,2,0,0,,,,,R,a,,
0x70003060,2,0,0,,,,,R,WEB,

#!wait
PEX_L0_PRSNT,3,0,0,,,,,R,LP1,
PEX_L0_PRSNT,1,0,0,,,,,R,Video,/data/111.mp4
SDMMC3_DAT5,4,0,0,,,,,R,,

#!CLEAN
```

Appendix H: Pinmux Tester Tool Output

```
6: [SFIO] ***Fail : SPDIF_OUT: R, expect 0x1fe (OD=1 IO_RESET=1 TRISTATE=1
LOCK=1 E_INPUT=1 PUPD=3 PM=2 ), get 0x00000000 (OD=0 IO_RESET=0 TRISTATE=0
LOCK=0 E_INPUT=0 PUPD=0 PM=0 )

7: [SFIO] ***Fail : SPDIF_OUT: R, expect 0x2 (OD=0 IO_RESET=0 TRISTATE=0 LOCK=0
E_INPUT=0 PUPD=0 PM=2 ), get 0x00000000 (OD=0 IO_RESET=0 TRISTATE=0 LOCK=0
E_INPUT=0 PUPD=0 PM=0 )

8: [SFIO] ***Fail : 0x70003354: R, expect 0x3 (OD=0 IO_RESET=0 TRISTATE=0 LOCK=0
E_INPUT=0 PUPD=0 PM=3 ), get 0x00000000 (OD=0 IO_RESET=0 TRISTATE=0 LOCK=0
E_INPUT=0 PUPD=0 PM=0 )

9:  ***Fail : 0x70003354: Invalid value

10: [SFIO] ***Fail : 0x70003354: R, expect 0x1 (OD=0 IO_RESET=0 TRISTATE=0 LOCK=0
E_INPUT=0 PUPD=0 PM=1 ), get 0x00000000 (OD=0 IO_RESET=0 TRISTATE=0 LOCK=0
E_INPUT=0 PUPD=0 PM=0 )

14: [SFIO] ***Fail : GPIO_PBB6: R, expect 0x0 (OD=0 IO_RESET=0 TRISTATE=0 LOCK=0
E_INPUT=0 PUPD=0 PM=0 ), get 0x00000020 (OD=0 IO_RESET=0 TRISTATE=0 LOCK=0
E_INPUT=1 PUPD=0 PM=0 )

15: [GPIO] ***Fail : SDMMC3_DAT5: R, expect 0x0 (OD=0 IO_RESET=0 TRISTATE=0
LOCK=0 E_INPUT=0 PUPD=0 PM=0 ), get 0x0000002a (OD=0 IO_RESET=0 TRISTATE=0
LOCK=0 E_INPUT=1 PUPD=2 PM=2 )

19: [SFIO]Succeeded: 0x70003060: W

20: [SFIO] ***Fail : GPIO_PBB6: R, expect 0x0 (OD=0 IO_RESET=0 TRISTATE=0 LOCK=0
E_INPUT=0 PUPD=0 PM=0 ), get 0x00000020 (OD=0 IO_RESET=0 TRISTATE=0 LOCK=0
E_INPUT=1 PUPD=0 PM=0 )
```

Appendix I: Pinmux Tester Tool Project Schedule

Timeline (inclusive)	**Project Stage**	**Details**
Day 1-3	Set up environment	Sync & build code, permissions, mailing lists, equipment, etc.
Day 4	Present Pinmux background, project outline	What are Pinmux, how are they used, debugging scenarios, motivation for this project. Purpose of this project, design (mostly the contents of this document). Emphasis on schedule and deliverables.
Days 5-7	Preliminary research	Implementation details, implementation tools research and learning, Pinmux research for each chip, Pinmux rules implementation design. *Internal data representation should be finalized before development start.*
Day 8	Design feedback meeting	Meet with sys eng, HW, dev. Demo tool and gather feedback on design. Adjust design as needed (may require 1 additional day)
Days 9-12	**Core Deliverable 1:** Front-end and CSV parser	A working front-end command line interface, able to parse CSV files into internal data
Days 13-16	**Core Deliverable 4:** Back-end	A working back-end implementation that is able to take internal data parsed from user input and generate and apply commands to target via ADB or UART under specific use cases
Days 17-20	**Optional Deliverable 1:** Rules files and engine	A working, fully implemented API to validate a set of PM configurations according to rules files. Rules files are to be finalized in this deliverable.

Days 21-22	**Optional Deliverable 2:** GFShell Interface	Allow PM read/writes to be done through GFShell rather than through reg_dump
Days 23-25	**Optional Deliverable 3:** Interactive mode	A working front-end interactive command line interface, able to parse user input into internal data
Days 26-27	Dedicated time for testing and debugging the utility.	At the end of this task, the utility should be ready for use.
Day 28	Feedback meeting with end users	Meet with sys eng, HW, dev. Demo tool and gather feedback. Depending on feedback, may require additional time to implement new features or change existing ones
Days 29-32	Documentation & Check-in	This includes finishing up code commenting; creating/updating a thorough wiki to be used by both users and maintainers of this tool; review process for checking the code in.
Day 33	Wrap up	After code review is done, final testing, wrapping up documentation.
Days 34-35	Presentation (and demo?)	Create project presentation, demo, etc.
Days 36-38	Spare time - OR - I2C Sysfs interface/utility	Spare time for schedule slips, or to be used to port QA power state script. In case of the latter, part of this time will be allocated before development start on Day 9. - OR - Port I2C Sysfs interface/utility from Motorola Olympus to GB, HC, ICS branches.
Day 39	Wrap up	Last working day wrap up. All scheduled work must be implemented, tested, checked-in, and documented by this point.