Project Number: DXF-NV08

NVestigate

A Major Qualifying Project Report

submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by:

____

Peter Lucia

____

Joseph Mann

____

Evan May

Date: 5 March, 2008

Approved:

____

Professor David Finkel, CS Advisor

Abstract

The goal of the NVIDIA NVestigate project was to develop a tool to assist the reading and writing of hardware specific registers. The programs that were created to accomplish this goal provide many previously unavailable capabilities, including the ability to interact with registers based on their name instead of their memory offset. Also, an intuitive Graphical User Interface was created to interact with both of the Command Line Interfaces developed.

Executive Summary

Attempting to remember the numerical address of an 8 digit number is difficult – trying to remember what each of those 8 digits is responsible for and the possible values for each is even more difficult. Make it a few hundred of these associations, and the task becomes daunting for even those with excellent memories. Referencing a table of these values wastes precious time, and becomes impractical on a large scale; yet, this was exactly what was required of many employees at NVIDIA.

The memory configuration space for hardware devices is set up as a series of memory offsets to memory mapped registers, with the value in memory at that offset being the actual register value. This is, obviously, not the most user friendly set-up, requiring the memorization of memory offsets and potential values. NVestigate allows the user to reference these values through the name of the register and provides the names of the masks the register holds. This approach removes much of the memorization, allowing a greater number of programmers to easily work with the registers. This is of particular interest to many of the programmers within the software storage division of NVIDIA, who are looking at this tool as a valuable asset to their verification of register values. Also, NVestigate will be extended in the future to interact with many more devices, such as network devices.

This project updated Command Line Interfaces (CLI) to allow the use of register names, and a Graphical User Interface (GUI) was developed to present all the information intuitively. NVParse was one of the CLIs, and allowed the user to interact with the machine that the program was running on. NVATA, on the other hand, allowed the user to debug a remotely running machine. This GUI can interact with both NVParse and NVATA, providing an even simpler technique to use these programs. Most of the functionality of the CLIs was incorporated into the GUI, along with some additional features. The GUI is able to read or write to NVParse and WinDbg, create screenshots of the current register values, create a differential view with old values, and perhaps most importantly, provide the user with a simple and intuitive interface to control all of the interface's options.

Through the use of this tool, important information within the registers is now accessible to a greater number of employees at NVIDIA. It will also significantly increase the efficiency of working with registers, as it is now possible to read and write many different registers at a time. These two important metrics, ease of use and productivity, help to indicate the true effect that NVestigate will have upon NVIDIA.

Contents

1 Introduction

In this section we will introduce some basic background to our project as well as provide an introduction to the basic technologies needed to understand this project.

1.1 Corporate Background

NVIDIA is a global company known around the world for its work in graphics cards and motherboards. It was founded in 1993 and has grown to over 4,000 employees and is now publically traded on the stock market. In addition to their work on graphics processors, they perform extensive work on consumer electronic devices and mobile devices. These products are used primarily in graphics intensive applications such as image and video manipulation and gaming. NVIDIA is headquartered in Santa Clara California and has offices around the world. (1)

NVIDIA has several brands, each focused on a slightly different market segment. The NVIDIA GeForce brand is the most well known of the brands, focusing on graphics and video. The NVIDIA GoForce brand focuses on low power mobile graphics processors. The NVIDIA Quadro brand concentrates on high performance graphics processing. Finally, the division we worked with was NVIDIA nForce. This division produces media and communications processors for motherboards. These devices facilitate communication between networking, storage and digital media devices. (1)

1.2 Division Background

This project worked with the division at NVIDIA that produces motherboard chipsets, the nForce division. This division started with work on the XBox game console and progressed into the area of AMD motherboard chipsets. Their chipset is now on its seventh generation and accounts for two-thirds of all AMD motherboard chipsets sold currently. These chipsets are responsible for communicating with nearly all embedded devices on the computer. This includes:

- USB
- PCI Express (PICe)
- Keyboard
- Mouse
- Serial
- Parallel
- Ethernet
- Storage
    - Advanced Technology Attachment (ATA)

    - Serial ATA (SATA)
- And more

The hardware made by the nForce division interfaces with drivers that they make as well. These drivers provide the software to hardware (and vice versa) bridge that is needed at the interface between software and hardware. The software that NVIDIA produces for this product includes: SATA and RAID Drivers (for hard drive control), management tools and firmware.

NVIDIA's SATA drivers are layered on top of the open source standard Advanced Host Controller Interface (AHCI) (2). AHCI provides a standard interface for which systems can interact with the hardware, but NVIDIA's implementation of AHCI also provides advanced functionality to improve performance. Our project focused on extending existing debugger support for AHCI to include support for NVIDIA's advanced functionality as the primary goal.

2 Technologies

This project will use several different technologies and existing tools. In this section, a brief overview of each of these will be provided.

2.1 Peripheral Component Interconnect (PCI)

The Peripheral Component Interconnect (PCI) interface is a common interface used to connect devices to the system's motherboard. It allows for devices connected to the system to communicate with the processor and other buses on the system. Each device connected to the PCI bus is given 256 bytes of memory in PCI Configuration Memory Space (PCI Config Space). The format for the first 64 bytes of PCI Config Space is standardized and includes the device's ID, as well as offsets into virtual memory space. Each device is given a portion of virtual memory space (memory space) that can be used to communicate directly with the device's hardware. Several different types of devices, including Serial ATA controllers, are often implemented to use the PCI interface.(3)

2.2 Serial ATA

Serial ATA, or SATA as it is typically called, was created primarily to interface between computer hardware and storage devices, such as hard drives. SATA is currently available in two different varieties, SATA I, which can transfer at 1.5 Gbit/s, and SATA II, which can transfer at 3.0 Gbit/s. This interface was designed as a successor to the Advanced Technology Attachment, or ATA, standard which was popular in the past. Currently, the standard interface for SATA controllers is the Advanced Host Controller Interface (AHCI). This interface allows advanced functions such as command queuing, an important feature for many of today's programs (4).

One of the main changes from ATA to SATA is in the power and data cables. The data cable used for SATA has seven conductors which includes three ground and four active lines found in two pairs. This data cable is much thinner than ATA cables, and is also able to be much longer with no loss in performance. The new power connector is a 15 pin connector, identifying it easily from the data cable. Various voltages are supplied through the power cord, including a 3.3V, 5V, and 12V power source. Although adapters exist to allow a SATA device to use a standard Molex connector, this type of adaption does not provide the SATA device with a 3.3V source. Many companies are aware of this limitation, and have not made their SATA devices to require a 3.3V source; however, this does eliminate the ability for hot plugging, or switching components while the computer is still on, from the device (5).

2.3 Advanced Host Controller Interface

There are several different specifications and media for communicating with storage devices. One of these formats is the Advanced Host Controller Interface (AHCI) over Serial Advanced Technology Attachment (SATA). As specified in the AHCI specification [6], AHCI describes a system memory layout and a group of commands for communicating with a series of SATA devices. AHCI host bus adapters (HBAs) can support up to 32 devices at a time and must support both ATA and ATAPI devices. AHCI has only one use, which is to facilitate communication between a device or bus and a series of SATA devices. An AHCI HBA typically sits on top of a device chipset or connects directly to a PCI bus. AHCI is designed to be compliant with the PCI base specification. Because of this, AHCI can be used with any device that complies with the PCI specification.

2.3.1 Basic Specification and Architecture

An AHCI bus device can support up to 32 devices, is capable of doing 64-bit addressing, is hot pluggable, and is also able to manage the power of the devices attached to it. The AHCI specification does not specify information about the transport, physical or data link levels of SATA. The HBA contains 64 bytes of registers for the PCI Header, plus additional space for PCI Power Management and Message Signal Interruption. There is an additional 4,351 bytes set aside for specific AHCI registers. (5)

2.3.1.1 Registers

The AHCI spec divides its registers into three separate categories. These categories are Generic Host Control, Vendor Specific, and Port Registers. The General Host Control Registers hold all the information the system requires to communicate with the AHCI HBA. These registers specify the host capabilities of the HBA and allow for control of the functions of the HBA not related to its communication with the devices connected to it. The Vendor Specific Registers span 240 bytes and allow a vendor to extend specific sections of the AHCI specification.

The Port Registers, the final register group on the HBA, allow for communication with the devices connected to the HBA. Each device has its own collection of registers, each spanning 128 bytes. These registers contain interrupt information about the device, as well as the status of the device and any notifications that are required. (5)

2.3.2 NVIDIA Specific Registers

The AHCI specification was designed with extendibility in mind. To accomplish this, the ACHI spec specifies a section of memory that allows a vendor to implement their own set of registers. But because these registers are specific to the vendor that is using them, it is not possible to create a generic tool to debug or inspect these specific memory addresses. This makes it necessary for the vendor to create a custom in house tool rather than rely on the tools that are currently available.

2.4 Windows Debugger (WinDbg)

The Windows debugger has a variety of features and uses. Unlike the Windows Visual Studio debugger, WinDbg can be used to debug any windows executable program, kernel level programs, and even drivers written to run on a Windows platform. WinDbg also allows for the use of any symbol table while debugging a program. This program is freely available from Microsoft and comes as a full suite of programs for low level Windows development and debugging. We primarily only used the WinDbg program from this suite, but we will also need to use the bundled build utilities to compile our extensions to WinDbg.

2.4.1 Uses

WinDbg has several important features. One of its most powerful is the ability to debug a computer running in Windows at a kernel level. By connecting a host and a client PC, each running Windows, with a null-modem or IEEE1394 cable, one can debug the copy of Windows running on the client PC. This is very useful in that it allows one to debug a driver running on another machine without needing to worry about the machine running the debugger crashing or the debugger's footprint showing up on the hardware being debugged. It also allows the user to monitor the loading of the driver before Windows has completed booting, which can be crucial in certain debugging environments. (6)

2.4.2 Extensions

One of the features of WinDbg is the ability to write DLL for it that extend its capabilities. These extensions allow one to write debugger commands that can be used from within the debugger. These extensions are useful for automation and to allow the programmer to streamline simple debugging operations. Debugger extensions are capable of working with any legal debugging targets, including Windows' executables and drivers. However, when debugging remotely, the extension is not allowed to talk directly to the process being debugged; it must connect through the link via WinDbg commands. Debugger extensions are useful when the debugger is reading information that is not normally supported for output. In this manner, a debugger extension can be used to manage and parse data that is not recognized otherwise.

2.4.3 NVIDIA Debugger Extension

An extension to WinDbg which allows registry values found in the AHCI Specification to be viewed in the debugger existed prior to our project; however, there was no debugger support for NVIDIA's proprietary extensions to the AHCI Specification. One aspect of our project involved the creation of a WinDbg extension to allow NVIDIA's developers to view the registry values that they need to see.

3 Problem Statement

NVIDIA's nForce division's work with low level drivers and firmware involves programming at or below the operating system level. This necessitates special tools to assist with debugging because normal user level programs do not have broad enough access to hardware level memory registers for effective debugging. Although no specific tools exist to directly access the information NVIDIA needs to debug their software, Windows provides an extendable debugger for kernel level debugging. Our goal is to extend the Windows Debugger (WinDbg) to allow access to NVIDIA's proprietary memory addresses and also to map these to registry names. Our end product aimed to be future proof in that it will accept new address range mappings in the future and display them as well. As part of this aspect of the project, we enabled the program to create the necessary header files on the fly to be used by the extension. In the past, each of these files needed to be created by hand. Also, we took an existing tool used within NVIDIA, called NVPCI, which allows the programmer access to the PCI data that is located within each device, and adapted it to also utilize the registers names instead of solely the registry values. Finally we created a Graphical User Interface to provide easier access to both the WinDBG extension and the NVPCI extension.

4 Process

Our process started with understanding the tools that we were asked to extend. The current features and limitations drove our design to optimize usability by programmers while leaving the code as open to future modifications as possible.

4.1 Schedule

During our first week at NVIDIA, we met with our mentor to determine the deliverables that would be expected at various points during our project. The milestones that were set included a technical demo, followed by an alpha, beta, and final release.

4.2 Technical Demo

The technical demo occurred at the end of our second week at NVIDIA. The main goal of this presentation was to demonstrate our understanding of the project, as well as to address any questions or concerns that we had. At this point, we were able to show the basic functionality of NVParse, including parsing the reference file and the ability to read and write to registers based on name, instead of memory offset. WinDbg had only the most basic of capabilities, limited to the data structure that would eventually be used. At this demonstration, the addition of a graphical user interface was also made known to us.

4.3 Alpha Release

During the fourth week of work at NVIDIA, we presented our alpha release. The goals set for us for this release were to have a running, non-feature complete program with bugs. We succeeded in surpassing this goal, presenting a nearly feature complete NVParse, along with a vastly improved NVATA program. NVParse could now create C-style macros, create dump files for the other programs to utilize, in addition to the features that it had from the technical demo. WinDbg could now read memory based on the register names, and was also making progress on the additional features that were required.

Also, during this presentation, two different mock-ups of the GUI were presented. One featured a table view of the registers, based off of the previously existing NVPCI tool. Another utilized two different lists to display the information needed by the program. Both techniques, however, used the bottom section of the program to display further information about a selected register, along with buttons for commonly used functions. The eventual users of our tool chose the display with two different lists, but were otherwise happy with the layout of the mock up.

4.4 Beta Release

This release occurred during our sixth week of work, and was meant to be an almost feature complete program, with some bugs that did not completely crash the program. Again, this goal was met completely, and even exceeded in some aspects. NVParse did not gain any substantial new features, but rather was more heavily documented, and had undergone a few format revisions for output. WinDbg could now write to registers based on the name of them, and could also handle different ports along with different output formats.

The GUI was able to parse a dump file generated by NVParse, display information about the Registers in two lists, display more detailed information about the Register when clicked on, and could interact with both WinDbg and NVParse. Also, the GUI had the functionality to write Registers one at a time, or utilizing a queue. At this time, the ability to filter what was displayed in the lists according to tabs was suggested. Also, the ability to automatically load reference files based on a device that is chosen was expressed as a desired function.

Additionally, during this presentation, a desire was expressed to include PCI configuration space registers and introduce the use of multiple reference files. Implementing these required new features in all three aspects of the program.

4.5 Final Release

The Final Release's goal was to be as bug free and stable as possible, while also including all of the features originally laid out for the program. To meet this goal, the ability to handle PCI configuration space was added to NVParse, WinDbg, and the GUI. Also, NVParse and WinDbg gained the capability to determine the devices installed on the computer. This allowed the GUI to also have this knowledge, and we added the ability for the program to automatically load the appropriate reference files for a selected device. This information was stored within a library file that was created for this purpose. Also, the GUI and NVParse gained the ability for the user to specify multiple reference files.

5 System Level Design

The design of our extension was most greatly influenced by the limitations of the existing technologies that we built on top of. Coming into this project we were told to use whatever programming languages and styles that we saw fit the problem; however these choices were limited as we discovered the constraints of the available tools.

It was fairly clear at the beginning of this project that it would have four major parts, a reference (ref) file parser, a WinDbg Extension, a NVPCI wrapper, and a graphical user interface (GUI). We quickly realized that each of these would be best implemented in a different language. For the WinDbg extension we were constrained to using C or C++, Perl was an obvious choice for parsing the ref file and creating the wrapper for NVPCI and C# presented itself as the language of choice for creating a GUI. Perl was chosen to parse the ref file due to the ability to utilize regular expressions, allowing much quicker parsing than would have been possible otherwise. For GUI creation, C# allows many different options through both code and a powerful drag and drop designer, and is one of the most prolific choices in professional applications. This choice for programming languages neatly split our project into three manageable pieces. We also realized early on that these three languages could complement each other by talking to each other through sockets and directly reading each other's command line output. This design greatly simplified the individual parts while placing a decent bit of emphasis on text parsing in each of the three parts.

5.1 Data Structures

This project primarily focuses on creating an easy way to read values from low level memory. In memory, these values are stored in named registers of 32 or 64 bits. Each one of these registers is further broken down with masks that specify which portion of a given register contains the value for a specific flag. Finally, each mask has a set of valid values that could populate it.

5.1.1 Registers

Registers have a name and an offset into memory where they are located. Unlike masks and values, they also can be global or port specific. Registers that are global are exactly what they sound like; there is one instance of them for the whole system. The port specific registers however are repeated once for each port enabled on the system. Finally, each register has a set of masks.

5.1.2 Masks

Each mask belongs to exactly one register. Masks specify which bits in a register represent a certain piece of information. This piece of information has a name as well as a range of bits that signify where in the register the information of interest is. Finally, each mask has a set of flags that the mask could hold.

5.1.3 Flags

Flags have a human understandable string and a numerical representation that will be found in the masked bits specified by this object's parent. Each flag belongs to exactly one mask, although common mask flags ("TRUE 1") may exist under many masks, each mask has its own unique value object for this flag.

6 Module Design

Each module worked on a specific part of the overall project, each doing its job in a slightly different way with a slightly different set of design constraints.

6.1 NVPCI wrapper (NVParse)

The goal of the NVPCI wrapper, also known as NVParse, was to develop a program that would extend the usefulness of NVPCI and enable NVIDIA to parse several ref files. To do this, we choose to implement our program in Perl. Perl was chosen because it allows for powerful yet simple manipulation of strings, which was important for parsing ref files. It was also easy to implement a program that could run on top of NVPCI.

While designing NVParse, we developed the data structure mentioned in Section 5.1. This data structure was very easy to implement in Perl using hash and array references. These structures were later transformed into classes to improve the extendibility of our program. These classes are used by both the parsing system of NVParse, as well as the NVPCI wrapper section of the program.

6.1.1 Parsing System

The original parsing system was implemented as a single function and was designed to read in each register from the ref file and include the associated masks and flags with that register. These registers were then formatted for one of three uses, to be: dumped to a file for use with NVATA or the GUI, written into C style macros, or used by the program internally. The system was originally designed to read in any ref file, but it was modified to work optimally with the AHCI spec by specifying which registers are port registers. These modifications were designed to function transparently as to not affect ref files that are not AHCI.

The ability to add a reference file that specifies PCI Configuration Memory Space Registers was later added to the parsing system. This feature was requested by our mentor and was an easy feature to add despite the rather late request. This was because the compartmentalized design of our parser made it very easy to add another group of registers.

The last major feature added to the parser was the ability to handle multiple ref files for the same memory space. Despite having this feature requested only a week before our final presentation, we were able to implement this feature by changing the program to call the parsing function inside a loop as opposed to statically. The only problem with this design is if the user specifies multiple ref files that contain registers at the same offset. For example, if two ref files contain a register at offset 0x10 the program will parse both registers, but if a register is requested by offset, the system will assume it is the first register it finds with that offset. This could cause problems if the user was expecting the system to return a different register.

6.1.1.1 Dump File Design

When designing the layout for the dump file, we chose a format that was simple to parse for languages that do not have easy access to regular expressions. Shown in

This appendix describes the many features of NVParse, WinDbg, and the Graphical User Interface. Also, information that is especially important, or behavior that may not be expected, is highlighted in red.

NVParse and WinDbg (NVata)

- Displays register information in the following format:

```
Register Name:    <Register Name>
Port:             <port number>||<Type of Reg>
Register Offset:  <memory offset>
Register Value:   <value in memory>
Register Flags:
<Mask Name>:<Value String / or numerical value>
<Mask Name>:<Value String / or numerical value>
<Mask Name>:<Value String / or numerical value>
<Mask Name>:<Value String / or numerical value>
Red denotes NVParse only
```

- Provides a listing of all devices in hardware along with Bus #, Device #, Function #, Vendor ID, Device ID and a unique identifying ID to be used with other commands. The –dev flag generates this output.
    - NVParse provides human descriptions of the device (aka: "Memory Controller"
    - The presence of this flag overrides any other flags and prevents their execution.
- Allows a base memory address to be specified with the –b <hex offset> command.
    - This offset into memory overrides any memory offset automatically pulled from an AHCI device.
- When given no ID and asked to dump all registers, program defaults to the AHCI device found on the system.
- If no base memory offset is specified, uses BAR 5 as the memory offset if the current ID is an AHCI device.
- Provides standardized "code formatted" output with the –code flag for parsing by the GUI.

- Uses Hexadecimal values for everything except port numbers.
- Allows a particular port to be specified for viewing / writing with the –p # flag.
    - Specifying a number 0-31 returns the port specific registers for that port.
    - Specifying -1 returns all Global registers
    - Specifying -2 returns all PCI Config Space registers
    - **If memory writing is requested for a Port Specific register and no port is specified, an error is returned.**
- Allows a specific register to be specified for viewing / writing with the –n <register name> flag.
    - <register name> can be any register (Global, Port, or PCI)
    - If no port was specified (-p flag), all ports of the given port register are shown.
- Provides a help screen with the -? Command.
- Allows Global and Port specific (but not PCI Config) register values to be written.
    - –n <register name>=<hex> writes the given hex value into the given register, completely overwriting all bits of the register.
    - –n <register name>=<mask>:<value name>,<mask2>:<value name2>,… writes the numerical values behind the value names into the given masks, leaving the rest of the register untouched.
    - –n <register name>=<mask>:<hex>,<mask>:<hex>,… writes the given hex values into the given masks.
- Each flag can only be specified once per execution unless otherwise noted.

NVParse

- Parses a PCI Config Space Ref file (-pci <filename>) and Physical Memory (-f <filename>) ref file into a pre-parsed ref file for use by WinDbg and the GUI.
    - Must have at least one ref file specified (-f <filename> or –pci <filename>) with each execution of the program.
    - Can parse an arbitrary number of either type of ref file if the registers for a device are spread across multiple ref files.
- Allows the creation and execution of a batch file of commands that all run in quick succession.
    - This allows the GUI to send batch write instructions without having to reload the virtual driver repeatedly.

- Allows similar functionality as –n except with a memory offset instead of a register name by using the –o <offset> command.
    - This command also supports the same memory writing options as –n
- Allows the creation of a C style header file with the information pulled from the ref file with the –h <filename> command.
- Allows information about all registers to be dumped with the –all flag.
- Allows #<Port number> to be specified after the register name in a –n declaration. This overrides the port specified by –p.

WinDbg (NVata)

- Allows a specific pre-parsed ref file to be specified for use through the use of the –f <filename> flag
    - Otherwise defaults to “dump_file” in the WinDbg root directory.
- Allows commands to come through a socket connection and pipes the results of those commands back to the requester.
    - Only allows connections from localhost.
    - This facilitates communication with the GUI.
- Defaults to dumping information about all registers
    - -all flag can be specified, but it is redundant.
    - Specifying the –n or –p commands limits what is shown.

GUI Main Screen

- Displays two lists – one contains either Memory registers or PCI registers, while the other displays a specific port’s registers. These are selectable via tabs located above the lists.
    - Each list contains the register name, the port number (if applicable), and the current value of the register.
- When a register is selected from either list, more specific information about that register appears in the bottom section of the display
    - The register name and the current value are displayed. The value may be changed by clicking in the box and typing. This will automatically check to see if it a valid hex number, and update the masks accordingly.
    - Each mask is listed with its name and bit positions.

    - Each mask has an associated dropdown of its possible values.
        - This dropdown allows the user to select a known possible value (displayed as a string followed by the hex value).
        - The User can also type a hex value in the dropdown.
            - This hex value is checked for invalid hex characters and to ensure it is the proper bit width to fit the mask.
                - If it is invalid the value is reverted to the original value.
            - When the box is left the value of the register is updated accordingly..
        - All masks that have been changed are highlighted in green.
    - Changes may be written to a queue using the 'Add to Queue' button.
        - This changes the value displayed in the top section, as well as highlights it in green indicating it is awaiting writing.
        - This button is only enabled when changes have been made to the register.
    - Changes may be written to the register permanently using the 'Write Changes' button.
        - This will only save the currently selected register, and will also update all register information from the system.
        - This will only write the masks that have changed.
        - This button is only enabled when changes have been made to the register.
    - The number of columns used to display the mask information is dynamically determined based on the length of the names of the masks.
- On the right side of the top of the screen, there are four main buttons: 'Refresh All', 'Snapshot', 'Do Diff', and 'Write Queue'.
    - 'Refresh All' will query the system again using the program specified in the configuration, and repopulate all lists, updating the information.
    - 'Snapshot' will create a temporary buffer that stores the current configuration of every register.
        - Using the '…' next to Snapshot allows the user to specify a file to save this information to. This file will be later used by the "Diff …" button.
    - 'Do Diff' loads the information stored in the temporary buffer and compares it to the current values of the registers.
        - Any registers that are different will be highlighted in red.

        - When the user clicks on these values, a new read-only field will appear in the bottom section labeled "Old Value," which is the value that was loaded from the temporary file.
        - Using the '…' next to Do Diff allows the user to select a file created with "Snapshot …" to load, rather than simply using the temporary buffer.
    - 'Write Queue' will write all of the changes that were added to the queue.
        - It will also update all of the register information within the lists. (The same as if you had pressed "Refresh All")
        - This button is not enabled until changes have been added to the queue.
- The menu strip has two main options, File and Edit.
    - File has Exit
        - Exit will exit the program, after prompting the user to make sure they truly wish to exit.
    - Edit has Do Diff, Do Diff…, Snapshot, Snapshot…, and Properties
        - **Do Diff uses the locally saved snapshot to perform a diff with**
        - **Do Diff… uses a file saved snapshot to perform a diff with**
        - **Snapshot creates a locally saved snapshot**
        - **Snapshot… creates a file saved snapshot**
        - Properties has Device and Source options
            - Device will bring up the configuration window with the device screen active
            - Source will bring up the configuration window with the source screen active
- The status bar at the bottom of the screen displays important information about program execution. This includes:
    - Any errors returned by WinDbg or NVParse.
    - A progress bar counting how long until we decide that WinDbg isn't going to get back to us with the requested information
- **INCOMPLETE/MISSING/BROKEN FEATURES**
    - **If you leave a register without saving changes, all changes are lost.**

GUI (Configuration and Setup)

Source

- User can select either NVPCI or WinDBG as the source
- If WinDGB is chosen, the User can select the location for WinDBG, the default is "C:\Program Files\Debugging Tools for Windows". If the user wishes to select a folder, he must select a folder that has windbg.exe in it; otherwise, an error message is shown and the user must try again.

Device

- The system prompts the user with a list of Devices on the target system, including the Vendor ID, Device ID, Chip Name, and the associated ref files, and whether the device is AHCI or not.
    - The chip name and ref files will only be displayed if there is an entry in the systems library file.
    - The user is able to select one device from this list
- The user can specify a ref file(s) for the system to use. If the selected device has a library entry with ref files specified, the system will default to them.
- The user can specify a base offset for the system to use.
    - If the device being used is AHCI, and nothing or "---" is entered, the system will default to using the address in BAR5.
- After the user clicks the OK button, the GUI will populate itself, referencing the necessary configuration data.
- The system will auto generate the dump file that it needs to correctly display the register data. This dump file will also be copied to the WinDbg folder as 'dump_file' (the default pre-parsed ref file WinDbg uses) so WinDbg knows what registers to work with.
- **Missing Features/Bugs**
    - **If NVPCI does not exist, the system will display an error and then close the program.**

Appendix D: Future Features this format specifies one entry per line and delimitates registers, masks, and values with leading tabs (zero, one, or two, respectively). After each register name a new line occurs, which is followed by the register's offset and port number. If the register is a PCI Config register, the port number is specified as -2, if it is a memory space register but not a port register the port number is specified as -1. This data is followed by the associated masks and flags, each containing their associated values. This design gave us a format that was trivial for the GUI and NVATA to parse.

6.1.1.2 C-Style Macros

The C-style macros were designed to be used with the drivers written for NVIDIA's SATA drivers. These macros were designed specifically for use with the AHCI specification. They were created from the parsed ref file and were of a similar structural format to the dump file, where macros for each register existed, and constants for the masks and flags.

6.1.2 NVPCI Wrapper

This section of the program was designed to extend the functionality of NVPCI. NVPCI allows a user to read and write to specific sections of memory. The NVPCI wrapper, however, allows the user to specify a Register name as well as an offset into memory. By using NVParse, the user no longer needs to memorize each register's offset.

6.1.2.1 Command Line Arguments and HCI Choices

The format of NVParse's command line arguments is very complicated, largely due to the many features implemented. The original design of NVParse was based on the command line format of NVPCI, where each argument follows the '-(alphanumeric character)' format, as can be seen in **Error! Reference source not found.**. This appendix shows all the parameters available to NVParse and which ones are also applicable to NVATA.

NVParse, in addition to including NVPCI's error checks, incorporates its own error checks. It will allow the user to specify any string for a value, even if it does not equate to a valid hexadecimal number. In addition, NVParse will allow write calls to any section of memory, even if that memory is not legally accessible by the program. Rather than return an error, the program will simply show the memory again as it normally would, but the new value will not be written. This is a constraint of NVPCI. Since NVPCI allows the user to attempt to write anywhere in memory, there is no way to determine what writes are legal and what writes are not.

6.1.2.2 Functionality

The main functionality of NVParse, besides the parsing of ref files, is to allow the user to specify names of registers, masks, and flags instead of hexadecimal offsets and values. This was implemented by searching the parsed ref file data structure for the specific register name and substituting its name for its offset. This offset is then passed to NVPCI. Originally, the system would do each individual read and write separately; however, this was not only very slow, but also could not be done more than ten times or NVPCI would fail inexplicably. The original work around worked on a lazy access system, where the data would only be fetched when needed. While this temporarily solved the problem, it did not solve the root issue. The solution that was implemented utilized a batch file, created by NVPCI, to complete all of the reads and writes at once. This has the advantage of not only being quicker, but also more reliable, relying on only one call to NVPCI for each series of reads and writes.

6.1.3 WinDbg Extension

The WinDbg extension does a great deal of the same things that our NVPCI wrapper does, however, it uses the low level memory reading commands offered by WinDbg. This has the advantage of allowing the full weight of WinDbg to be combined with this extension. For example, breakpoints can be set and used in the execution of windows or active drivers, low level data structures may be analyzed, and more, all through the capabilities that WinDbg provides the user natively. Our goal was to add as much of the functionality added to NVParse to WinDbg as possible; however, this was difficult with the realization that the WinDbg development environment offers many more constraints than NVPCI does.

6.1.3.1 Implementation

We initially set out designing a well structured object based solution to storing the necessary register – mask – value information as described in Data Structures above. This would have consisted of an object for each type of data, with functions in each for printing, reading/writing memory, and a vector in each to hold that object’s children. We coded a basic implementation of this in Visual Studio, compiled and ran it. It parsed the data file without problem and stored everything without issue. The next step was to integrate this with an existing sample WinDbg extension; however, at this point many different problems arose.

Although everything ran in Visual Studio, nothing would build using the Windows Driver Development kit build environment. As WinDbg extensions need access to kernel level functions, using the Windows Driver Development Kit build environment is required. As such, a large number of operations, such as printing to the console and reading or writing to memory, could not be done except for within the main C++ file. This was only discovered, however, after a good deal of work and the realization that normal C and C++ libraries could not always be used in this environment. This required us to refactor our code to eliminate the usage of strings, vectors, stdio, and nearly anything else besides basic C operators. This finally allowed our program to compile, but we were still encountering runtime exceptions without an obvious cause. This necessitated another refactoring to move nearly all functionality out to the main C++ file, leaving the other objects as dumb data structures. This greatly complicated the main C++ file but simplified the other objects. With these realizations and changes, the extension began to take shape.

Once things were finally working, a final minor round of refactoring was needed to introduce a moderate level of flexibility to allow this extension to be used both from the WinDbg command line and from our graphical user interface while it was running in WinDbg. We decided to use sockets to facilitate this communication due to the constraints of working within WinDbg. For additional information regarding sockets, please see Socket Communication below.

6.1.3.2 Command Line Arguments

The command line arguments for NVATA are very similar to NVParse. These commands are fully documented in **Error! Reference source not found.**. The main difference between the two programs' commands come from NVParse's ability to use Ref files, whereas NVATA uses a dump file. Beyond that, NVATA cannot do some of the things that NVParse can do, such as work from memory offsets or generate dump and C-Style macros.

6.1.3.3 Dump File Parsing

As mentioned above, the dump file format was designed with ease of parsing in mind. The structure of the dump file allowed NVATA to simply read it line by line, creating new objects as it determined each line to be a Register, Mask or Flag. The register objects were then linked into three lists, one for Global, one for PCI, and one for Port registers. As new masks and flags are added they are appended to the most recently added parent object, as the dump file format guarantees we will always get the information in order. Once the dump file has been processed, we are left with a data structure that is easily read whenever information is needed. This data structure is document in more detail within the Data Structures section above.

6.1.3.4 Program Flow

As previously indicated, NVATA is a very linear program. Most of the behavior of the program is determined within dumpnvReal(). This function begins by parsing the string argument passed into it into flag variables which are responsible for dictating the behavior or the later portions of the program. These flags are initially set to cause the program to return as much information as possible. This information, however, may be narrowed down depending on which flags have been set by the user. To ensure that a valid combination of arguments has been provided, the program progresses through a series of error checks which fill in missing flags where appropriate, or return an error to the user if invalid input was provided.

Once the flags have been set, execution passes through three major blocks, one for each of the main functions of NVATA: device listing, writing registers, and reading registers. The device listing section only occurs if it was explicitly requested, or if it has not yet run. We chose this behavior because scanning through all the devices on the system to generate a list of them proved to be very time consuming. If a full device listing has been requested, a function call is made that loops over each function, on each device on each PCI bus. This call populates an ArrayList which holds the device information for later use.

The second major block the program passes through is the writing block. If no write was requested, this block is skipped entirely. If the write command passed to the program was a command where an entire register's new value was specified, a write call is made right away. If the write command was only to write specific masks and not the entire register, the program captures the current value of the register, alters the value to reflect the changes to the masks, and writes the value back.

Finally, the third block the program passes through reads each register of interest and returns information about it. Within this block there are three loops, one for PCI config registers, one for Global registers, and one for Port registers. Depending on what was specified in the command line, the program falls through these loops printing as much as it can without violating any of the parameters passed. The actual printing of the registers varies slightly because how memory is read changes depending on if we are reading PCI Config registers or Global / Port registers. Additionally, how we calculate the memory location varies between Global and Port Registers, necessitating slightly different loops for both, with different memory reading and writing commands for each. Once the program has passed these three stages it returns the results to the caller.

6.1.3.5 Socket Communication

The problem of how to get WinDbg to communicate with a graphical front end was something we became aware of very early in the project. In the beginning, we were told to simply focus on creating a functional command line interface due to the complexity of this task. Once we had the command line interface working, we turned our attention to communication with the GUI. To interact with NVParse, we simply had the GUI call the program and read the results that were returned to stdio. This was not an option for WinDbg, however, as NVATA runs inside of WinDbg and cannot be called directly. The first method we tried turned out to be a successful one: socket communication. Sockets are a very commonly used communication tool for communication between programs on a single computer, as well as between multiple computers (7).

A socket connection is made up of a server and a client, both of which can send and receive messages to and from the other. We setup NVATA to be the server for this connection because in most cases, it would be running before NVestigate would be launched. NVestigate was setup to connect as the client when WinDbg was selected as the source. With the connection established, NVATA simply waits for an instruction to come across the socket. Once an instruction is received, the socket listening code calls the dumpnvReal() function, exactly as direct command line instructions would. The results of this instruction are returned back to the socket handling functions which push the results back out across the socket to the calling connection.

In general this technique worked very well, but it was not without difficulty. We first encountered a problem where our results data was being truncated without obvious reason. After some investigation, it was determined that only one packet of data was successfully making it from program to program. Further investigation revealed that the socket handling code in C# copied one packet worth of data into a buffer before returning a 'data ready' flag. We had initially thought this flag meant all the data was ready, but instead we had to copy all the data into a larger buffer when this flag was returned and repeat the process until we received an 'end of data' flag. Updating the code to handle this ensured that all the data made it to NVestigate from NVATA.

The other issue we encountered when working with sockets revolved around timing. The fact that WinDbg requires a fairly significant amount of time to obtain results from a machine that is being remotely debugged (often over a slow serial cable) meant that results back to the GUI were not instantaneous. This proved to be a problem as the GUI would completely freeze while waiting for results from WinDbg. Also, if WinDbg crashed, or otherwise failed to return data, there was no way for the GUI to recover. To address this issue, we introduced code that polled the 'end of data' flag every second for a configurable number of seconds. During this process, all controls on the GUI were disabled, but the program could still be closed, minimized and moved while it waited. To indicate to the user that the program was simply waiting and had not frozen, we introduced a progress bar to the bottom of the screen. This progress bar incremented each time a poll was made to 'end of data'. If the progress bar becomes full before the data was returned, we declared the connection a failure and prompt the user to try again.

6.2 Graphical User Interface

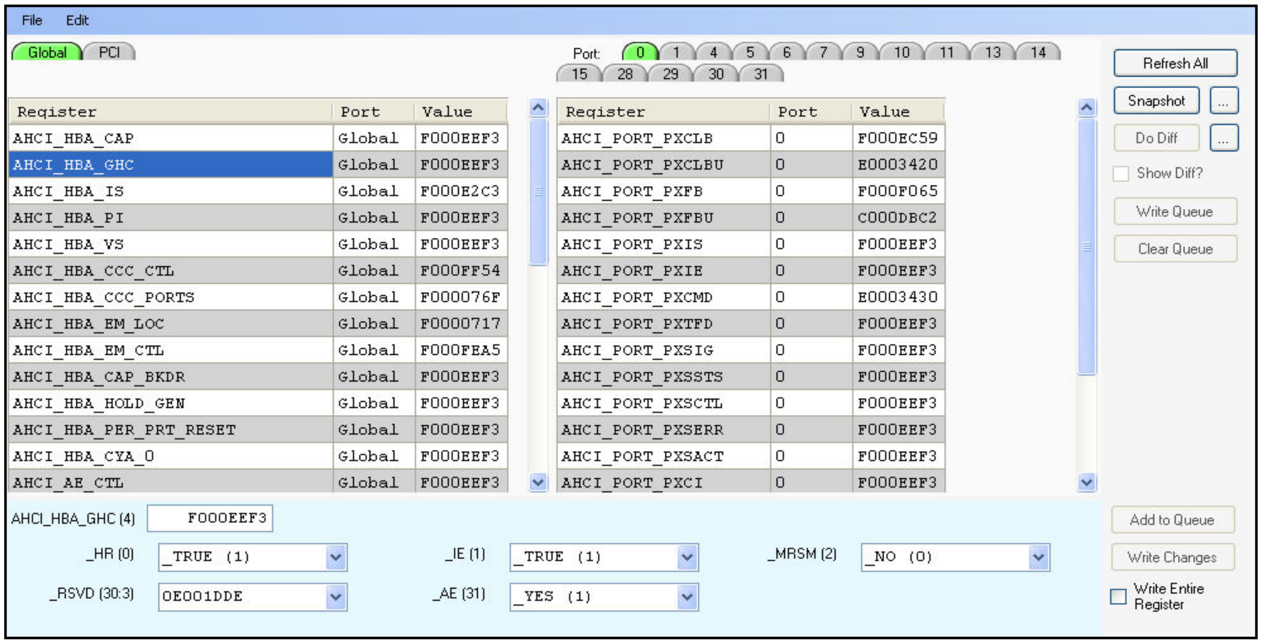
The main goal of the GUI that our team created was to provide a graphical interface for the user to interact with both NVParse and the WinDbg extension. To accomplish this goal, we developed a GUI framework using C#. C# was chosen due to its widespread use and its simplicity in developing GUI environments.

6.2.1 Implementation

While designing the GUI, we implemented the data structures explained within Section 5.1. To accomplish this, we used a system of classes and ArrayLists to store all of the data that was necessary. Additional information was added to the different classes as necessary for the specific needs of the GUI. For instance, a flag was added to the Register class to indicate whether or not it had been altered. Also, another string was added to store the original value of the register before it was changed. These are two examples of the information that was added to our general data structure. This technique allowed the program to stay extensible and also kept the code simple and compact. It also helped us overcome the specific challenges that the GUI presented.

6.2.1.1 Layout

The most important aspect of any User Interface is the layout. We designed the GUI to be as intuitive as possible, and largely succeeded (See Results). To accomplish this goal, we decided to show the Register information in two separate lists on the top half of the screen, as can be seen in Figure 1. Also, these lists only show a limited amount of information in order to keep the screen as uncluttered as possible. The information within the lists may be changed by the user by utilizing the custom tabs located above them. Selecting different tabs will show either the Global or PCI registers, as well as the different ports located within the system. Originally, the two lists were simply split in half; however, later we found this did not provide the user with an easy to read format. Based upon this, the decision was made to alter the technique used to create a more intuitive design.



Register	Port	Value
AHCI_HBA_CAP	Global	F000EEF3
AHCI_HBA_GHC	Global	F000EEF3
AHCI_HBA_IS	Global	F000E2C3
AHCI_HBA_PI	Global	F000EEF3
AHCI_HBA_VS	Global	F000EEF3
AHCI_HBA_CCC_CTL	Global	F000FF54
AHCI_HBA_CCC_PORTS	Global	F000076F
AHCI_HBA_EM_LOC	Global	F0000717
AHCI_HBA_EM_CTL	Global	F000FEA5
AHCI_HBA_CAP_BKDR	Global	F000EEF3
AHCI_HBA_HOLD_GEN	Global	F000EEF3
AHCI_HBA_PER_PRT_RESET	Global	F000EEF3
AHCI_HBA_CYA_0	Global	F000EEF3
AHCI_AE_CTL	Global	F000EEF3

Register	Port	Value
AHCI_PORT_PXCLB	0	F000EC59
AHCI_PORT_PXCLBU	0	E0003420
AHCI_PORT_PXFB	0	F000F065
AHCI_PORT_PXFBU	0	C000DBC2
AHCI_PORT_PXIS	0	F000EEF3
AHCI_PORT_PXIE	0	F000EEF3
AHCI_PORT_PXCMD	0	E0003430
AHCI_PORT_PXTFD	0	F000EEF3
AHCI_PORT_PXSIG	0	F000EEF3
AHCI_PORT_PXSSTS	0	F000EEF3
AHCI_PORT_PXSCTL	0	F000EEF3
AHCI_PORT_PXSERR	0	F000EEF3
AHCI_PORT_PXSACT	0	F000EEF3
AHCI_PORT_PXCI	0	F000EEF3

AHCI_HBA_GHC (4) F000EEF3

_HR (0)	_TRUE (1)	_IE (1)	_TRUE (1)	_MRSM (2)	_NO (0)
_RSVD (30:3)	0E001DDE	_AE (31)	_YES (1)		

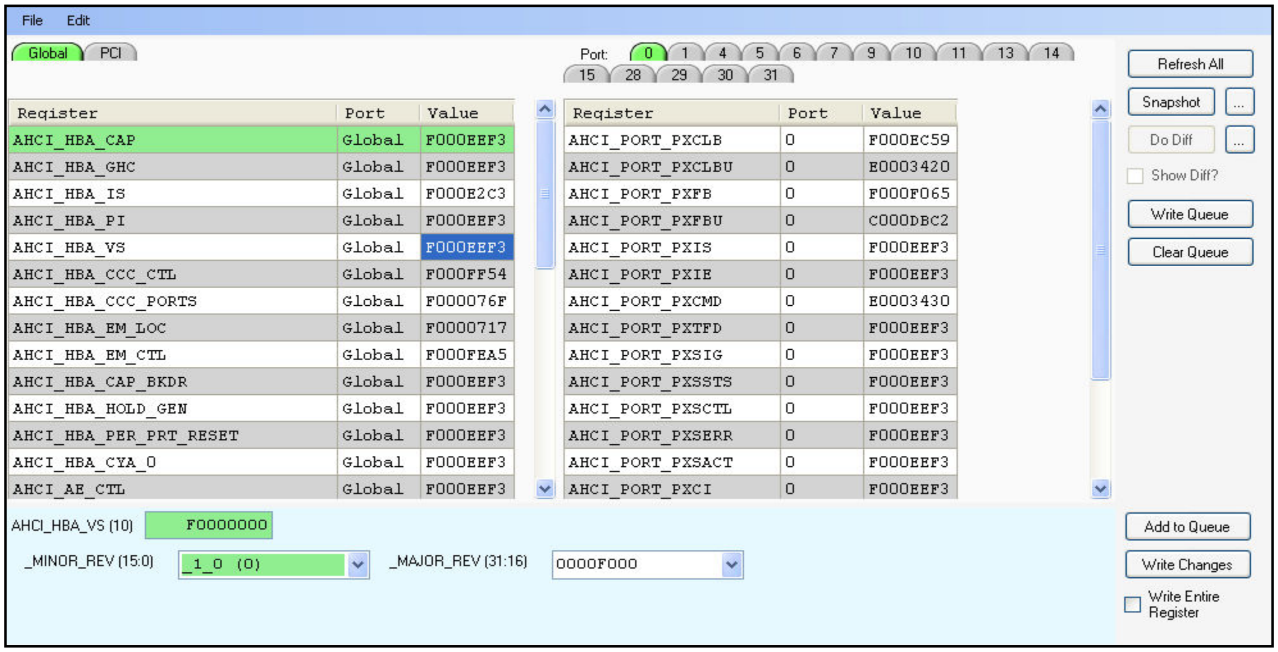
Figure 1: Main Screen of the GUI

This screen also contains buttons on the right to accomplish all of the program's main features. This technique, while not practical for a program with many features, worked perfectly for our program, as there is a fairly limited set of possible actions. We decided that the best way to show the Mask information was to dynamically show it in the bottom half of the program. This way, we would limit the amount of clutter within the screen, while ensuring that only pertinent information is displayed.

Initially, we only displayed the mask names next to a ComboBox populated by the names of the possible flags that the mask could hold. However, after one of the demos, it was suggested that we should also allow the user to input their own value for the mask. Following this suggestion, we added this feature, as well as error checking for this input. Finally, to avoid confusion, we update the displayed register value to match the newly entered values.

6.2.2 Feedback

When a user performs an action it is important for the user to know that this action was received and was acted on. With this in mind we designed the GUI to provide feedback to the user whenever possible. Whenever a value is changed, whether it is a mask value, or a register value, the change is highlighted in green to draw attention to it. Also, when a value is written to the queue, the register is highlighted in green in the list. Both of these features may be seen in Figure 2 below.



Register	Port	Value
AHCI_HBA_CAP	Global	F000EEF3
AHCI_HBA_GHC	Global	F000EEF3
AHCI_HBA_IS	Global	F000E2C3
AHCI_HBA_PI	Global	F000EEF3
AHCI_HBA_VS	Global	F000EEF3
AHCI_HBA_CCC_CTL	Global	F000FF54
AHCI_HBA_CCC_PORTS	Global	F000076F
AHCI_HBA_EM_LOC	Global	F0000717
AHCI_HBA_EM_CTL	Global	F000FEA5
AHCI_HBA_CAP_BKDR	Global	F000EEF3
AHCI_HBA_HOLD_GEN	Global	F000EEF3
AHCI_HBA_PER_PRT_RESET	Global	F000EEF3
AHCI_HBA_CYA_0	Global	F000EEF3
AHCI_AE_CTL	Global	F000EEF3

Register	Port	Value
AHCI_PORT_PXCLB	0	F000EC59
AHCI_PORT_PXCLBU	0	E0003420
AHCI_PORT_PXFB	0	F000F065
AHCI_PORT_PXFBU	0	C000DBC2
AHCI_PORT_PXIS	0	F000EEF3
AHCI_PORT_PXIE	0	F000EEF3
AHCI_PORT_PXCMD	0	E0003430
AHCI_PORT_PXTFD	0	F000EEF3
AHCI_PORT_PXSIG	0	F000EEF3
AHCI_PORT_PXSSTS	0	F000EEF3
AHCI_PORT_PXSCTL	0	F000EEF3
AHCI_PORT_PXSERR	0	F000EEF3
AHCI_PORT_PXSACT	0	F000EEF3
AHCI_PORT_PXCI	0	F000EEF3

AHCI_HBA_VS (10) F0000000

_MINOR_REV (15:0) _1_0 (0) _MAJOR_REV (31:16) 0000F000

Figure 2: The main screen of the GUI with changed registers

As can be seen in Figure 2 above, it remains clear which registers have been altered but have not yet been saved. Also, the error checking we implemented for text areas provides immediate feedback by disabling inappropriate keys. For example, if the value is supposed to be in hexadecimal format, than only characters that are found within a hexadecimal numbers are allowed.

Another technique that was implemented to provide greater feedback was to disable any buttons that were not currently applicable. For example, the "Write Changes" button is disabled until the value of the register has actually changed. This may also be seen in Figure 2 above, as the register that is currently selected has changed, thus enabling both the "Write Changes" and the "Add to Queue" buttons. This ensures that the user does not try to utilize functionality that is not currently available to them.

Socket communication with WinDbg can be slow, especially when WinDbg is connected to a remote computer over a serial connection. With this in mind, when commands are issued to WinDbg that may take a significant amount of time we display a progress bar. This progress bar keeps track of how many times we have checked for results, and when it reaches 100% a timeout occurs indicating that WinDbg is being so slow it may no longer be connected or working. This provides the user with an indication that the program did indeed receive their click and is indeed trying to perform the action requested.

7 Testing

Throughout the design and implementation of our project, we used several methods to test the different sections. These testing methodologies varied based on the requirements and limitations of each section.

7.1 NVParse

The testing for NVParse was automated using a script written in Perl. By automating the tests in this way, we were able to add new tests to the suite easily; while still maintaining the base tests that we created at the beginning of our project. This allowed us to ensure that our code base remained stable and robust. These tests were run each time a significant commit was made for NVParse, assuring that there was always stable code at the head of the tree.

At the beginning of the project, NVParse's test suite consisted of a few tests. These tests ensured that NVParse effectively parsed the ref files into dump files and C macros. As the ability to read and write memory was added to NVParse, the tests had to expand as well. These tests were more difficult to automate. Since the locations of memory that were being written to and read from could change between tests, the information that was being read from had to be independently verified using another program, either NVPCI or WinDBG.

7.2 NVATA

The testing for NVATA was done much differently than the testing for NVParse. Because NVATA is an extension for WinDBG, its tests could not be automated. Since NVATA can only be used from the WinDbg command line; our testing methodology involved copying and pasting pre-written command line arguments into WinDBG and checking the results. While these tests were neither fast nor efficient, they were the only way we could accurately and systematically test the features of NVATA. These tests, in their general form, have been included in This appendix describes each of the buttons that may be found in the main screen of the GUI. The description includes information about how the button works, what it does, and also how to test it. Any information that could be important, such as unexpected behavior, is also noted.

Refresh All

- How it works
  - Repopulates every list with the current register information

- How to test
    1. Change a writable register value using NVParse
    2. Click on ‘Refresh All’
    3. Verify that the value changed within the GUI

Snapshot

- How it works
    - Creates a local save of the current register information within a string
    - Formatted the same as the information that is read into the GUI (Register info –code format in Formats document)
- How to test
    1. Click on ‘Snapshot’
    2. Use debugging tools to verify the information contained within the ‘snapshot’ variable in topSectionEvents.cs

… (Next to Snapshot)

- How it works
    - Creates a file save of the current register information
    - Formatted the same as the information that is read into the GUI (Register info –code format in Formats document)
    - Also creates a local save of the snapshot
- How to test
    1. Click on ‘…’
    2. Save into a file
    3. Using a text editor, open the file, and verify the contents

Diff

- How it works
    - Loads the last local save of the snapshot
    - Loads all the information into the “oldValue” variable within each register object in allInformation
    - Using the cell formatting of InfoGridLeft and InfoGridRight, the cells are changed red if “oldValue” and “value” are not equal

  - The old mask values are shown in red next to the mask name in the bottom section if they changed. These are determined by using “oldValue”, which is also displayed next to the current value
- How to test
  1. Create a local snapshot
  2. Change at least one register value
  3. Click on ‘Diff’
  4. The changed register should now be highlighted in red, and when clicked on, should display the normal information on the bottom along with the old value for the register, as well as the old value of each mask

… (Next to Diff)

- How it works
  - Loads a save from a file
  - Loads all the information into the “oldValue” variable within each register within allInformation
  - Using the cell formatting of InfoGridLeft and InfoGridRight, the cells are changed red if “oldValue” and “value” are not equal
  - The old mask values are shown in red next to the mask name in the bottom section if they changed. These are determined by using “oldValue”, which is also displayed next to the current value
- How to test
  1. Save a snapshot into a file
  2. Change at least one register value
  3. Click on ‘Diff’
  4. The changed register should now be highlighted in red, and when clicked on, should display the normal information on the bottom along with the old value for the register, as well as the old value of each mask

Show Diff?

- How it works
  - After a Diff has been done, it becomes checked and the cells that are different are highlighted in red

    - When unchecked, the cells are no longer highlighted red
    - A simple if statement within the cell formatting event for InfoGridLeft and InfoGridRight controls this functionality
- How to test
    1. Perform a Diff – the changed cells should be highlighted red
    2. Uncheck the box – the changed cells should no longer be highlighted red
    3. Check the box again – the changed cells should be highlighted red

Write Queue

- How it works
    - Writes all of the registers in “changedregisters” ArrayList within BottomSectionEvents
- How to test
    1. Add at least one register to the queue (best would be to add one register with changed masks, one with a changed value, and one with the entire register)
    2. Verify that the registers changed to reflect the changes appropriately
    3. Note: The program will not tell you if a register is read-only, but the write will fail. Also, WinDbg will not write to any PCI register, regardless of whether or not it is writable. Do not mistake this as a bug.

Clear Queue

- How it works
    - Clears the “changedregisters” ArrayList within BottomSectionEvents
- How to test
    1. Change a register and add it to the queue – it should now be highlighted green
    2. Clear the queue – it should no longer be highlighted green
    3. Write the queue – nothing should change

Add to Queue

- How it works
    - If Write Entire Register is not checked, adds the changed masks to the “changedregisters” ArrayList within BottomSectionEvent
    - If Write Entire Register is checked, adds all of the masks to the “changedregisters” ArrayList within BottomSectionEvent
- How to test

1. Add at least one register to the queue (best would be to add one register with changed masks, one with a changed value, and one with the entire register)
2. Verify that the registers changed to reflect the changes appropriately
3. Note: The program will not tell you if a register is read-only, but the write will fail. Do not mistake this for a bug.

Write Changes

- How it works
  - If Write Entire Register is not checked, immediately writes the changed masks
  - If Write Entire Register is checked, immediately writes all of the masks
  - Note: In either method, the write queue will be destroyed
- How to test
  1. Change a register
  2. Write the changes
  3. The list should now reflect the change
  4. Note: The program will not tell you if a register is read-only, but the write will fail. Also, WinDbg will not write to any PCI register, regardless of whether or not it is writable. Do not mistake this for a bug.

Write Entire Register

- How it works
  - If checked, when a Register is written (whether to the queue, or written immediately), the entire Register is written
  - If unchecked, when a Register is written (whether to the queue, or written immediately), only the changed masks are written
- How to test
  1. Change a few masks within a Register
  2. Change the Register value completely using NVPCI or NVATA
  3. Write the changes with the checkbox unchecked
  4. Verify that the only the masks that you meant to change were changed
  5. Change a few masks within a Register
  6. Change the Register value completely using NVPCI or NVATA
  7. Write the changes with the checkbox checked

8. Verify that all of the masks were changed to what you wrote
9. Note: The program will not tell you if a register is read-only, but the write will fail. Do not mistake this for a bug.

Port and Global/PCI Buttons

- How they work
    - Change what should currently be displayed, and repopulate the visible lists
    - Note: Changing what is displayed does not refresh the data
- How to test
    1. Click on one that is not currently selected
    2. The display should change to reflect your choice

Appendix G: Command Line Testing for future developers to reference.

7.3 GUI

Testing for a GUI is never an easy prospect. While tools exist to automate GUI testing, the ever changing nature of project and our fast development cycles did not allow for easy GUI test automation. Instead, we designed a high level mockup of our GUI and developed a series of manual tests to execute on each section of the GUI. These tests were designed to make sure each section of the GUI stayed functional and performed fast enough as to not limit its functionality. These tests were combined into the document seen in This appendix provides examples of the different formats that are used throughout the NVestigate software suite. These examples are also described in detail to provide a thorough understanding.

Dump File

No comments are allowed in the actual file.

```
AHCI_PORT_MP                 //Register Name
17C                          //Memory Offset
0                            //0=Port Register, -1=Global, -2= PCI
[tab]   _MPA                    //Mask name
80                              //Bit mask
[tab]   [tab]    _SET                 //Flag name
```

```
1                                   //Value of Flag
[tab]   [tab]    _DEFAULT           //Next Flag
0
[tab]   [tab]    _CLEAR
0
[tab]   _MPRSVD_1             //Next mask
40
[tab]   [tab]    _0
0
[tab]   _MPIS
20
[tab]   [tab]    _ZERO
0
[tab]   [tab]    _SET
1
[tab]   [tab]    _CLEAR
1
AHCI_HBA_CAP                  //Next Register
//...
```

Library File

No comments are allowed in the actual file. Multiple PCI and Global/Port ref files can be specified for a given device by listing each complete path on the appropriate line separated by white space.

```
__BEGIN_ENTRY__                           //Entry header
10DE                                      //Vendor ID
044C                                      //Device ID
MCP65                                     //Device Name
mcp65\manuals\dev_nv_proj__fpci_sata.ref  //PCI Ref files
mcp65\manuals\dev_nv_proj__sata.ref       //Global / Port ref files
```

Register Information –code

This output is returned when a command requesting register information is given including the –code flag. Comments are not part of output.

```
__BEGIN_REGISTER__    //Header at start of each register
AHCI_HBA_CAP          //Register Name
-1                    //-1=Global, -2= PCI, 0<= Port number
00000000              //Memory Offset
```

```
E7229F03              //Value in memory
_NP:_4                //Mask:Flag
_ABCD:4D              //Mask:Hex Value (if no flag found)
_SXS:_FALSE           //And so on for each mask
```

Device Listing (-dev -code)

This is the output format for a –dev –code command.

```
__PCI_CONFIG_DATA__      //Header once at top of output
78:1022:1103             //One device per line
58:1022:1102             //ID:Vendor ID:Device ID
*9:10DE:0554             //Preceded by a * if the device is AHCI
38:1022:1101             //And so on...
```

Appendix F: GUI Button Tests for use by future developers.

8 Results

The most important result of our project was the NVestigate software suite, including the newly developed NVParse, the GUI, and the vastly updated NVATA extension. We managed to complete the primary goal of our project, namely the ability to read and write registers by name instead of by memory offset. In addition to this, we completed many additional goals that were added as the project continued, including the creation of a GUI that could interact with both NVParse and NVATA, the ability to interact with PCI configuration space, the ability to perform snapshots and view differentials within the GUI, and the capability to install and uninstall our program using a single, unified installer.

While the main result of the project was NVestigate as a whole, including the three different programs, we also created a large amount of documentation for NVIDIA to use after we have left. Appendix C: Features contains the feature list of all three of the programs. Appendix A: User Documentation describes the GUI, detailing how to install the program, as well as step by step instructions for many of its common uses. A list of all of the command line syntax for WinDbg and NVParse may be found within **Error! Reference source not found.**. This appendix also contains many different examples of how the command line interfaces may be used. These three appendices are the components of a user's guide that we developed for NVIDIA.

In addition to the user guide, we also created a developer's guide containing much more detailed information about the different programs. This guide contains the Application Programming Interface (API) for both NVATA and the GUI (html documents included with report).

Throughout this project we have strived to keep this code as extendable as possible. We have accomplished this through designing clean, well documented, object oriented code. We have ensured that each function only performs on major task and is reusable when other aspects of the program may need the same behavior. If the program modifications must be made, the HTML help file, the in code documentation, and this document are useful resources.

Non-AHCI Device Support

One of the early requests for extensibility was to make this program work for more devices than just AHCI devices. This additional feature is largely implemented already; however, additional work is required to ensure that it works flawlessly.

Currently, NVATA (WinDbg) and NVParse will return values for all the port registers specified in the given ref file for as many ports as it finds in AHCI_HBA_PI (offset 0xC) regardless of if the device being looked at is AHCI or not. This technique, however, does not work correctly if the device is non-AHCI. In this case, the offset 0xC may not necessarily be the correct location to find the number of ports. To change this, the following general steps would need to be implemented:

- The location of the proper register containing this information should be encoded into the Library entry for each device.
- NVParse should be extended to build this information into the dump file.
- WinDbg and NVParse should be extended to use this information when determining what register to check for ports.
- The GUI should be modified to not display a list for port information if no ports exist in the dump file.

Additional effort may be required, however, if the current method of specifying which ports are currently in use is changed.

Finally, global and port specific registers are currently differentiated by their memory offset. Both NVParse and NVATA declare a register to be a port specific register if its offset is above 0x100 which may not hold true for other non-AHCI devices.

Network-Redundant Library and Ref Files

Over time, the original Library file, as well as the ref files, which were installed automatically, could become out of date as devices are added or modified. An easy technique to maintain these files across many users is to store the files centrally on a file share. The program currently supports reading the library file and ref files from the network. The space for improvement upon this comes in the case where the network is not available at run time. Additional code could be added to the GUI to allow it to take two parameters in config.ini, one for a network location for the files, and one for a local copy of the files. Also, a simple mechanism could be implemented that ensures that the most up-to-date files are copied locally when a network connection is available.

NVParse Efficiency Improvements

Currently every command to NVParse requires specifying at least one ref file. This ref file is parsed into a local data structure and then transformed to a dump file and/or used to perform memory reads and writes. The act of parsing the ref file is fairly processor intensive, and the act of fetching the ref file from a network drive may also be time consuming. A possible improvement would involve modifying NVParse to enable parsing a dump file. This file would be significantly easier to parse, and could be cached locally between calls. This modification would need to implement a function similar to an existing function, parse(), and provide a mechanism for choosing between ref file parsing and dump file parsing.

Indicate Read Only Registers

A feature requested later in the development of the program was the ability to show which registers are read only. This information is currently stored in the ref files, but is ignored by the parsing function. A modification to NVParse would have to be made to extract this information from the comment after each register and store it within the dump file. This information would have to then be pulled from the dump file by the GUI and the display updated accordingly.

Allow Sorting of Registers

The list of registers shown in the main view is currently shown in the same order in which the registers were found in the ref file. This is not a terrible solution, as this order is generally logical and is often the order that the registers exist in memory. However, it would be a nice feature to allow the user to sort in accordance to their data. For example, to sort the lists alphabetically to assist in finding a specific register more quickly.

The list of registers shown is actually an ArrayList of Registers displayed in a C# DataGridView object. ArrayLists do not have the ability to be sorted out of the box, but modifying this list to be a binding list and implementing the SortColumns property and Sort() family of functions would allow the list to be sorted. There are several other possible ways of sorting a DataGridView, and it may be found that another approach is easier; we would recommend some research be done prior to attempting this change.

BUGS

Throughout the project, diligence was shown in testing for bugs; however, as with any project, a few are known to exist. Included below is a list of the known bugs within the program, as well as suggestions for correcting them in the future.

Refresh With Items in Queue

This bug may be recreated by following these steps:

1. Select a register
2. Change the register value and add it to the queue
3. Refresh the list (either through the 'Refresh All' button or writing another register value)
4. The register that was changed will no long be highlighted in green, and will not display the NEW VALUE field when clicked on, but it will still be written when 'Write Queue' is clicked on

The reason this bug exists is because the field that is responsible to control the display of NEW VALUE and highlight the register value green is oldValue within the Register class. This is over-written, however, when the lists are refreshed. This sets oldValue back to null, and eliminates the display of the new value; however, because the queue is also kept in a separate ArrayList, it is not destroyed there and may still be written.

This may be fixed by adding an additional field to the Register class, perhaps entitled newValue, to store the value that is currently being held within the queue. Than, within the cellFormatting event in topSectionEvents, change the calls to match this new functionality. Also, within displayBottomLeft() in BottomSection, change the display calls to match this change. Finally, when a refresh is done, the changedRegisters ArrayList within BottomSection should be looped through, and for each Register within it set the corresponding newValue in Register within allInformation (found within topSection).

Queue and Diff Conflict

This bug occurs when a register exists in the queue, waiting to be written, and the user performs a diff in which the same register exists.

To recreate this bug, follow these steps:

1. Create a snapshot
2. Write a change to a register
3. For the same register, add a change to the queue
4. Load the diff

This shows that only the diff is displayed, and not the queue. If show diff is turned off, the green will be shown again, however the value within NEW VALUE will be the value loaded for the diff. Another aspect of this bug is that if a new value is added to the queue for this same register, and the diff is being shown, than the OLD VALUE field will change to reflect this change. This bug may be fixed following the same steps listed above.

Appendix E: Formats, describes the different file formats that are used throughout NVestigate. A list of possible future features is included within

This appendix describes the many features of NVParse, WinDbg, and the Graphical User Interface. Also, information that is especially important, or behavior that may not be expected, is highlighted in red.

NVParse and WinDbg (NVata)

- Displays register information in the following format:

```
Register Name:     <Register Name>
Port:              <port number>||<Type of Reg>
Register Offset:   <memory offset>
Register Value:    <value in memory>
Register Flags:
<Mask Name>:<Value String / or numerical value>
<Mask Name>:<Value String / or numerical value>
<Mask Name>:<Value String / or numerical value>
<Mask Name>:<Value String / or numerical value>
Red denotes NVParse only
```

- Provides a listing of all devices in hardware along with Bus #, Device #, Function #, Vendor ID, Device ID and a unique identifying ID to be used with other commands. The –dev flag generates this output.
    - NVParse provides human descriptions of the device (aka: “Memory Controller”
    - The presence of this flag overrides any other flags and prevents their execution.
- Allows a base memory address to be specified with the –b <hex offset> command.
    - This offset into memory overrides any memory offset automatically pulled from an AHCI device.
- When given no ID and asked to dump all registers, program defaults to the AHCI device found on the system.
- If no base memory offset is specified, uses BAR 5 as the memory offset if the current ID is an AHCI device.
- Provides standardized “code formatted” output with the –code flag for parsing by the GUI.
- Uses Hexadecimal values for everything except port numbers.

- Allows a particular port to be specified for viewing / writing with the –p # flag.
  - Specifying a number 0-31 returns the port specific registers for that port.
  - Specifying -1 returns all Global registers
  - Specifying -2 returns all PCI Config Space registers
  - **If memory writing is requested for a Port Specific register and no port is specified, an error is returned.**
- Allows a specific register to be specified for viewing / writing with the –n <register name> flag.
  - <register name> can be any register (Global, Port, or PCI)
  - If no port was specified (-p flag), all ports of the given port register are shown.
- Provides a help screen with the -? Command.
- Allows Global and Port specific (but not PCI Config) register values to be written.
  - –n <register name>=<hex> writes the given hex value into the given register, completely overwriting all bits of the register.
  - –n <register name>=<mask>:<value name>,<mask2>:<value name2>,… writes the numerical values behind the value names into the given masks, leaving the rest of the register untouched.
  - –n <register name>=<mask>:<hex>,<mask>:<hex>,… writes the given hex values into the given masks.
- Each flag can only be specified once per execution unless otherwise noted.

NVParse

- Parses a PCI Config Space Ref file (-pci <filename>) and Physical Memory (-f <filename>) ref file into a pre-parsed ref file for use by WinDbg and the GUI.
  - Must have at least one ref file specified (-f <filename> or –pci <filename>) with each execution of the program.
  - Can parse an arbitrary number of either type of ref file if the registers for a device are spread across multiple ref files.
- Allows the creation and execution of a batch file of commands that all run in quick succession.
  - This allows the GUI to send batch write instructions without having to reload the virtual driver repeatedly.
- Allows similar functionality as –n except with a memory offset instead of a register name by using the –o <offset> command.

    - This command also supports the same memory writing options as –n
- Allows the creation of a C style header file with the information pulled from the ref file with the –h <filename> command.
- Allows information about all registers to be dumped with the –all flag.
- Allows #<Port number> to be specified after the register name in a –n declaration. This overrides the port specified by –p.

WinDbg (NVata)

- Allows a specific pre-parsed ref file to be specified for use through the use of the –f <filename> flag
    - Otherwise defaults to "dump_file" in the WinDbg root directory.
- Allows commands to come through a socket connection and pipes the results of those commands back to the requester.
    - Only allows connections from localhost.
    - This facilitates communication with the GUI.
- Defaults to dumping information about all registers
    - -all flag can be specified, but it is redundant.
    - Specifying the –n or –p commands limits what is shown.

GUI Main Screen

- Displays two lists – one contains either Memory registers or PCI registers, while the other displays a specific port's registers. These are selectable via tabs located above the lists.
    - Each list contains the register name, the port number (if applicable), and the current value of the register.
- When a register is selected from either list, more specific information about that register appears in the bottom section of the display
    - The register name and the current value are displayed. The value may be changed by clicking in the box and typing. This will automatically check to see if it a valid hex number, and update the masks accordingly.
    - Each mask is listed with its name and bit positions.
    - Each mask has an associated dropdown of its possible values.
        - This dropdown allows the user to select a known possible value (displayed as a string followed by the hex value).

        - The User can also type a hex value in the dropdown.
            - This hex value is checked for invalid hex characters and to ensure it is the proper bit width to fit the mask.
                - If it is invalid the value is reverted to the original value.
            - When the box is left the value of the register is updated accordingly..
        - All masks that have been changed are highlighted in green.
    - Changes may be written to a queue using the 'Add to Queue' button.
        - This changes the value displayed in the top section, as well as highlights it in green indicating it is awaiting writing.
        - This button is only enabled when changes have been made to the register.
    - Changes may be written to the register permanently using the 'Write Changes' button.
        - This will only save the currently selected register, and will also update all register information from the system.
        - This will only write the masks that have changed.
        - This button is only enabled when changes have been made to the register.
    - The number of columns used to display the mask information is dynamically determined based on the length of the names of the masks.
- On the right side of the top of the screen, there are four main buttons: 'Refresh All', 'Snapshot', 'Do Diff', and 'Write Queue'.
    - 'Refresh All' will query the system again using the program specified in the configuration, and repopulate all lists, updating the information.
    - 'Snapshot' will create a temporary buffer that stores the current configuration of every register.
        - Using the '…' next to Snapshot allows the user to specify a file to save this information to. This file will be later used by the "Diff …" button.
    - 'Do Diff' loads the information stored in the temporary buffer and compares it to the current values of the registers.
        - Any registers that are different will be highlighted in red.
        - When the user clicks on these values, a new read-only field will appear in the bottom section labeled "Old Value," which is the value that was loaded from the temporary file.

        - Using the '…' next to Do Diff allows the user to select a file created with "Snapshot …" to load, rather than simply using the temporary buffer.
    - 'Write Queue' will write all of the changes that were added to the queue.
        - It will also update all of the register information within the lists. (The same as if you had pressed "Refresh All")
        - This button is not enabled until changes have been added to the queue.
- The menu strip has two main options, File and Edit.
    - File has Exit
        - Exit will exit the program, after prompting the user to make sure they truly wish to exit.
    - Edit has Do Diff, Do Diff…, Snapshot, Snapshot…, and Properties
        - **Do Diff uses the locally saved snapshot to perform a diff with**
        - **Do Diff… uses a file saved snapshot to perform a diff with**
        - **Snapshot creates a locally saved snapshot**
        - **Snapshot… creates a file saved snapshot**
        - Properties has Device and Source options
            - Device will bring up the configuration window with the device screen active
            - Source will bring up the configuration window with the source screen active
- The status bar at the bottom of the screen displays important information about program execution. This includes:
    - Any errors returned by WinDbg or NVParse.
    - A progress bar counting how long until we decide that WinDbg isn't going to get back to us with the requested information
- **INCOMPLETE/MISSING/BROKEN FEATURES**
    - **If you leave a register without saving changes, all changes are lost.**

GUI (Configuration and Setup)

Source

- User can select either NVPCI or WinDBG as the source

- If WinDGB is chosen, the User can select the location for WinDBG, the default is "C:\Program Files\Debugging Tools for Windows". If the user wishes to select a folder, he must select a folder that has windbg.exe in it; otherwise, an error message is shown and the user must try again.

Device

- The system prompts the user with a list of Devices on the target system, including the Vendor ID, Device ID, Chip Name, and the associated ref files, and whether the device is AHCI or not.
    - The chip name and ref files will only be displayed if there is an entry in the systems library file.
    - The user is able to select one device from this list
- The user can specify a ref file(s) for the system to use. If the selected device has a library entry with ref files specified, the system will default to them.
- The user can specify a base offset for the system to use.
    - If the device being used is AHCI, and nothing or "---" is entered, the system will default to using the address in BAR5.
- After the user clicks the OK button, the GUI will populate itself, referencing the necessary configuration data.
- The system will auto generate the dump file that it needs to correctly display the register data. This dump file will also be copied to the WinDbg folder as 'dump_file' (the default pre-parsed ref file WinDbg uses) so WinDbg knows what registers to work with.
- **Missing Features/Bugs**
    - **If NVPCI does not exist, the system will display an error and then close the program.**

Appendix D: Future Features, along with a short description of how they may be implemented. Finally, within This appendix provides examples of the different formats that are used throughout the NVestigate software suite. These examples are also described in detail to provide a thorough understanding.

Dump File

No comments are allowed in the actual file.

```
AHCI_PORT_MP                //Register Name
17C                         //Memory Offset
0                           //0=Port Register, -1=Global, -2= PCI
```

```
[tab]   _MPA                    //Mask name
80                              //Bit mask
[tab]   [tab]    _SET                 //Flag name
1                                     //Value of Flag
[tab]   [tab]    _DEFAULT             //Next Flag
0
[tab]   [tab]    _CLEAR
0
[tab]   _MPRSVD_1               //Next mask
40
[tab]   [tab]    _0
0
[tab]   _MPIS
20
[tab]   [tab]    _ZERO
0
[tab]   [tab]    _SET
1
[tab]   [tab]    _CLEAR
1
AHCI_HBA_CAP                    //Next Register
//...
```

Library File

No comments are allowed in the actual file. Multiple PCI and Global/Port ref files can be specified for a given device by listing each complete path on the appropriate line separated by white space.

```
__BEGIN_ENTRY__                           //Entry header
10DE                                      //Vendor ID
044C                                      //Device ID
MCP65                                     //Device Name
mcp65\manuals\dev_nv_proj__fpci_sata.ref  //PCI Ref files
mcp65\manuals\dev_nv_proj__sata.ref       //Global / Port ref files
```

Register Information –code

This output is returned when a command requesting register information is given including the –code flag. Comments are not part of output.

```
__BEGIN_REGISTER__   //Header at start of each register
```

```
AHCI_HBA_CAP          //Register Name
-1                    //-1=Global, -2= PCI, 0<= Port number
00000000              //Memory Offset
E7229F03              //Value in memory
_NP:_4                //Mask:Flag
_ABCD:4D              //Mask:Hex Value (if no flag found)
_SXS:_FALSE           //And so on for each mask
```

Device Listing (-dev -code)

This is the output format for a –dev –code command.

```
__PCI_CONFIG_DATA__      //Header once at top of output
78:1022:1103             //One device per line
58:1022:1102             //ID:Vendor ID:Device ID
*9:10DE:0554             //Preceded by a * if the device is AHCI
38:1022:1101             //And so on...
```

Appendix F: GUI Button Tests, is a list of each GUI button, along with how it works and how to test it was created to facilitate testing in the future. A similar list for the command line interfaces is located within This appendix describes each of the buttons that may be found in the main screen of the GUI. The description includes information about how the button works, what it does, and also how to test it. Any information that could be important, such as unexpected behavior, is also noted.

Refresh All

- How it works
    - Repopulates every list with the current register information
- How to test
    4. Change a writable register value using NVParse
    5. Click on 'Refresh All'
    6. Verify that the value changed within the GUI

Snapshot

- How it works
    - Creates a local save of the current register information within a string
    - Formatted the same as the information that is read into the GUI (Register info –code format in Formats document)
- How to test

3. Click on ‘Snapshot’
4. Use debugging tools to verify the information contained within the ‘snapshot’ variable in topSectionEvents.cs

… (Next to Snapshot)

- How it works
  - Creates a file save of the current register information
  - Formatted the same as the information that is read into the GUI (Register info –code format in Formats document)
  - Also creates a local save of the snapshot
- How to test
  4. Click on ‘…’
  5. Save into a file
  6. Using a text editor, open the file, and verify the contents

Diff

- How it works
  - Loads the last local save of the snapshot
  - Loads all the information into the “oldValue” variable within each register object in allInformation
  - Using the cell formatting of InfoGridLeft and InfoGridRight, the cells are changed red if “oldValue” and “value” are not equal
  - The old mask values are shown in red next to the mask name in the bottom section if they changed. These are determined by using “oldValue”, which is also displayed next to the current value
- How to test
  5. Create a local snapshot
  6. Change at least one register value
  7. Click on ‘Diff’
  8. The changed register should now be highlighted in red, and when clicked on, should display the normal information on the bottom along with the old value for the register, as well as the old value of each mask

… (Next to Diff)

- How it works
  - Loads a save from a file
  - Loads all the information into the “oldValue” variable within each register within allInformation
  - Using the cell formatting of InfoGridLeft and InfoGridRight, the cells are changed red if “oldValue” and “value” are not equal
  - The old mask values are shown in red next to the mask name in the bottom section if they changed. These are determined by using “oldValue”, which is also displayed next to the current value
- How to test
  5. Save a snapshot into a file
  6. Change at least one register value
  7. Click on ‘Diff’
  8. The changed register should now be highlighted in red, and when clicked on, should display the normal information on the bottom along with the old value for the register, as well as the old value of each mask

Show Diff?

- How it works
  - After a Diff has been done, it becomes checked and the cells that are different are highlighted in red
  - When unchecked, the cells are no longer highlighted red
  - A simple if statement within the cell formatting event for InfoGridLeft and InfoGridRight controls this functionality
- How to test
  4. Perform a Diff – the changed cells should be highlighted red
  5. Uncheck the box – the changed cells should no longer be highlighted red
  6. Check the box again – the changed cells should be highlighted red

Write Queue

- How it works
  - Writes all of the registers in “changedregisters” ArrayList within BottomSectionEvents

- How to test
    4. Add at least one register to the queue (best would be to add one register with changed masks, one with a changed value, and one with the entire register)
    5. Verify that the registers changed to reflect the changes appropriately
    6. Note: The program will not tell you if a register is read-only, but the write will fail. Also, WinDbg will not write to any PCI register, regardless of whether or not it is writable. Do not mistake this as a bug.

Clear Queue

- How it works
    - Clears the "changedregisters" ArrayList within BottomSectionEvents
- How to test
    4. Change a register and add it to the queue – it should now be highlighted green
    5. Clear the queue – it should no longer be highlighted green
    6. Write the queue – nothing should change

Add to Queue

- How it works
    - If Write Entire Register is not checked, adds the changed masks to the "changedregisters" ArrayList within BottomSectionEvent
    - If Write Entire Register is checked, adds all of the masks to the "changedregisters" ArrayList within BottomSectionEvent
- How to test
    4. Add at least one register to the queue (best would be to add one register with changed masks, one with a changed value, and one with the entire register)
    5. Verify that the registers changed to reflect the changes appropriately
    6. Note: The program will not tell you if a register is read-only, but the write will fail. Do not mistake this for a bug.

Write Changes

- How it works
    - If Write Entire Register is not checked, immediately writes the changed masks
    - If Write Entire Register is checked, immediately writes all of the masks
    - Note: In either method, the write queue will be destroyed

- How to test
    5. Change a register
    6. Write the changes
    7. The list should now reflect the change
    8. Note: The program will not tell you if a register is read-only, but the write will fail. Also, WinDbg will not write to any PCI register, regardless of whether or not it is writable. Do not mistake this for a bug.

Write Entire Register

- How it works
    - If checked, when a Register is written (whether to the queue, or written immediately), the entire Register is written
    - If unchecked, when a Register is written (whether to the queue, or written immediately), only the changed masks are written
- How to test
    10. Change a few masks within a Register
    11. Change the Register value completely using NVPCI or NVATA
    12. Write the changes with the checkbox unchecked
    13. Verify that the only the masks that you meant to change were changed
    14. Change a few masks within a Register
    15. Change the Register value completely using NVPCI or NVATA
    16. Write the changes with the checkbox checked
    17. Verify that all of the masks were changed to what you wrote
    18. Note: The program will not tell you if a register is read-only, but the write will fail. Do not mistake this for a bug.

Port and Global/PCI Buttons

- How they work
    - Change what should currently be displayed, and repopulate the visible lists
    - Note: Changing what is displayed does not refresh the data
- How to test
    3. Click on one that is not currently selected
    4. The display should change to reflect your choice

Appendix G: Command Line Testing.

The results of this project as a whole cannot be easily quantified, especially this soon after it was released to its end users. We were able to see as mall glimpse of the results during our beta demo. During the demo, a group of the same programmers who will be using this program were given a chance to use it for themselves. After a short demonstration, they were able to sit at the program and use it as designed without further instruction. The GUI was intuitive enough that they were able to use it to manipulate the register values and see results in the system indicative of the registers they had changed. As a glimpse of future results, the results of the beta were very encouraging.

8.1 Future Features

Throughout development of this project we encountered several minor features that we determined would be desirable to have, but were outside the scope of this project. The most important of these are documented in an extensive file included in

This appendix describes the many features of NVParse, WinDbg, and the Graphical User Interface. Also, information that is especially important, or behavior that may not be expected, is highlighted in red.

NVParse and WinDbg (NVata)

- Displays register information in the following format:

```
Register Name:    <Register Name>
Port:             <port number>||<Type of Reg>
Register Offset:  <memory offset>
Register Value:   <value in memory>
Register Flags:
<Mask Name>:<Value String / or numerical value>
<Mask Name>:<Value String / or numerical value>
<Mask Name>:<Value String / or numerical value>
<Mask Name>:<Value String / or numerical value>
Red denotes NVParse only
```

- Provides a listing of all devices in hardware along with Bus #, Device #, Function #, Vendor ID, Device ID and a unique identifying ID to be used with other commands. The –dev flag generates this output.
    - NVParse provides human descriptions of the device (aka: "Memory Controller"
    - The presence of this flag overrides any other flags and prevents their execution.
- Allows a base memory address to be specified with the –b <hex offset> command.
    - This offset into memory overrides any memory offset automatically pulled from an AHCI device.
- When given no ID and asked to dump all registers, program defaults to the AHCI device found on the system.
- If no base memory offset is specified, uses BAR 5 as the memory offset if the current ID is an AHCI device.

- Provides standardized “code formatted” output with the –code flag for parsing by the GUI.
- Uses Hexadecimal values for everything except port numbers.
- Allows a particular port to be specified for viewing / writing with the –p # flag.
  - Specifying a number 0-31 returns the port specific registers for that port.
  - Specifying -1 returns all Global registers
  - Specifying -2 returns all PCI Config Space registers
  - **If memory writing is requested for a Port Specific register and no port is specified, an error is returned.**
- Allows a specific register to be specified for viewing / writing with the –n <register name> flag.
  - <register name> can be any register (Global, Port, or PCI)
  - If no port was specified (-p flag), all ports of the given port register are shown.
- Provides a help screen with the -? Command.
- Allows Global and Port specific (but not PCI Config) register values to be written.
  - –n <register name>=<hex> writes the given hex value into the given register, completely overwriting all bits of the register.
  - –n <register name>=<mask>:<value name>,<mask2>:<value name2>,… writes the numerical values behind the value names into the given masks, leaving the rest of the register untouched.
  - –n <register name>=<mask>:<hex>,<mask>:<hex>,… writes the given hex values into the given masks.
- Each flag can only be specified once per execution unless otherwise noted.

NVParse

- Parses a PCI Config Space Ref file (-pci <filename>) and Physical Memory (-f <filename>) ref file into a pre-parsed ref file for use by WinDbg and the GUI.
  - Must have at least one ref file specified (-f <filename> or –pci <filename>) with each execution of the program.
  - Can parse an arbitrary number of either type of ref file if the registers for a device are spread across multiple ref files.
- Allows the creation and execution of a batch file of commands that all run in quick succession.
  - This allows the GUI to send batch write instructions without having to reload the virtual driver repeatedly.

- Allows similar functionality as –n except with a memory offset instead of a register name by using the –o <offset> command.
    - This command also supports the same memory writing options as –n
- Allows the creation of a C style header file with the information pulled from the ref file with the –h <filename> command.
- Allows information about all registers to be dumped with the –all flag.
- Allows #<Port number> to be specified after the register name in a –n declaration. This overrides the port specified by –p.

WinDbg (NVata)

- Allows a specific pre-parsed ref file to be specified for use through the use of the –f <filename> flag
    - Otherwise defaults to “dump_file” in the WinDbg root directory.
- Allows commands to come through a socket connection and pipes the results of those commands back to the requester.
    - Only allows connections from localhost.
    - This facilitates communication with the GUI.
- Defaults to dumping information about all registers
    - -all flag can be specified, but it is redundant.
    - Specifying the –n or –p commands limits what is shown.

GUI Main Screen

- Displays two lists – one contains either Memory registers or PCI registers, while the other displays a specific port’s registers. These are selectable via tabs located above the lists.
    - Each list contains the register name, the port number (if applicable), and the current value of the register.
- When a register is selected from either list, more specific information about that register appears in the bottom section of the display
    - The register name and the current value are displayed. The value may be changed by clicking in the box and typing. This will automatically check to see if it a valid hex number, and update the masks accordingly.
    - Each mask is listed with its name and bit positions.

    - Each mask has an associated dropdown of its possible values.
        - This dropdown allows the user to select a known possible value (displayed as a string followed by the hex value).
        - The User can also type a hex value in the dropdown.
            - This hex value is checked for invalid hex characters and to ensure it is the proper bit width to fit the mask.
                - If it is invalid the value is reverted to the original value.
            - When the box is left the value of the register is updated accordingly..
        - All masks that have been changed are highlighted in green.
    - Changes may be written to a queue using the 'Add to Queue' button.
        - This changes the value displayed in the top section, as well as highlights it in green indicating it is awaiting writing.
        - This button is only enabled when changes have been made to the register.
    - Changes may be written to the register permanently using the 'Write Changes' button.
        - This will only save the currently selected register, and will also update all register information from the system.
        - This will only write the masks that have changed.
        - This button is only enabled when changes have been made to the register.
    - The number of columns used to display the mask information is dynamically determined based on the length of the names of the masks.
- On the right side of the top of the screen, there are four main buttons: 'Refresh All', 'Snapshot', 'Do Diff', and 'Write Queue'.
    - 'Refresh All' will query the system again using the program specified in the configuration, and repopulate all lists, updating the information.
    - 'Snapshot' will create a temporary buffer that stores the current configuration of every register.
        - Using the '…' next to Snapshot allows the user to specify a file to save this information to. This file will be later used by the "Diff …" button.
    - 'Do Diff' loads the information stored in the temporary buffer and compares it to the current values of the registers.
        - Any registers that are different will be highlighted in red.

        - When the user clicks on these values, a new read-only field will appear in the bottom section labeled “Old Value,” which is the value that was loaded from the temporary file.
        - Using the ‘…’ next to Do Diff allows the user to select a file created with “Snapshot …” to load, rather than simply using the temporary buffer.
    - ‘Write Queue’ will write all of the changes that were added to the queue.
        - It will also update all of the register information within the lists. (The same as if you had pressed “Refresh All”)
        - This button is not enabled until changes have been added to the queue.
- The menu strip has two main options, File and Edit.
    - File has Exit
        - Exit will exit the program, after prompting the user to make sure they truly wish to exit.
    - Edit has Do Diff, Do Diff…, Snapshot, Snapshot…, and Properties
        - **Do Diff uses the locally saved snapshot to perform a diff with**
        - **Do Diff… uses a file saved snapshot to perform a diff with**
        - **Snapshot creates a locally saved snapshot**
        - **Snapshot… creates a file saved snapshot**
        - Properties has Device and Source options
            - Device will bring up the configuration window with the device screen active
            - Source will bring up the configuration window with the source screen active
- The status bar at the bottom of the screen displays important information about program execution. This includes:
    - Any errors returned by WinDbg or NVParse.
    - A progress bar counting how long until we decide that WinDbg isn’t going to get back to us with the requested information
- **INCOMPLETE/MISSING/BROKEN FEATURES**
    - **If you leave a register without saving changes, all changes are lost.**

GUI (Configuration and Setup)

Source

- User can select either NVPCI or WinDBG as the source
- If WinDGB is chosen, the User can select the location for WinDBG, the default is "C:\Program Files\Debugging Tools for Windows". If the user wishes to select a folder, he must select a folder that has windbg.exe in it; otherwise, an error message is shown and the user must try again.

Device

- The system prompts the user with a list of Devices on the target system, including the Vendor ID, Device ID, Chip Name, and the associated ref files, and whether the device is AHCI or not.
    - The chip name and ref files will only be displayed if there is an entry in the systems library file.
    - The user is able to select one device from this list
- The user can specify a ref file(s) for the system to use. If the selected device has a library entry with ref files specified, the system will default to them.
- The user can specify a base offset for the system to use.
    - If the device being used is AHCI, and nothing or "---" is entered, the system will default to using the address in BAR5.
- After the user clicks the OK button, the GUI will populate itself, referencing the necessary configuration data.
- The system will auto generate the dump file that it needs to correctly display the register data. This dump file will also be copied to the WinDbg folder as 'dump_file' (the default pre-parsed ref file WinDbg uses) so WinDbg knows what registers to work with.
- **Missing Features/Bugs**
    - **If NVPCI does not exist, the system will display an error and then close the program.**

Appendix D: Future Features. These features included the ability to indicate which registers are read only, to work with a wider range of devices, and to sort registers alphabetically. For further information on all of these and other ideas, please see

This appendix describes the many features of NVParse, WinDbg, and the Graphical User Interface. Also, information that is especially important, or behavior that may not be expected, is highlighted in red.

NVParse and WinDbg (NVata)

- Displays register information in the following format:

```
Register Name:     <Register Name>
Port:              <port number>||<Type of Reg>
Register Offset:   <memory offset>
Register Value:    <value in memory>
Register Flags:
<Mask Name>:<Value String / or numerical value>
<Mask Name>:<Value String / or numerical value>
<Mask Name>:<Value String / or numerical value>
<Mask Name>:<Value String / or numerical value>
Red denotes NVParse only
```

- Provides a listing of all devices in hardware along with Bus #, Device #, Function #, Vendor ID, Device ID and a unique identifying ID to be used with other commands. The –dev flag generates this output.
  - NVParse provides human descriptions of the device (aka: "Memory Controller"
  - The presence of this flag overrides any other flags and prevents their execution.
- Allows a base memory address to be specified with the –b <hex offset> command.
  - This offset into memory overrides any memory offset automatically pulled from an AHCI device.
- When given no ID and asked to dump all registers, program defaults to the AHCI device found on the system.
- If no base memory offset is specified, uses BAR 5 as the memory offset if the current ID is an AHCI device.

- Provides standardized “code formatted” output with the –code flag for parsing by the GUI.
- Uses Hexadecimal values for everything except port numbers.
- Allows a particular port to be specified for viewing / writing with the –p # flag.
    - Specifying a number 0-31 returns the port specific registers for that port.
    - Specifying -1 returns all Global registers
    - Specifying -2 returns all PCI Config Space registers
    - **If memory writing is requested for a Port Specific register and no port is specified, an error is returned.**
- Allows a specific register to be specified for viewing / writing with the –n <register name> flag.
    - <register name> can be any register (Global, Port, or PCI)
    - If no port was specified (-p flag), all ports of the given port register are shown.
- Provides a help screen with the -? Command.
- Allows Global and Port specific (but not PCI Config) register values to be written.
    - –n <register name>=<hex> writes the given hex value into the given register, completely overwriting all bits of the register.
    - –n <register name>=<mask>:<value name>,<mask2>:<value name2>,… writes the numerical values behind the value names into the given masks, leaving the rest of the register untouched.
    - –n <register name>=<mask>:<hex>,<mask>:<hex>,… writes the given hex values into the given masks.
- Each flag can only be specified once per execution unless otherwise noted.

NVParse

- Parses a PCI Config Space Ref file (-pci <filename>) and Physical Memory (-f <filename>) ref file into a pre-parsed ref file for use by WinDbg and the GUI.
    - Must have at least one ref file specified (-f <filename> or –pci <filename>) with each execution of the program.
    - Can parse an arbitrary number of either type of ref file if the registers for a device are spread across multiple ref files.
- Allows the creation and execution of a batch file of commands that all run in quick succession.
    - This allows the GUI to send batch write instructions without having to reload the virtual driver repeatedly.

- Allows similar functionality as –n except with a memory offset instead of a register name by using the –o <offset> command.
    - This command also supports the same memory writing options as –n
- Allows the creation of a C style header file with the information pulled from the ref file with the –h <filename> command.
- Allows information about all registers to be dumped with the –all flag.
- Allows #<Port number> to be specified after the register name in a –n declaration. This overrides the port specified by –p.

WinDbg (NVata)

- Allows a specific pre-parsed ref file to be specified for use through the use of the –f <filename> flag
    - Otherwise defaults to "dump_file" in the WinDbg root directory.
- Allows commands to come through a socket connection and pipes the results of those commands back to the requester.
    - Only allows connections from localhost.
    - This facilitates communication with the GUI.
- Defaults to dumping information about all registers
    - -all flag can be specified, but it is redundant.
    - Specifying the –n or –p commands limits what is shown.

GUI Main Screen

- Displays two lists – one contains either Memory registers or PCI registers, while the other displays a specific port's registers. These are selectable via tabs located above the lists.
    - Each list contains the register name, the port number (if applicable), and the current value of the register.
- When a register is selected from either list, more specific information about that register appears in the bottom section of the display
    - The register name and the current value are displayed. The value may be changed by clicking in the box and typing. This will automatically check to see if it a valid hex number, and update the masks accordingly.
    - Each mask is listed with its name and bit positions.

  - Each mask has an associated dropdown of its possible values.
    - This dropdown allows the user to select a known possible value (displayed as a string followed by the hex value).
    - The User can also type a hex value in the dropdown.
      - This hex value is checked for invalid hex characters and to ensure it is the proper bit width to fit the mask.
        - If it is invalid the value is reverted to the original value.
      - When the box is left the value of the register is updated accordingly..
    - All masks that have been changed are highlighted in green.
  - Changes may be written to a queue using the 'Add to Queue' button.
    - This changes the value displayed in the top section, as well as highlights it in green indicating it is awaiting writing.
    - This button is only enabled when changes have been made to the register.
  - Changes may be written to the register permanently using the 'Write Changes' button.
    - This will only save the currently selected register, and will also update all register information from the system.
    - This will only write the masks that have changed.
    - This button is only enabled when changes have been made to the register.
  - The number of columns used to display the mask information is dynamically determined based on the length of the names of the masks.
- On the right side of the top of the screen, there are four main buttons: 'Refresh All', 'Snapshot', 'Do Diff', and 'Write Queue'.
  - 'Refresh All' will query the system again using the program specified in the configuration, and repopulate all lists, updating the information.
  - 'Snapshot' will create a temporary buffer that stores the current configuration of every register.
    - Using the '…' next to Snapshot allows the user to specify a file to save this information to. This file will be later used by the "Diff …" button.
  - 'Do Diff' loads the information stored in the temporary buffer and compares it to the current values of the registers.
    - Any registers that are different will be highlighted in red.

        - When the user clicks on these values, a new read-only field will appear in the bottom section labeled "Old Value," which is the value that was loaded from the temporary file.
        - Using the '…' next to Do Diff allows the user to select a file created with "Snapshot …" to load, rather than simply using the temporary buffer.
    - 'Write Queue' will write all of the changes that were added to the queue.
        - It will also update all of the register information within the lists. (The same as if you had pressed "Refresh All")
        - This button is not enabled until changes have been added to the queue.
- The menu strip has two main options, File and Edit.
    - File has Exit
        - Exit will exit the program, after prompting the user to make sure they truly wish to exit.
    - Edit has Do Diff, Do Diff…, Snapshot, Snapshot…, and Properties
        - **Do Diff uses the locally saved snapshot to perform a diff with**
        - **Do Diff… uses a file saved snapshot to perform a diff with**
        - **Snapshot creates a locally saved snapshot**
        - **Snapshot… creates a file saved snapshot**
        - Properties has Device and Source options
            - Device will bring up the configuration window with the device screen active
            - Source will bring up the configuration window with the source screen active
- The status bar at the bottom of the screen displays important information about program execution. This includes:
    - Any errors returned by WinDbg or NVParse.
    - A progress bar counting how long until we decide that WinDbg isn't going to get back to us with the requested information
- **INCOMPLETE/MISSING/BROKEN FEATURES**
    - **If you leave a register without saving changes, all changes are lost.**

GUI (Configuration and Setup)

Source

- User can select either NVPCI or WinDBG as the source
- If WinDGB is chosen, the User can select the location for WinDBG, the default is "C:\Program Files\Debugging Tools for Windows". If the user wishes to select a folder, he must select a folder that has windbg.exe in it; otherwise, an error message is shown and the user must try again.

Device

- The system prompts the user with a list of Devices on the target system, including the Vendor ID, Device ID, Chip Name, and the associated ref files, and whether the device is AHCI or not.
    - The chip name and ref files will only be displayed if there is an entry in the systems library file.
    - The user is able to select one device from this list
- The user can specify a ref file(s) for the system to use. If the selected device has a library entry with ref files specified, the system will default to them.
- The user can specify a base offset for the system to use.
    - If the device being used is AHCI, and nothing or "---" is entered, the system will default to using the address in BAR5.
- After the user clicks the OK button, the GUI will populate itself, referencing the necessary configuration data.
- The system will auto generate the dump file that it needs to correctly display the register data. This dump file will also be copied to the WinDbg folder as 'dump_file' (the default pre-parsed ref file WinDbg uses) so WinDbg knows what registers to work with.
- **Missing Features/Bugs**
    - **If NVPCI does not exist, the system will display an error and then close the program.**

Appendix D: Future Features.

9 Conclusion

The process of completing this project has been a great educational experience. As discussed in the Results section, we were able to deliver a working product that will meet NVIDIA's needs for some time to come. Beyond this, however, the learning experience we have gained greatly outweighs any benefit this product provides to NVIDIA. Introduction to a large corporation work environment, learning new languages and technologies, and revisiting project flow were all invaluable experiences.

9.1 New Technologies

Throughout this project we were introduced to new technologies which we had to learn in a very short time frame. None of us had used C# before or the version of Visual Studio we worked with. In many other cases, one of us had experience in the technology but the other two were unfamiliar with it. Unfamiliarity with the tools and languages we used made us better at finding resources to improve our understanding of these tools. We also had to discriminate between tutorials and resources that followed good design and modern implementations and those that did not. This skill is something that is invaluable and can only be gained through real life experience.

9.2 Project Flow

In an academic setting with multiple simultaneous classes and a limited timeframe is difficult at best to gain real experience with the flow of a project from initial requirements to a finished product. Through this project we were able to focus only on the project and bring it from idea to completion in a scheduled amount of time. The steps taken to get from start to end and the changes to the project made as the process progressed were invaluable to see and experience. The process of working through these changes with our advisor was greatly enlightening, as it provided the opportunity to balance the features that were going to be added to the project, while also keeping in mind the time frame that we had to complete it. This experience will help us in the future to schedule and plan our own projects realistically.

9.3 Techniques

The most important outcome of this project for us was the experience we gained by working on this project with a large corporation. Observing how current employees work with each other, how code ownership works, and how the process for incorporating changes into version control works at a real company was invaluable. Another reassuring part of this process was observing the mentality that in a company everyone is working towards a common goal and, as such, it's encouraged to seek out help when you do not understand something. Having an understanding of these things at a company provides a new perspective on the techniques outlined in the Software Engineering course and insight on how things may work differently from company to company. This understanding will greatly ease the transition from an academic setting to a workplace environment.

9.4 Final Word

The learning experience offered by completing a project of the scope that is undertaken during an MQP, while in a new environment and under extreme time constraints, is invaluable. The lessons acquired during this project will be with us for years to come, with time sure to reveal more lessons as we move forward to our next project.

10 Works Cited

1. **NVIDIA.** Company Info. *NVIDIA.* [Online] November 6, 2007. [Cited: November 6, 2007.] http://www.nvidia.com/page/companyinfo.html.

2. **wiseGeek.** What is SATA or Serial ATA? [Online] [Cited: November 29, 2007.] http://www.wisegeek.com/what-is-sata-or-serial-ata.htm.

3. **The Peripheral Component Interconnect Special Interest Group (PCI-SIG).** Conventional PCI 3.0 & 2.3. *PCI-SIG.* [Online] Febuary 10, 2008. [Cited: Febuary 10, 2008.] http://www.pcisig.com/specifications/conventional/.

4. **The Serial ATA International Organization.** Serial ATA FAQs. *Serial ATA.* [Online] The Serial ATA International Organization, November 29, 2007. [Cited: November 29, 2007.] http://www.serialata.org/faqs.asp.

5. **Intel Corporation.** AHCI Specification Rev 1.2. *Intel .* [Online] 2007. http://download.intel.com/technology/serialata/pdf/rev1_2.pdf. ISBN/ISSN.

6. **Microsoft.** Debugging Tools for Windows. *Microsoft.* [Online] 2007. [Cited: 12 29, 2007.] http://www.microsoft.com/whdc/devtools/debugging/default.mspx.

7. **GNU.** Sockets. *The GNU C Library.* [Online] 3 1, 2008. [Cited: 3 1, 2008.] http://www.gnu.org/software/libtool/manual/libc/Sockets.html.

8. **Intel.** Homepage. *AHCI Specification for Serial ATA.* [Online] Intel, December 2, 2007. [Cited: December 2, 2007.] http://www.intel.com/technology/serialata/ahci.htm.

Appendix A: User Documentation

This appendix will describe the many different features of the Graphical User Interface. Also, any information which is not readily apparent, such as unexpected behavior, is noted.

Feature List

- Configuration Window
    - Select either WinDbg or NVParse to perform reads and writes throughout the program
    - User configurable location of WinDbg
    - User configurable time-out time for connection to WinDbg
    - When the user selects a known device from the list, the program automatically selects and uses the appropriate ref files
    - The user may also select their own ref files for use within the program
    - User configurable memory offset with automatic calculation of memory offset for AHCI devices
    - The program saves almost all settings from the Configuration window between sessions
- Main Program
    - Displays AHCI, PCI, and Port names and values in easy to read lists
    - Displays individual mask names, values, and bit offsets for the selected Register
    - Updates mask values automatically given a new Register value
    - User may select a value for an individual mask from a drop down menu, or may input their own values and the Register value changes accordingly
    - All user inputs are error checked automatically
    - The user may create local screenshots, which last for the duration of the session, or write the screenshot to a file
    - The user may use the local screenshot, or a screenshot from a file, and create a diff
    - The user may write an entire register, or only the changed masks, to the computer
    - The user may add an entire register, or only the changed masks, to a queue for later writing
    - All register and port information may be updated at any time

Installation

To install NVestigate simply double click the installer icon. The installer scans for preinstalled components and prompts to install any components that it needs and cannot find. Installation also prompts for the location of WinDbg which is needed for proper function of the program. The installer will also check for Perl and WinDbg on the system. It will prompt the user to install them, but won't require it if the user wishes to continue without installing them

How to Use

While a large portion of the program is intuitive, this section will describe a few of the features that may not be as common. How the GUI interacts with both NVParse and NVATA will also be explained in further detail.

Configuration Window

The configuration window is the first screen that is presented to the user when the program is run. It can also be accessed through the Edit menu. While most of the screen's functionality is self-evident, some aspects of the interaction with other parts of the program may not be obvious. The WinDbg location field is to the location specified in the config file; however, the user can change this if necessary. This is later used for many different features, such as the location to copy files to. The time-out is the number of seconds to wait for a connection to WinDbg. This is important because this communication while this connection is made using a socket connection, and while it may be very quick if you are debugging a local, over a serial connection it may take much longer.

The next screen of the Configuration window is the device screen. This allows the user to select a device from an automatically generated list. The list of devices is populated by using the source chosen in the first screen – only those devices that are present on the system are displayed. If NVParse is selected as the source, the list is completely repopulated each time that it is displayed on the screen. However, because of the time needed for WinDbg to do a complete device scan, NVATA caches the device list after the first time it generates it. This cached list is returned to the GUI during subsequent configuration screen loads. If for some reason you need a fresh copy of the device list, restart WinDbg or run "!dumpnv –dev force" from the WinDbg command line will ensure that the next configuration screen load will get a new copy of the list.

The information for the name of the device, as well as what ref files are associated with it, is located within a library file located on a network share. The library file may be updated with additional devices, and as long as it adheres to the format specified in Appendix J, it will be recognized by the program automatically. Also, because the library file is within a network share, if one person changes the file than every user will have the updated file. The user may also choose ref files for memory and PCI configuration space. Finally, the memory offset must also be specified. The user must supply this information on his own by either looking in the PCI Config space with either WinDbg or NVParse, or obtain it elsewhere.

Main Form

The main window is where most time is spent during program use. It displays the registers found using the selected ref files for the selected device and allows for editing of these values.

One of the most important features of the main form is the ability to write to individual registers. This can be done several ways, although the result is similar in all cases. Unless the 'Write Entire Register' checkbox is checked, than only masks that have changed are written to memory. The 'Write Entire Register' checkbox specifies all masks are written to memory regardless of if they have changed or not. This holds true for both writing immediately, and registers added to the write queue. The queue writes all of the registers added to it to memory at once. An important aspect of this, however, is to keep in mind that the queue is destroyed if the program is closed. How exactly they are written depends on which source is chosen. If NVParse is being used, one call is made to NVParse, were each register is specified in its own '-n' parameter. Otherwise, if WinDbg is the source, a call is made to WinDbg for each register, using the standard register writing format (!dumpnv –id # -b #### -n <name>=<mask>:<value> -p #). In either case the entire list of registers is refreshed after the write, ensuring that only current values are shown. Another important note is that if you attempt to write to a register which is read only, the program will let you attempt the write, but will show the old value when it is updated again. Also, WinDbg will not allow the alteration of any PCI register, whereas NVParse will allow it as long as the register is writable.

Another feature of the main form is the ability to create snapshots. This will save the current state either locally or into a file specified by the user. If stored locally, than it will only exist until the program is closed. An important aspect to note is that if a screenshot is saved into a file, it also saves it locally. Only one snapshot may be saved locally at a time. Using the diff function, the user may either use the locally saved snapshot (this option is not available until one exists), or choose a saved screen shot. If an item occurs both in the write queue, and was changed from when the screen shot was taken, only the diff view will be available, although it still exists within the queue.

Missing Features

- Does not show which registers are read-only
- Cannot sort the list of Registers
- Does not remember which device was last selected in the Configuration window

Modifying Config.ini

The configuration file in NVestigate's directory contains information the system requires, including the default backend, the paths for NVParse and WinDBG, the library file path, as well as the timeout for the connection to WinDBG. The default backend, WinDBG path and the timeout for WinDBG are handled by the GUI, modifying these sections is not recommended. If the user wishes to change the path for NVParse, the user should make sure that NVParse exists in that directory; the same holds true for modifying the library file path. These entries can accept both relative and absolute paths. The ID and Base entries are deprecated and modifying them will have no effect. Any other entries in the ini file will be ignored.

Library.lb

The library file contains information about NVIDIA devices and allows for the automatic insertion of ref files based on the device (for format specification, see: Formats.doc). If the user modifies this file, he should make sure that his modifications strictly adheres to the format specified, as NVestigate's behavior is unpredictable with an improperly formatted library file.

Appendix B: Command Line Syntax Reference

This appendix contains a quick reference for all available command line parameters for NVATA and NVParse along with several examples of how they are used.

USAGE:

```
<NVParse.pl | !dumpnv> -f FILENAMES [-id DEVICEID] [-b BASEADDRESS]
            [-n NAME[[=VALUE] |
                   [=[MASK:<FLAG|VALUE>][,MASK:<FLAG|VALUE>] ...  ]
                  ]
            ]
            [-all] [-code] [-p PORT] [-dev] [-?]
```

-f FILENAMES

NVParse.pl: Filename(s) of .reg files, must be specified to run any commands other than help, these reg files should contain registers in memory space

!dumpnv: Filename of dump file to use. Defaults to dump_file if none specified.

-id ID

Unique id of the device to be read from, '-dev' can be used to get a list of unique ids

-b BASEADDRESS

The base address of the registers you will be writing/reading from, either this or -id is necessary to read and/or write from memory

-n NAME[#PORT][[=VALUE] | [=[MASK:<FLAG|VALUE>][,MASK:<FLAG|VALUE>] ...]]]

Register name to be examined, if a port is given, only that port register is examined, specifying a port for a non-port registers will return no registers, if a value is given, the value will be written to the register, if a mask is given, followed by the appropriate flag or value, the information is written to the range of the mask, if a value is specified, it should be set to the least significant bits. NOTE: NVATA does not support the [#PORT] specification shown here. This specification is to allow multiple -n calls in one execution of NVParse, this is not supported in NVATA.

-p PORT

The port we are interested in for port specific registers. Using this parameter without a -n parameter will show the values for all registers of the specified port, -1 will show all Global registers, -2 will show PCI Config Space registers.

-all

Prints information about all the registers. This is the default behavior in NVATA.

-code

Prints data in a format that is easy to parse rather than easy to read (This is used by the gui and should not be necessary for the average user

-dev [force]s

Brings up a list of all the PCI devices on the bus, specifying which ones are AHCI compliant. If -code is specified, the PCI devices are listed in a parsable format. The optional parameter 'force' causes NVATA to rescan all devices instead of returning a cached list.

-?

Brings up this help file

Additional Parameters for functionality only found in NVParse.pl

```
Nvparse.pl -pci FILENAMES
        [[-o OFFSET[[=VALUE]
              | [=[MASK:<FLAG|VALUE>][,MASK:<FLAG|VALUE>] ...  ]]] |
         [-d DUMPFILE] [-h HEADERFILE]
```

-pci FILENAMES

Filename(s) of .reg files, must be specified to run any commands other than help, these reg files should contain registers in PCI Config space. Both this parameter and -f should be specified if a device has PCI and Global / Port registers.

-d DUMPFILE

Filename of the file created that will contain a simplified format of the registers

-h HEADERFILE

Filename of the header file that will be created that will contain C style macros

-o OFFSET[=VALUE]

Register offset to be examined, if a value is given, that value will be written to the register, if a mask is given, followed by the appropriate flag or value, the information is written to the range of the mask, if a value is specified, it should be set to the least significant bits. This functionality is the same as -n execept an offset is specified instead of a register name.

Examples:

<program> is used to indicate either NVParse.pl or !dumpnv can be used.

Help functions work as follows:

```
<program> -?
```

Calling the program without any arguments will also bring up the help command in NVParse.

To see all the devices on the system, use the following command:

```
<program> -dev
```

This will list all of the devices on the system, if the '-code' argument is added, the output will be in a parsable format

To do a dump that will be parsed by other programs, use the following commands:

```
nvparse.pl -f memory_space_ref_file -d dump_file
nvparse.pl -pci pci_config_ref_file -d dump_file
nvparse.pl -f memory_space_ref_file -pci pci_config_ref_file -d dump_file
```

This will dump all the reference data into an easy to parse format NOTE: an id or base address is not necessary

To dump the data to a c-style header:

```
            nvparse.pl -f ref_file -h header.h
```

Like -d, this will create a file with c-style macros, no other arguments are necessary

To print out all the register information:

```
            <program> -f file -all -b # or (if AHCI/RAID)
            <program> -f file -all -id #
```

In NVParse file should be a ref file, in NVATA it should be a dump file. This will dump all the register information to the screen, if the -p argument is used as follows, the program will print all the registers associated with that port

```
            <program> -f file -all -id # -p #
```

Adding the -code argument will print the same info, but in a format that is easy for another program to understand, much like the dump file

```
            <program> -f file -all -id # -code
```

To read from a specific location in memory:

```
            <program> -f file -id # -n AHCI_HBA_CCC_PORTS
            nvparse.pl -f file -id # -o 18
```

The system will take a name for any recognized register. Nvparse can also use an offset. If an offset is used, a hexadecimal number must be used

To write to a specific location:

```
            <program> -f file -b # -n AHCI_HBA_EM_CTL=F
            <program> -f file -b # -n AHCI_HBA_EM_CTL=_STS_MR:1
            <program> -f file -b # -n AHCI_HBA_EM_CTL=_STS_MR:_TRUE,_SUPP_SGPIO:_FALSE
            <program> -f file -b # -n AHCI_HBA_CCC_PORTS=15

            nvparse.pl -f ref_file -b # -o 0x18=FFFF
            nvparse.pl -f ref_file -b # -o 24=_PRT:_RESET_VAL
```

As you can see there are many options to write to memory. The user can choose to write all of a register in one call, but they can also set specific flags. These flags must include the value of the mask for that part of the register. After the flag, the user can either specify a value (in either hex or decimal) or a flag. This value will be shifted over, masked, and written to the register. These same calls will work for writing by offset as described above.

Appendix C: Features

This appendix describes the many features of NVParse, WinDbg, and the Graphical User Interface. Also, information that is especially important, or behavior that may not be expected, is highlighted in red.

NVParse and WinDbg (NVata)

- Displays register information in the following format:

```
Register Name:    <Register Name>
Port:             <port number>||<Type of Reg>
Register Offset:  <memory offset>
Register Value:   <value in memory>
Register Flags:
<Mask Name>:<Value String / or numerical value>
<Mask Name>:<Value String / or numerical value>
<Mask Name>:<Value String / or numerical value>
<Mask Name>:<Value String / or numerical value>
Red denotes NVParse only
```

- Provides a listing of all devices in hardware along with Bus #, Device #, Function #, Vendor ID, Device ID and a unique identifying ID to be used with other commands. The –dev flag generates this output.
    - NVParse provides human descriptions of the device (aka: “Memory Controller”
    - The presence of this flag overrides any other flags and prevents their execution.
- Allows a base memory address to be specified with the –b <hex offset> command.
    - This offset into memory overrides any memory offset automatically pulled from an AHCI device.
- When given no ID and asked to dump all registers, program defaults to the AHCI device found on the system.
- If no base memory offset is specified, uses BAR 5 as the memory offset if the current ID is an AHCI device.
- Provides standardized “code formatted” output with the –code flag for parsing by the GUI.
- Uses Hexadecimal values for everything except port numbers.
- Allows a particular port to be specified for viewing / writing with the –p # flag.
    - Specifying a number 0-31 returns the port specific registers for that port.

    - Specifying -1 returns all Global registers
    - Specifying -2 returns all PCI Config Space registers
    - **If memory writing is requested for a Port Specific register and no port is specified, an error is returned.**
- Allows a specific register to be specified for viewing / writing with the –n <register name> flag.
    - <register name> can be any register (Global, Port, or PCI)
    - If no port was specified (-p flag), all ports of the given port register are shown.
- Provides a help screen with the -? Command.
- Allows Global and Port specific (but not PCI Config) register values to be written.
    - –n <register name>=<hex> writes the given hex value into the given register, completely overwriting all bits of the register.
    - –n <register name>=<mask>:<value name>,<mask2>:<value name2>,… writes the numerical values behind the value names into the given masks, leaving the rest of the register untouched.
    - –n <register name>=<mask>:<hex>,<mask>:<hex>,… writes the given hex values into the given masks.
- Each flag can only be specified once per execution unless otherwise noted.

NVParse

- Parses a PCI Config Space Ref file (-pci <filename>) and Physical Memory (-f <filename>) ref file into a pre-parsed ref file for use by WinDbg and the GUI.
    - Must have at least one ref file specified (-f <filename> or –pci <filename>) with each execution of the program.
    - Can parse an arbitrary number of either type of ref file if the registers for a device are spread across multiple ref files.
- Allows the creation and execution of a batch file of commands that all run in quick succession.
    - This allows the GUI to send batch write instructions without having to reload the virtual driver repeatedly.
- Allows similar functionality as –n except with a memory offset instead of a register name by using the –o <offset> command.
    - This command also supports the same memory writing options as –n

- Allows the creation of a C style header file with the information pulled from the ref file with the –h <filename> command.
- Allows information about all registers to be dumped with the –all flag.
- Allows #<Port number> to be specified after the register name in a –n declaration. This overrides the port specified by –p.

WinDbg (NVata)

- Allows a specific pre-parsed ref file to be specified for use through the use of the –f <filename> flag
  - Otherwise defaults to “dump_file” in the WinDbg root directory.
- Allows commands to come through a socket connection and pipes the results of those commands back to the requester.
  - Only allows connections from localhost.
  - This facilitates communication with the GUI.
- Defaults to dumping information about all registers
  - -all flag can be specified, but it is redundant.
  - Specifying the –n or –p commands limits what is shown.

GUI Main Screen

- Displays two lists – one contains either Memory registers or PCI registers, while the other displays a specific port’s registers. These are selectable via tabs located above the lists.
  - Each list contains the register name, the port number (if applicable), and the current value of the register.
- When a register is selected from either list, more specific information about that register appears in the bottom section of the display
  - The register name and the current value are displayed. The value may be changed by clicking in the box and typing. This will automatically check to see if it a valid hex number, and update the masks accordingly.
  - Each mask is listed with its name and bit positions.
  - Each mask has an associated dropdown of its possible values.
    - This dropdown allows the user to select a known possible value (displayed as a string followed by the hex value).
    - The User can also type a hex value in the dropdown.

                - This hex value is checked for invalid hex characters and to ensure it is the proper bit width to fit the mask.
                    - If it is invalid the value is reverted to the original value.
                - When the box is left the value of the register is updated accordingly..
            - All masks that have been changed are highlighted in green.
    - Changes may be written to a queue using the ‘Add to Queue’ button.
        - This changes the value displayed in the top section, as well as highlights it in green indicating it is awaiting writing.
        - This button is only enabled when changes have been made to the register.
    - Changes may be written to the register permanently using the ‘Write Changes’ button.
        - This will only save the currently selected register, and will also update all register information from the system.
        - This will only write the masks that have changed.
        - This button is only enabled when changes have been made to the register.
    - The number of columns used to display the mask information is dynamically determined based on the length of the names of the masks.
- On the right side of the top of the screen, there are four main buttons: ‘Refresh All’, ‘Snapshot’, ‘Do Diff’, and ‘Write Queue’.
    - ‘Refresh All’ will query the system again using the program specified in the configuration, and repopulate all lists, updating the information.
    - ‘Snapshot’ will create a temporary buffer that stores the current configuration of every register.
        - Using the ‘…’ next to Snapshot allows the user to specify a file to save this information to. This file will be later used by the “Diff …” button.
    - ‘Do Diff’ loads the information stored in the temporary buffer and compares it to the current values of the registers.
        - Any registers that are different will be highlighted in red.
        - When the user clicks on these values, a new read-only field will appear in the bottom section labeled “Old Value,” which is the value that was loaded from the temporary file.
        - Using the ‘…’ next to Do Diff allows the user to select a file created with “Snapshot …” to load, rather than simply using the temporary buffer.

        - 'Write Queue' will write all of the changes that were added to the queue.
            - It will also update all of the register information within the lists. (The same as if you had pressed "Refresh All")
            - This button is not enabled until changes have been added to the queue.
- The menu strip has two main options, File and Edit.
    - File has Exit
        - Exit will exit the program, after prompting the user to make sure they truly wish to exit.
    - Edit has Do Diff, Do Diff…, Snapshot, Snapshot…, and Properties
        - **Do Diff uses the locally saved snapshot to perform a diff with**
        - **Do Diff… uses a file saved snapshot to perform a diff with**
        - **Snapshot creates a locally saved snapshot**
        - **Snapshot… creates a file saved snapshot**
        - Properties has Device and Source options
            - Device will bring up the configuration window with the device screen active
            - Source will bring up the configuration window with the source screen active
- The status bar at the bottom of the screen displays important information about program execution. This includes:
    - Any errors returned by WinDbg or NVParse.
    - A progress bar counting how long until we decide that WinDbg isn't going to get back to us with the requested information
- **INCOMPLETE/MISSING/BROKEN FEATURES**
    - **If you leave a register without saving changes, all changes are lost.**

GUI (Configuration and Setup)

Source

- User can select either NVPCI or WinDBG as the source
- If WinDGB is chosen, the User can select the location for WinDBG, the default is "C:\Program Files\Debugging Tools for Windows". If the user wishes to select a folder, he must select a

folder that has windbg.exe in it; otherwise, an error message is shown and the user must try again.

Device

- The system prompts the user with a list of Devices on the target system, including the Vendor ID, Device ID, Chip Name, and the associated ref files, and whether the device is AHCI or not.
    - The chip name and ref files will only be displayed if there is an entry in the systems library file.
    - The user is able to select one device from this list
- The user can specify a ref file(s) for the system to use. If the selected device has a library entry with ref files specified, the system will default to them.
- The user can specify a base offset for the system to use.
    - If the device being used is AHCI, and nothing or “---“ is entered, the system will default to using the address in BAR5.
- After the user clicks the OK button, the GUI will populate itself, referencing the necessary configuration data.
- The system will auto generate the dump file that it needs to correctly display the register data. This dump file will also be copied to the WinDbg folder as ‘dump_file’ (the default pre-parsed ref file WinDbg uses) so WinDbg knows what registers to work with.
- **Missing Features/Bugs**
    - **If NVPCI does not exist, the system will display an error and then close the program.**

Appendix D: Future Features

Throughout this project we have strived to keep this code as extendable as possible. We have accomplished this through designing clean, well documented, object oriented code. We have ensured that each function only performs on major task and is reusable when other aspects of the program may need the same behavior. If the program modifications must be made, the HTML help file, the in code documentation, and this document are useful resources.

Non-AHCI Device Support

One of the early requests for extensibility was to make this program work for more devices than just AHCI devices. This additional feature is largely implemented already; however, additional work is required to ensure that it works flawlessly.

Currently, NVATA (WinDbg) and NVParse will return values for all the port registers specified in the given ref file for as many ports as it finds in AHCI_HBA_PI (offset 0xC) regardless of if the device being looked at is AHCI or not. This technique, however, does not work correctly if the device is non-AHCI. In this case, the offset 0xC may not necessarily be the correct location to find the number of ports. To change this, the following general steps would need to be implemented:

- The location of the proper register containing this information should be encoded into the Library entry for each device.
- NVParse should be extended to build this information into the dump file.
- WinDbg and NVParse should be extended to use this information when determining what register to check for ports.
- The GUI should be modified to not display a list for port information if no ports exist in the dump file.

Additional effort may be required, however, if the current method of specifying which ports are currently in use is changed.

Finally, global and port specific registers are currently differentiated by their memory offset. Both NVParse and NVATA declare a register to be a port specific register if its offset is above 0x100 which may not hold true for other non-AHCI devices.

Network-Redundant Library and Ref Files

Over time, the original Library file, as well as the ref files, which were installed automatically, could become out of date as devices are added or modified. An easy technique to maintain these files across many users is to store the files centrally on a file share. The program currently supports reading the library file and ref files from the network. The space for improvement upon this comes in the case where the network is not available at run time. Additional code could be added to the GUI to allow it to take two parameters in config.ini, one for a network location for the files, and one for a local copy of the files. Also, a simple mechanism could be implemented that ensures that the most up-to-date files are copied locally when a network connection is available.

NVParse Efficiency Improvements

Currently every command to NVParse requires specifying at least one ref file. This ref file is parsed into a local data structure and then transformed to a dump file and/or used to perform memory reads and writes. The act of parsing the ref file is fairly processor intensive, and the act of fetching the ref file from a network drive may also be time consuming. A possible improvement would involve modifying NVParse to enable parsing a dump file. This file would be significantly easier to parse, and could be cached locally between calls. This modification would need to implement a function similar to an existing function, parse(), and provide a mechanism for choosing between ref file parsing and dump file parsing.

Indicate Read Only Registers

A feature requested later in the development of the program was the ability to show which registers are read only. This information is currently stored in the ref files, but is ignored by the parsing function. A modification to NVParse would have to be made to extract this information from the comment after each register and store it within the dump file. This information would have to then be pulled from the dump file by the GUI and the display updated accordingly.

Allow Sorting of Registers

The list of registers shown in the main view is currently shown in the same order in which the registers were found in the ref file. This is not a terrible solution, as this order is generally logical and is often the order that the registers exist in memory. However, it would be a nice feature to allow the user to sort in accordance to their data. For example, to sort the lists alphabetically to assist in finding a specific register more quickly.

The list of registers shown is actually an ArrayList of Registers displayed in a C# DataGridView object. ArrayLists do not have the ability to be sorted out of the box, but modifying this list to be a binding list and implementing the SortColumns property and Sort() family of functions would allow the list to be sorted. There are several other possible ways of sorting a DataGridView, and it may be found that another approach is easier; we would recommend some research be done prior to attempting this change.

BUGS

Throughout the project, diligence was shown in testing for bugs; however, as with any project, a few are known to exist. Included below is a list of the known bugs within the program, as well as suggestions for correcting them in the future.

Refresh With Items in Queue

This bug may be recreated by following these steps:

5. Select a register
6. Change the register value and add it to the queue
7. Refresh the list (either through the 'Refresh All' button or writing another register value)
8. The register that was changed will no long be highlighted in green, and will not display the NEW VALUE field when clicked on, but it will still be written when 'Write Queue' is clicked on

The reason this bug exists is because the field that is responsible to control the display of NEW VALUE and highlight the register value green is oldValue within the Register class. This is over-written, however, when the lists are refreshed. This sets oldValue back to null, and eliminates the display of the new value; however, because the queue is also kept in a separate ArrayList, it is not destroyed there and may still be written.

This may be fixed by adding an additional field to the Register class, perhaps entitled newValue, to store the value that is currently being held within the queue. Than, within the cellFormatting event in topSectionEvents, change the calls to match this new functionality. Also, within displayBottomLeft() in BottomSection, change the display calls to match this change. Finally, when a refresh is done, the changedRegisters ArrayList within BottomSection should be looped through, and for each Register within it set the corresponding newValue in Register within allInformation (found within topSection).

Queue and Diff Conflict

This bug occurs when a register exists in the queue, waiting to be written, and the user performs a diff in which the same register exists.

To recreate this bug, follow these steps:

5. Create a snapshot
6. Write a change to a register
7. For the same register, add a change to the queue
8. Load the diff

This shows that only the diff is displayed, and not the queue. If show diff is turned off, the green will be shown again, however the value within NEW VALUE will be the value loaded for the diff. Another aspect of this bug is that if a new value is added to the queue for this same register, and the diff is being shown, than the OLD VALUE field will change to reflect this change. This bug may be fixed following the same steps listed above.

Appendix E: Formats

This appendix provides examples of the different formats that are used throughout the NVestigate software suite. These examples are also described in detail to provide a thorough understanding.

Dump File

No comments are allowed in the actual file.

```
AHCI_PORT_MP                //Register Name
17C                         //Memory Offset
0                           //0=Port Register, -1=Global, -2= PCI
[tab]   _MPA                   //Mask name
80                             //Bit mask
[tab]   [tab]    _SET                //Flag name
1                                    //Value of Flag
[tab]   [tab]    _DEFAULT            //Next Flag
0
[tab]   [tab]    _CLEAR
0
[tab]   _MPRSVD_1              //Next mask
40
[tab]   [tab]    _0
0
[tab]   _MPIS
20
[tab]   [tab]    _ZERO
0
[tab]   [tab]    _SET
1
[tab]   [tab]    _CLEAR
1
AHCI_HBA_CAP                   //Next Register
//...
```

Library File

No comments are allowed in the actual file. Multiple PCI and Global/Port ref files can be specified for a given device by listing each complete path on the appropriate line separated by white space.

```
__BEGIN_ENTRY__                           //Entry header
10DE                                      //Vendor ID
044C                                      //Device ID
MCP65                                     //Device Name
mcp65\manuals\dev_nv_proj__fpci_sata.ref  //PCI Ref files
mcp65\manuals\dev_nv_proj__sata.ref       //Global / Port ref files
```

Register Information –code

This output is returned when a command requesting register information is given including the –code flag. Comments are not part of output.

```
__BEGIN_REGISTER__   //Header at start of each register
AHCI_HBA_CAP         //Register Name
-1                   //-1=Global, -2= PCI, 0<= Port number
00000000             //Memory Offset
E7229F03             //Value in memory
_NP:_4               //Mask:Flag
_ABCD:4D             //Mask:Hex Value (if no flag found)
_SXS:_FALSE          //And so on for each mask
```

Device Listing (-dev -code)

This is the output format for a –dev –code command.

```
__PCI_CONFIG_DATA__     //Header once at top of output
78:1022:1103            //One device per line
58:1022:1102            //ID:Vendor ID:Device ID
*9:10DE:0554            //Preceded by a * if the device is AHCI
38:1022:1101            //And so on...
```

Appendix F: GUI Button Tests

This appendix describes each of the buttons that may be found in the main screen of the GUI. The description includes information about how the button works, what it does, and also how to test it. Any information that could be important, such as unexpected behavior, is also noted.

Refresh All

- How it works
  - Repopulates every list with the current register information
- How to test

  7. Change a writable register value using NVParse
  8. Click on 'Refresh All'
  9. Verify that the value changed within the GUI

Snapshot

- How it works
  - Creates a local save of the current register information within a string
  - Formatted the same as the information that is read into the GUI (Register info –code format in Formats document)
- How to test

  5. Click on 'Snapshot'
  6. Use debugging tools to verify the information contained within the 'snapshot' variable in topSectionEvents.cs

… (Next to Snapshot)

- How it works
  - Creates a file save of the current register information
  - Formatted the same as the information that is read into the GUI (Register info –code format in Formats document)
  - Also creates a local save of the snapshot
- How to test

  7. Click on '…'
  8. Save into a file
  9. Using a text editor, open the file, and verify the contents

Diff

- How it works
    - Loads the last local save of the snapshot
    - Loads all the information into the “oldValue” variable within each register object in allInformation
    - Using the cell formatting of InfoGridLeft and InfoGridRight, the cells are changed red if “oldValue” and “value” are not equal
    - The old mask values are shown in red next to the mask name in the bottom section if they changed. These are determined by using “oldValue”, which is also displayed next to the current value
- How to test
    9. Create a local snapshot
    10. Change at least one register value
    11. Click on ‘Diff’
    12. The changed register should now be highlighted in red, and when clicked on, should display the normal information on the bottom along with the old value for the register, as well as the old value of each mask

… (Next to Diff)

- How it works
    - Loads a save from a file
    - Loads all the information into the “oldValue” variable within each register within allInformation
    - Using the cell formatting of InfoGridLeft and InfoGridRight, the cells are changed red if “oldValue” and “value” are not equal
    - The old mask values are shown in red next to the mask name in the bottom section if they changed. These are determined by using “oldValue”, which is also displayed next to the current value
- How to test
    9. Save a snapshot into a file
    10. Change at least one register value
    11. Click on ‘Diff’

12. The changed register should now be highlighted in red, and when clicked on, should display the normal information on the bottom along with the old value for the register, as well as the old value of each mask

Show Diff?

- How it works
    - After a Diff has been done, it becomes checked and the cells that are different are highlighted in red
    - When unchecked, the cells are no longer highlighted red
    - A simple if statement within the cell formatting event for InfoGridLeft and InfoGridRight controls this functionality
- How to test
    7. Perform a Diff – the changed cells should be highlighted red
    8. Uncheck the box – the changed cells should no longer be highlighted red
    9. Check the box again – the changed cells should be highlighted red

Write Queue

- How it works
    - Writes all of the registers in “changedregisters” ArrayList within BottomSectionEvents
- How to test
    7. Add at least one register to the queue (best would be to add one register with changed masks, one with a changed value, and one with the entire register)
    8. Verify that the registers changed to reflect the changes appropriately
    9. Note: The program will not tell you if a register is read-only, but the write will fail. Also, WinDbg will not write to any PCI register, regardless of whether or not it is writable. Do not mistake this as a bug.

Clear Queue

- How it works
    - Clears the “changedregisters” ArrayList within BottomSectionEvents
- How to test
    7. Change a register and add it to the queue – it should now be highlighted green
    8. Clear the queue – it should no longer be highlighted green
    9. Write the queue – nothing should change

Add to Queue

- How it works
    - If Write Entire Register is not checked, adds the changed masks to the "changedregisters" ArrayList within BottomSectionEvent
    - If Write Entire Register is checked, adds all of the masks to the "changedregisters" ArrayList within BottomSectionEvent
- How to test
    7. Add at least one register to the queue (best would be to add one register with changed masks, one with a changed value, and one with the entire register)
    8. Verify that the registers changed to reflect the changes appropriately
    9. Note: The program will not tell you if a register is read-only, but the write will fail. Do not mistake this for a bug.

Write Changes

- How it works
    - If Write Entire Register is not checked, immediately writes the changed masks
    - If Write Entire Register is checked, immediately writes all of the masks
    - Note: In either method, the write queue will be destroyed
- How to test
    9. Change a register
    10. Write the changes
    11. The list should now reflect the change
    12. Note: The program will not tell you if a register is read-only, but the write will fail. Also, WinDbg will not write to any PCI register, regardless of whether or not it is writable. Do not mistake this for a bug.

Write Entire Register

- How it works
    - If checked, when a Register is written (whether to the queue, or written immediately), the entire Register is written
    - If unchecked, when a Register is written (whether to the queue, or written immediately), only the changed masks are written
- How to test

19. Change a few masks within a Register
20. Change the Register value completely using NVPCI or NVATA
21. Write the changes with the checkbox unchecked
22. Verify that the only the masks that you meant to change were changed
23. Change a few masks within a Register
24. Change the Register value completely using NVPCI or NVATA
25. Write the changes with the checkbox checked
26. Verify that all of the masks were changed to what you wrote
27. Note: The program will not tell you if a register is read-only, but the write will fail. Do not mistake this for a bug.

Port and Global/PCI Buttons

- How they work
  - Change what should currently be displayed, and repopulate the visible lists
  - Note: Changing what is displayed does not refresh the data
- How to test
  5. Click on one that is not currently selected
  6. The display should change to reflect your choice

Appendix G: Command Line Testing

This appendix describes the different command line parameters, and how they work with both NVParse and NVATA. Also included within this file is a technique to test each parameter to ensure their continued functionality after development.

-f FILENAME

- How it works
  - In NVParse
    1. Filename(s) of .reg files, must be specified to run any commands other then help, these reg files should contain registers in memory space.
  - In NVATA
    1. Filename of dump file to use. Defaults to dump_file if none specified.
- How to test
  - In NVParse
    1. Attempt to run with an invalid file name
       a. An error should be returned
    2. Attempt to run with a valid file and in conjunction with the –d command.
       a. Verify that the dump file contents matches what was specified in the ref file.
  - In NVATA
    1. Do not specify this parameter and run a –n command.
       a. Verify that the offset shown for the specified register matches what was specified in dump_file (the default file)
    2. Specify this parameter with another dump file as the argument in conjunction with a –n command
       a. Verify that the offset shown for the specified register matches what was specified in parameter file.

-id ID

- How it works
  - Specifies which device on the system to use for PCI Config Space Registers.

- If this ID is for an AHCI device we automatically set the base offset (-b) to the address in Bar 5
- If no –id parameter is specified the system defaults to the AHCI device on the computer.

- How to test

7. Specify –id # for some valid device on the system with a –p -2 command to show the PCI config space registers.
    - Verify with NVPCI that the values shown are the same.
8. Specify –id <AHCI Unique ID> without a –b
    - Verify the result is the same as specifying –id <AHCI Unique ID> with a –b <Address found in BAR 5>.
9. Do not specify –id for a command
    - Verify the results are the same as if –id <AHCI Unique ID> was called
10. Use debugging tools to verify the information contained within the 'snapshot' variable in topSectionEvents.cs

-b BASEADDRESS

- How it works
    - The base address of the registers you will be writing/reading from, either this or -id is necessary to read and/or write from memory. All memory offsets are relative to this base offset.
- How to test

10. Read a register with a low (zero preferably) offset into memory with a –b BASEADDRESS as part of the command.
    - Verify the hex results match the results returned from a straight memory read of that offset using WinDbg or NVPCI.

-n REGISTERNAME

- How it works
    - Returns information about the specified register and its value in memory.
    - Should work with Port Specific Registers, Global registers and PCI Config Space Registers.
- How to test

13. Call –n REGISTERNAME on a known device with a known base offset and a register with a known offset.
    - Verify the results returned match the known offset.
    - Verify the memory value returned matches the value directly checked with NVPCI or WinDbg.
    - Verify the value specified should break into the masks / flags specified (compare with ref file).
14. Repeat the above verification with a Global, Port, and PCI Config Register.

-p PORT

- How it works
    - Specifies that the results returned should only be from the given port.
    - Specifies that the memory write requested should only affect the given port.
- How to test

13. Do not specify a –n or –p parameter. (NOTE: for NVParse, you must specify –all)
    - Verify that all registers in the dump file are shown. This should include Global, Port and PCI Config registers.
14. Do not specify a –n parameter but do specify a –p parameter.
15. Change at least one register value
16. Click on 'Diff'
17. The changed register should now be highlighted in red, and when clicked on, should display the normal information on the bottom along with the old value for the register, as well as the old value of each mask

Show Diff?

- How it works
    - After a Diff has been done, it becomes checked and the cells that are different are highlighted in red
    - When unchecked, the cells are no longer highlighted red
    - A simple if statement within the cell formatting event for InfoGridLeft and InfoGridRight controls this functionality
- How to test

10. Perform a Diff – the changed cells should be highlighted red

11. Uncheck the box – the changed cells should no longer be highlighted red
12. Check the box again – the changed cells should be highlighted red

Write Queue

- How it works
  - Writes all of the registers in “changedregisters” ArrayList within BottomSectionEvents
- How to test

10. Add at least one register to the queue (best would be to add one register with changed masks, one with a changed value, and one with the entire register)
11. Verify that the registers changed to reflect the changes appropriately
12. Note: The program will not tell you if a register is read-only, but the write will fail. Do not mistake this as a bug.

Clear Queue

- How it works
  - Clears the “changedregisters” ArrayList within BottomSectionEvents
- How to test

10. Change a register and add it to the queue – it should now be highlighted green
11. Clear the queue – it should no longer be highlighted green
12. Write the queue – nothing should change

Add to Queue

- How it works
  - If Write Entire Register is not checked, adds the changed masks to the “changedregisters” ArrayList within BottomSectionEvent
  - If Write Entire Register is checked, adds all of the masks to the “changedregisters” ArrayList within BottomSectionEvent
- How to test

10. Add at least one register to the queue (best would be to add one register with changed masks, one with a changed value, and one with the entire register)
11. Verify that the registers changed to reflect the changes appropriately
12. Note: The program will not tell you if a register is read-only, but the write will fail. Do not mistake this for a bug.

Write Changes

- How it works
  - If Write Entire Register is not checked, immediately writes the changed masks
  - If Write Entire Register is checked, immediately writes all of the masks
  - Note: In either method, the write queue will be destroyed
- How to test
  13. Change a register
  14. Write the changes
  15. The list should now reflect the change
  16. Note: The program will not tell you if a register is read-only, but the write will fail. Do not mistake this for a bug.

Write Entire Register

- How it works
  - If checked, when a Register is written (whether to the queue, or written immediately), the entire Register is written
  - If unchecked, when a Register is written (whether to the queue, or written immediately), only the changed masks are written
- How to test
  28. Change a few masks within a Register
  29. Change the Register value completely using NVPCI or NVATA
  30. Write the changes with the checkbox unchecked
  31. Verify that the only the masks that you meant to change were changed
  32. Change a few masks within a Register
  33. Change the Register value completely using NVPCI or NVATA
  34. Write the changes with the checkbox checked
  35. Verify that all of the masks were changed to what you wrote
  36. Note: The program will not tell you if a register is read-only, but the write will fail. Do not mistake this for a bug.

Port and Global/PCI Buttons

- How they work
  - Change what should currently be displayed, and repopulate the visible lists

    - Note: Changing what is displayed does not refresh the data

- How to test
    7. Click on one that is not currently selected
    8. The display should change to reflect your choice