Niagara Users Guide (4.3)

Firmware Test & Tools Team

Generated on December 12, 2017

Contents

Chapter 1

Introduction

In this manual, we look at Niagara, which is built on the CIL Drive Testing System, and how to use its various capabilities. CIL stands for *Common Interface Layer*. The CIL is named to emphasize the way the software is built: a set of testing components combined to form a full featured tool. The strength of this architecture is that it makes Niagara easy to modify and expand.

We will also look at Niagara's graphical and command line interface. The graphical interface is designed to be intuitive to those who are new to Niagara and convenient for those who have learned the tool. The command line interface is a more powerful tool that incorporates the TCL language. The command line interface is flexible and powerful. In fact, the entire GUI interface is built from the command line features alone!

1.1 Intended Audience

This manual is intended for someone who already understands concepts related to SCSI and drive testing. If terms like CDB are not familiar to you, I recommend learning more about SCSI before proceeding with this manual. Niagara can be very destructive software if misused. Trying out features at random can easily render a drive into an unusable state. Setting up Niagara carelessly on a new system can lead to a system that has its internal drives open for testing (this is a bad thing).

This manual does not assume that you are a programmer. An understanding of a language is beneficial for following some of the examples, however. The early chapters of the manual do not cover programming at all, while the chapters near the latter end of the manual begin to cover TCL, the scripting language integrated into Niagara.

1.2 What This Manual Covers

This manual is not meant to contain every last detail about Niagara and the CIL. The goals of the manual are as follows:

- To give the user an idea of Niagara's basic capabilities
- To inform the user of Niagara's capabilities that can be further understood by reading other manuals.
- To provide a reference to CIL commands.

In addition to this manual, *The CIL Programmers Guide* provides the following information:

- How to write CIL programs in C/C++
- Basic information on how to link the CIL to other languages
- How to extend the CIL's TCL interface using C/C++
- How to create a CIL device driver

In addition to this manual and the programmer's guide, programmers will find it helpful to have reference manuals available for the C++ and TCL languages (the two languages used to build Niagara and the CIL).

1.3 Basic Overview Of CIL Features

In this section we talk about major features of the CIL. Note that the CIL you are using may have more capabilities than the ones bulleted below.

1.3.1 Expandability

The CIL is very expandable. The key to this expandability is the UIL (Universal Interface Layer). This layer is an abstraction that all devices communicate through. This layer is very abstract (it is loosely based on UNIX device driver design). The abstract nature allows the UIL to be expanded to control everything from a SCSI/FCAL device to a serial port, a power supply, or virtually any other device.

There are several layers to the CIL that can be expanded (this is explained in more detail in the *CIL Programmers Guide*). At the upper layers, new procedures can be added to TCL directly from the command line. At lower layers, we find a C interface for adding higher performance TCL commands, a lower-level C interface for writing custom apps (or linking to other languages) and another layer for adding support for new devices. Adding a feature to a layer generally expands the capabilities of all of the upper layers as well. See figure 1.1 for a diagram that shows the CIL's architecture.

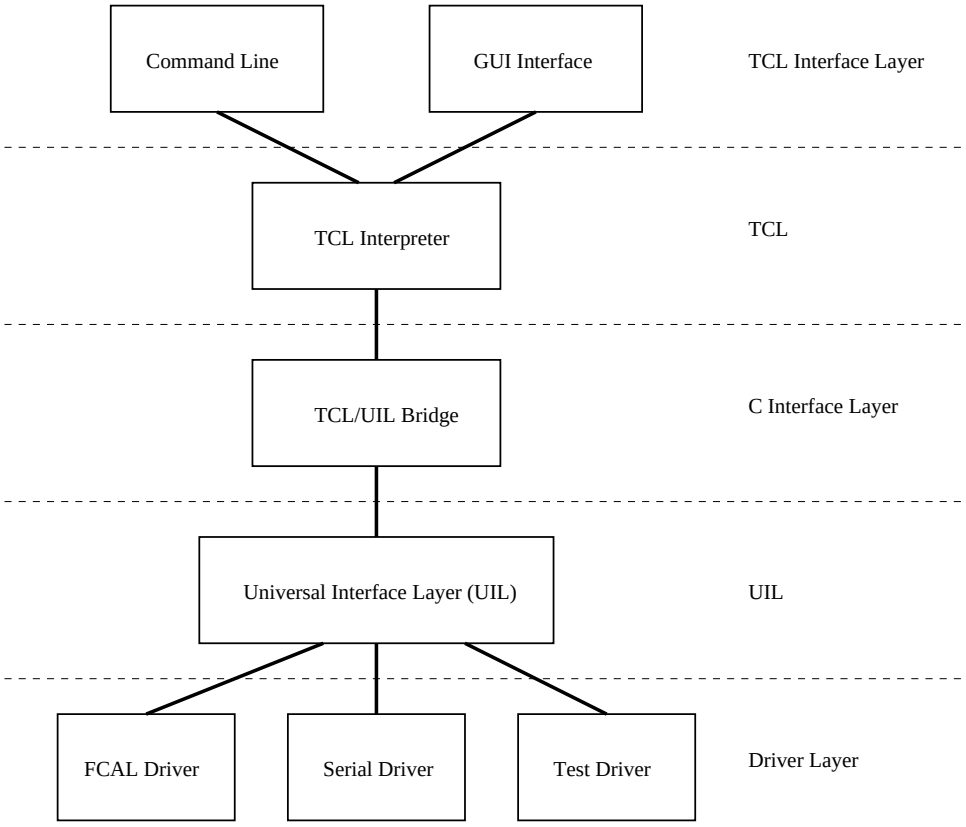


Figure 1.1: CIL Architecture

Most of CIL's expansion is accomplished through dynamic libraries. This is a plus because it means that features can be added to the CIL without reinstalling it.

1.3.2 Portability

One of the reasons that TCL was chosen was due to its portability, all the way to the graphical user interface. This, in combination with all ANSI standard C code, makes the CIL portable across multiple architectures. Most TCL scripts written on one architecture (such as Windows) will run on a different architecture (such as Linux) without change. This means many tests will only have to be written once. The C interface also provides an abstraction that permits a CIL application written in C to compile and run on different platforms without modifying

the source.

1.3.3 Quick Survey Of Current Features (Subject To Expansion)

It is difficult to provide a concrete feature list for the CIL, namely because the CIL is so expandable. Most features that will be needed can created as they are needed. Below, however is a list of currently implemented features:

- GUI Interface
  - Support for all standard CDB and ATA Commands
  - New CDB and ATA commands can be added/modified easily
  - Sense Decoding
  - Customizable quick buttons to speed frequently used features
  - Full suite of standardized tests (Sequential Read, Random Read, Seek Tests, Command Queuing)
  - Additional tests can easily be added
  - Easily perform an operation on multiple drives simultaneously
  - Quick Mode Sense / Select
  - Load Microcode on drive
  - Log dump
  - Buffer Editor
- Command Line Interface
  - Access to all features available in GUI
  - Full TCL scripting language support
  - Full TK GUI extension support
  - Dozens of added TCL commands that are specially designed to ease drive testing
  - Command History
  - "Easy" network socket support (via TCL)
  - DMA support for many drivers
- C Interface
  - Ability to add commands to TCL that fully integrate with the CIL
  - Command Queuing[1]
  - Command Aborts[2]
  - Card and Device Resets[3]
  - Lip control[4]
  - Ability to create new device drivers

[1] Not all drivers support command queuing
[2] Not all drivers support aborts
[3] Not all drivers support resets
[4] Not all drivers support LIPS

Chapter 2

Using The Graphical User Interface

2.1 Overview Of The Main Window

When you first start Niagara, you see two windows, one of which is the terminal (or Console) window. An example terminal window is shown in figure 2.1. Through this window you can enter commands, write scripts, and see the results of your commands. The other window is the GUI front end to Niagara, called the Command window. This window makes certain operations easier and provides customizable buttons and windows to make common repeatable tasks easier to use. An example Command window is shown in figure 2.1. Note that because the GUI can be customized, your Command window might not look exactly like the one shown.

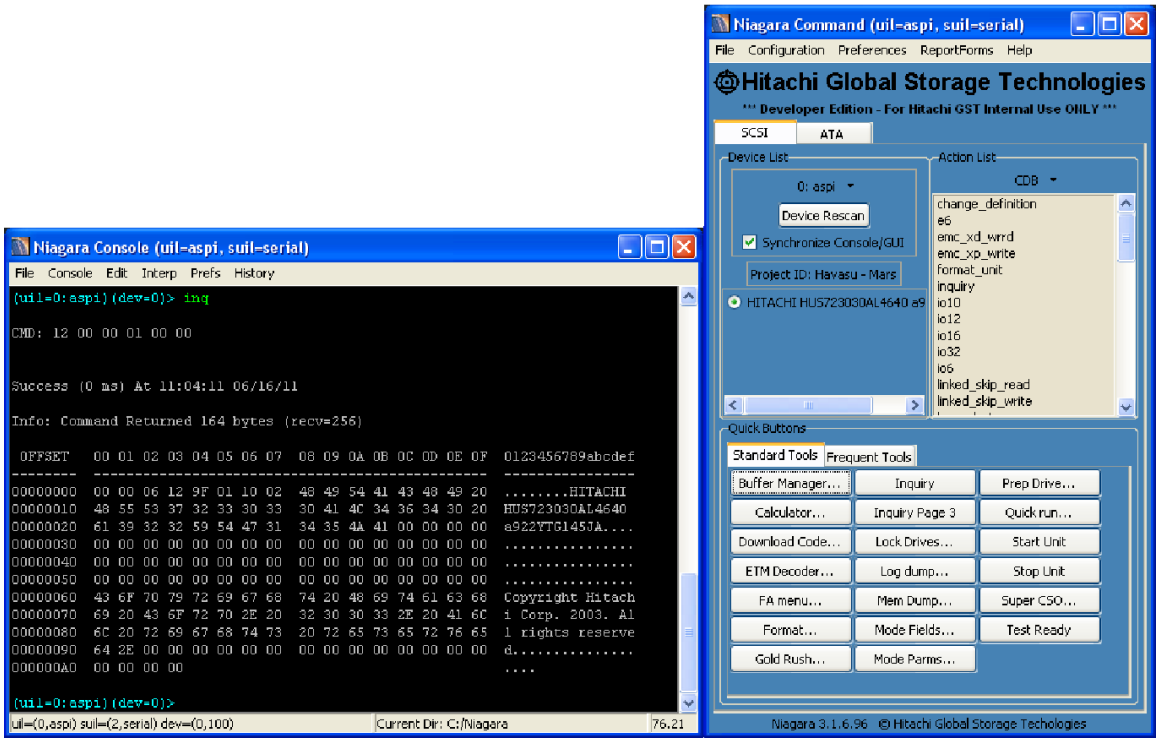


Figure 2.1: Niagara GUI

At the top of the Command window, there are tabs labeled SCSI and ATA. When users are interfacing with a SCSI drive, the SCSI tab should be selected so that all of the SCSI tools are available. Likewise, the ATA tab should be selected when interfacing with an ATA drive so that all of the ATA tools are available.

Each SCSI/ATA tab of the Command window consists of three sections. The Device List section is used to select devices and the current UIL driver. This section is described in section 2.2. The Action List section contains groups of actions that can be executed with the GUI. These groups include:

- **CDB Commands**: This group contains all of the CDB's that can be sent to a drive. CDB Commands are described in more detail in section 2.3 on page 19.

- **ATA/ATA Pass-Through Commands**: This group contains all of the ATA or ATA Pass-Through commands that can be sent to a drive.
- **Special Commands**: This group contains all of the special commands that can be sent to a drive.
- **HDD/SSD Tools**: This group contains additional tools to interact with a drive. A user can also add their own custom tools. Details of how to do this can be found in Appendix 11 on page 109

The Quick Buttons section of the Command window contains a set of quick buttons. This customizable set of buttons are used to access frequently used commands quickly. The default set of buttons contain the following types of functions:

- **Shortcuts to Common CDBs**: Often used CDB's such as "Test Unit Ready" and "Inquiry" are available with the click of a button. See section 2.3.1 for more information.
- **Buffer Management Functions**: These functions provide ways to see information that was transferred from the drive and to set up data to be transferred to the drive. See section 2.4 for more details.
- **Download Code**: This allows for microcode to be uploaded on a drive. See section 2.5 for more details.
- **Format**: This button provides users the chance to format their drive. See section 2.6 for more details.
- **GoldRush**: This button brings up the Gold Rush GUI, which provides a quick test of new firmware. See section **??** for more details.
- **Lock Drives**: This button provides users the chance to lock drives or disable system drive removal. See section 2.8 for more details.
- **Log Bin**: This button provides a log dump of a drive. See section **??** for more information.
- **Mode Selection**: This button brings up a dialog that allows for quick mode sense and selection operations. See section 2.9 for more details.
- **Super CSO**: This button brings up the Super CSO GUI, which contains a set of typical and customizable drive tests. See section 2.10 for more details.

Also in the Quick Buttons section is the Frequent Tools tab, which presents a list of the last tools used by the user.

2.2 Device Selection

The first panel of the Command window offers a way to easily choose what devices you would like to send a particular command to. It is important to note that not all commands work with the chosen device list, most CIL command line commands do not, whereas most of the GUI script commands do. To create scripts that take advantage of device selection please see 11.3.1 on 111. Devices can be selected individually by clicking on their corresponding radiobutton, or they can be selected in a group by unchecking the Synchronize Console/GUI button and selecting as many of the drives' checkboxes as desired. The device list also provides detailed drive information, while each device's vendor id and target id initially displayed, by holding the mouse over a certain device you can find see the rest of the device info in the information status bar at the bottom of the Command window. The information status bar displays serial number, code level, block size, and maximum LBA.

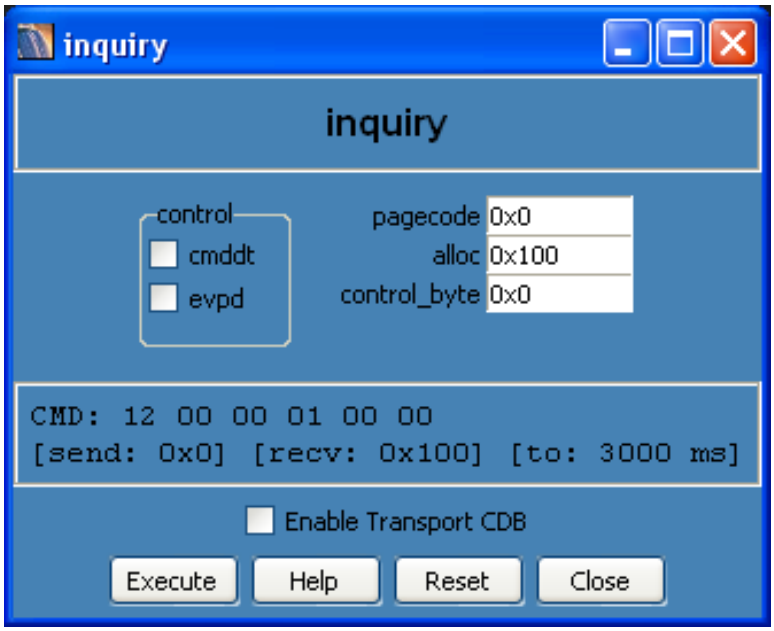


Figure 2.2: Example of a CDB GUI

2.3 Executing A CDB Command

Executing a CDB command through the GUI is trivial. Double click on a CDB in the CDB list-box (in the Action List panel of the Command window). This will bring up another window that is specific to that CDB. This GUI allows you to change various parameters and to send the command to all drives you have selected by clicking on execute. All output will be displayed in the Console window. Clicking on help will print out detailed help information in the Console window for that particular CDB. Cancel simply closes the window. In the CDB command window (figure 2.2 on page 19) you will notice bytes representing the CDB and transfer size just above the execute, help, and cancel buttons. This is the interactive real-time CDB builder. As you change values in the fields you will notice that the CDB builder will update to reflect these changes.

ATA and ATA Pass-Through commands, as accessed from the Action List in the ATA tab, work in the same manner.

2.3.1 Quick CDB Commands

Quick CDB commands are Quickbutton scripts that run a CDB on all selected devices without popping up a GUI, these include start unit and stop unit (both with immediate bit turned off), test unit ready, and inquiry (on page 0 with evpd off). They offer a way to quickly test to see if a drive is responding and to spin it up or down. All output goes to the Console window.

2.4 Buffer Manager

Niagara doesn't only send drive commands, but it is able to keep track of all the various buffer information as well. Three utilities form the entire buffer management suite: Buffer Dump, Buffer Diff, and Buffer Fill.

2.4.1 Buffer Dump

Buffer Dump (figure 2.3 on page 20 allows you to see what is in all of your buffers and save them to a file. The default buffer when the program begins is the receive buffer (Buffer 1). You can set your send and recieve buffers by using the slider and click the respective button. To dump the buffer to the command line window simply click

on the dump button. To save the buffer to a file either type in the filename manually within the file entry or choose browse for a file selection dialog. After configuring the tool to output to a file you can click the dump button to save the buffer to the file.

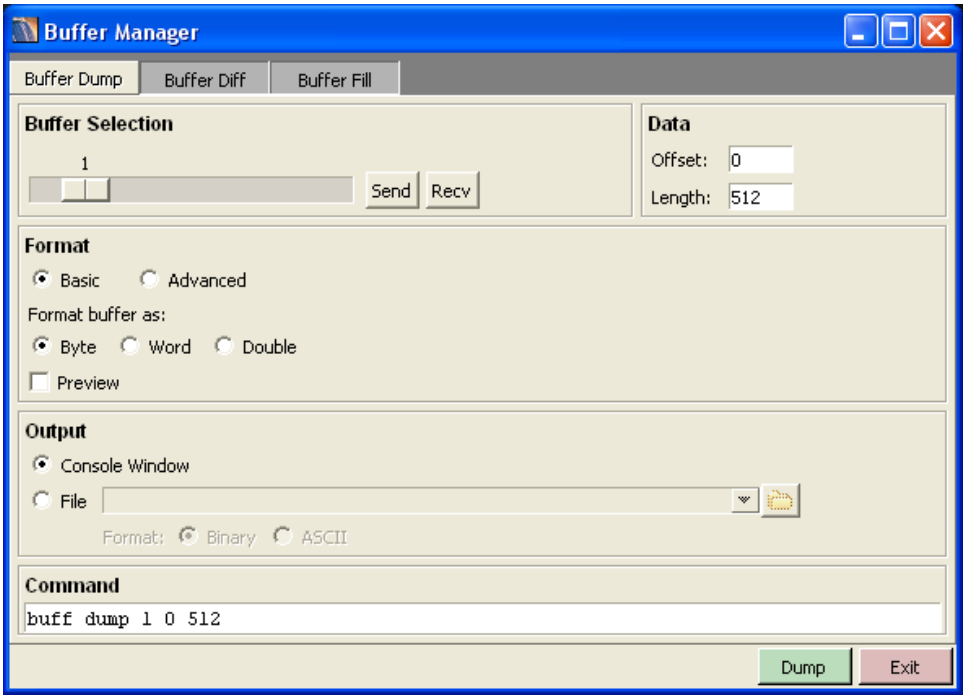


Figure 2.3: Buffer Dump

2.4.2 Buffer Diff

Buffer Diff (figure 2.4 on page 21 allows you to compare the contents of two buffers. You can select either a buffer or a file containing a buffer dump to compare. After selecting the two buffers you can click the diff button to compare the two buffers.

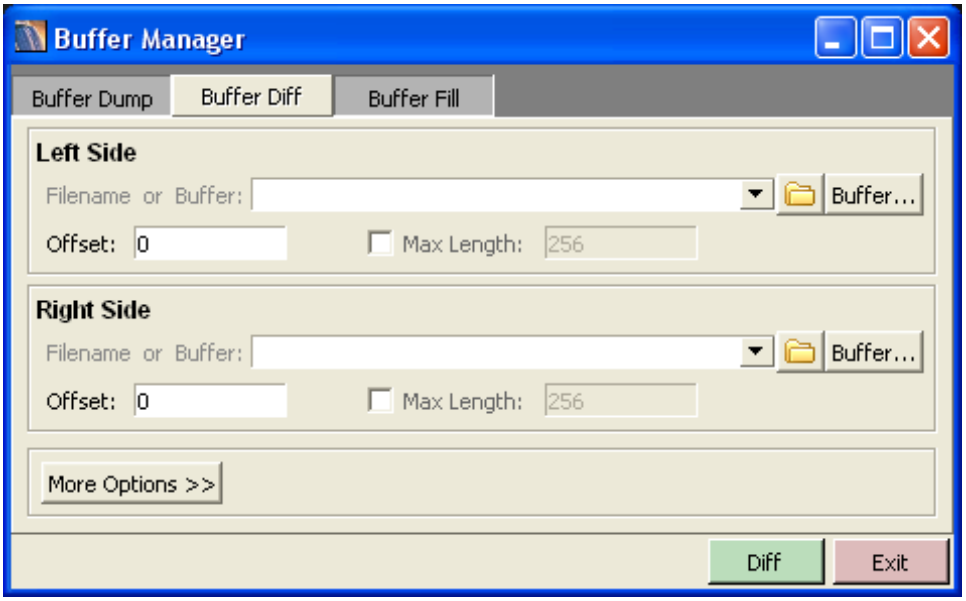


Figure 2.4: Buffer Diff

2.4.3 Buffer Fill

Buffer Fill (figure 2.5 on page 22) does the opposite of Buffer Dump, it allows you to insert values into a buffer. There are seven different ways you can do this.

Type Name	Description
`byte`	fill in one byte continuously for length bytes
`pattern`	fill in a certain pattern of bytes for length bytes
`string`	write a specific string to a buffer
`int`	write a specific integer to the buffer
`random`	fill in random bytes for length bytes
`sequential`	fill in sequential byte order from start to stop for length bytes
`load`	load an entire file into a buffer

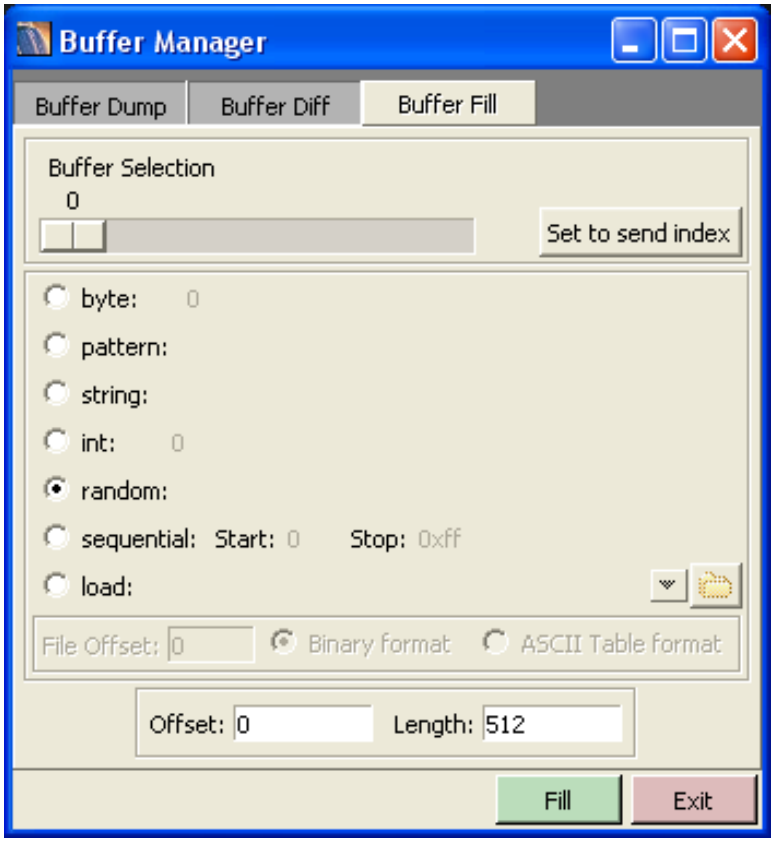


Figure 2.5: Buffer Fill Tool

2.5 Download Code

The Download and Save GUI downloads an entire microcode chunk. (figure 2.6 on page 23) The default file it looks for are .bin files, which is the entire microcode binary chunk. Clicking on Browse opens a file selection window which will put what you have selected in the File Name entry. Pressing the Execute button actually loads the microcode. Using the Advanced tab, the chunk size to download code can be adjusted. Once a drive is completely finished, the next drive undergoes the process.

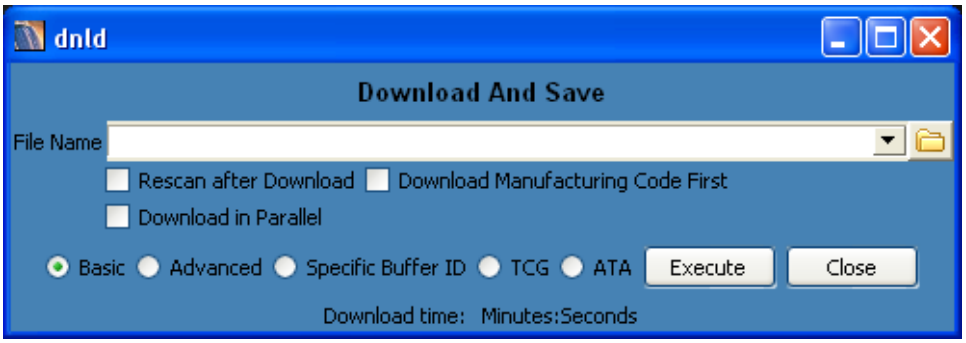


Figure 2.6: Download and Save

2.6 Format

The Format tool can be used to format the drive with different parameters. Figure 2.7 shows the Format GUI. A drive's block size or max LBA can be adjusted by entering new values in the appropriate fields. When you are satisfied with your parameter selection, the Format can be started by selecting the Format button.

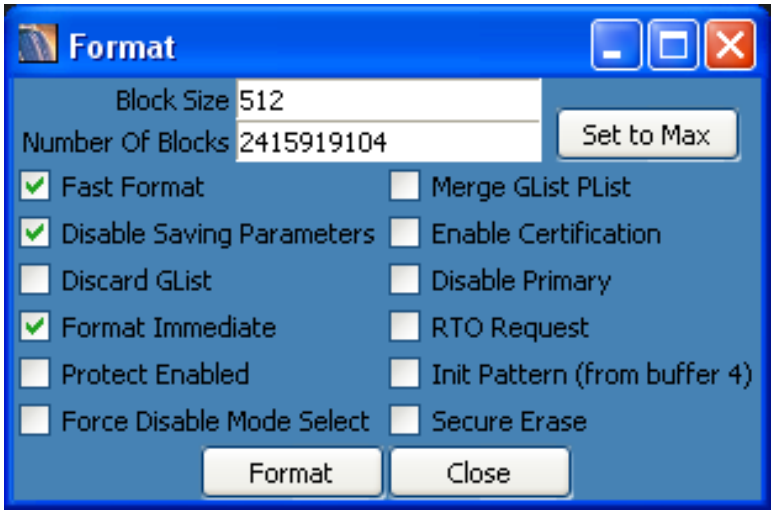


Figure 2.7: Format

2.7 Log Bin

The log bin utility allows you to perform a logdump. Use the logdump utility to parse the files produced by this utility. Figure 2.8 shows the logbin dialog.

Logbin is designed to dump logs from multiple drives at a time (if needed). All of the selected drives in the Command window are dumped (deselect the "Synchronize Console/GUI" checkbox to select multiple drives). Files are dumped in the "base Directory" using the "Filename Pattern". As set, the files will be named with the drive's serial number, code level and index number.

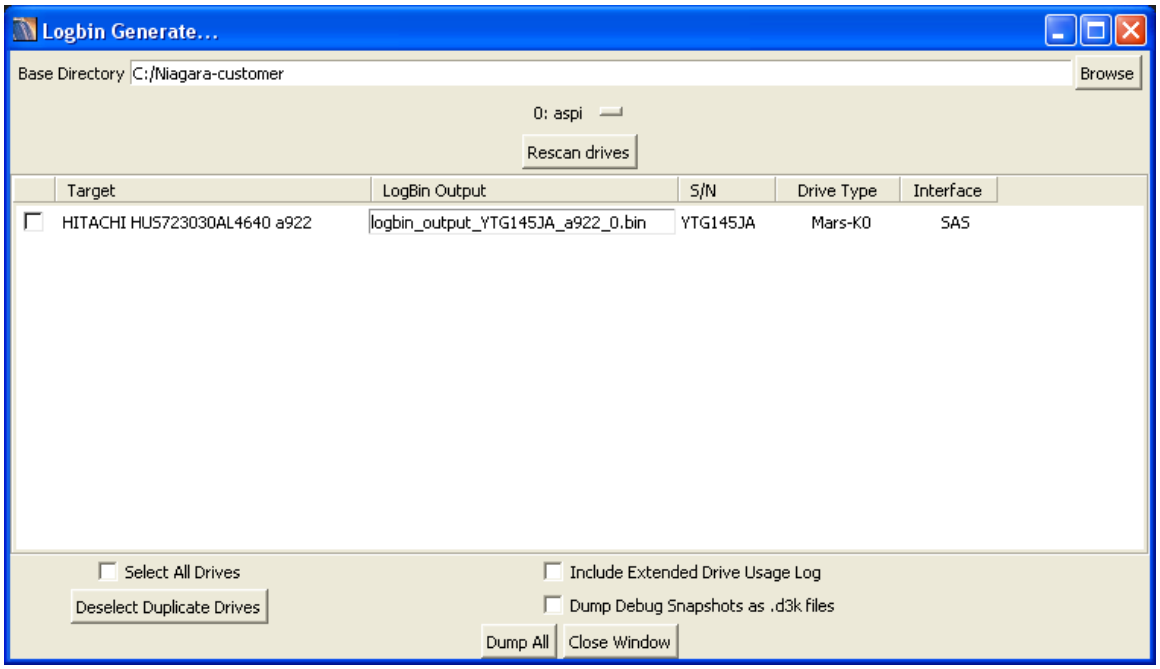


Figure 2.8: Log Bin

2.8 Lock Drives

This tool serves two main functions. First off, each detected device can be locked by Index or Serial Number. Locked devices will not accept drive commands. Secondly, the Remove system drives option can be enabled or disabled to determine if drives that have a Master Boot Record present are removed from Niagara. Figure 2.9 shows the Lock Drives GUI.

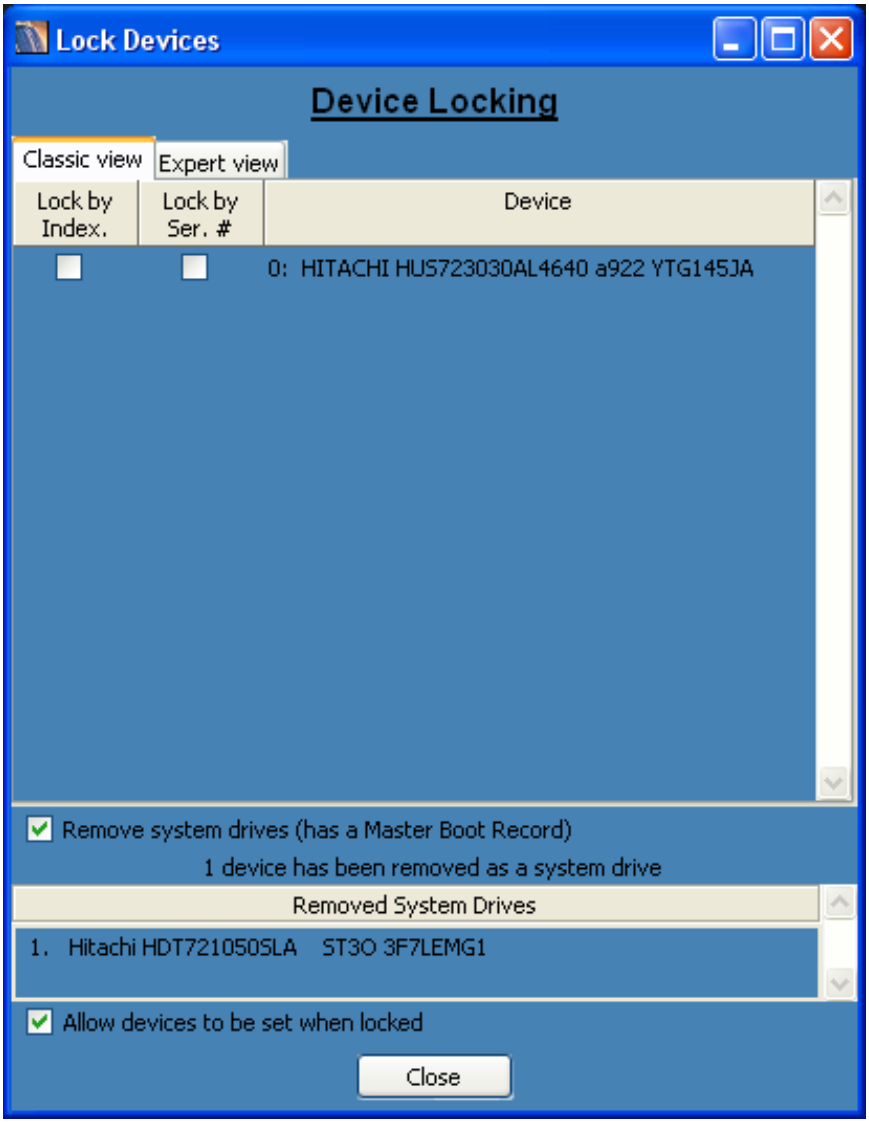


Figure 2.9: Lock Drives

2.9 Mode Selection

One advantage of using a GUI over the command line is that it can present a lot of information in an easier to read format and it can eliminate in-between setup steps. The Mode Parms and Mode Fields GUIs do both (figure 2.10 on page 26); it allows you to look at all the bytes of a mode page on a byte by byte basis, and it eliminates the setup for the mode select, allowing you to simply change the values. When using Mode Parms or Mode Fields on multiple drives, only the first drive will receive the mode sense but all the drives will receive the mode select. Make sure that all the drives selected have compatible mode pages.

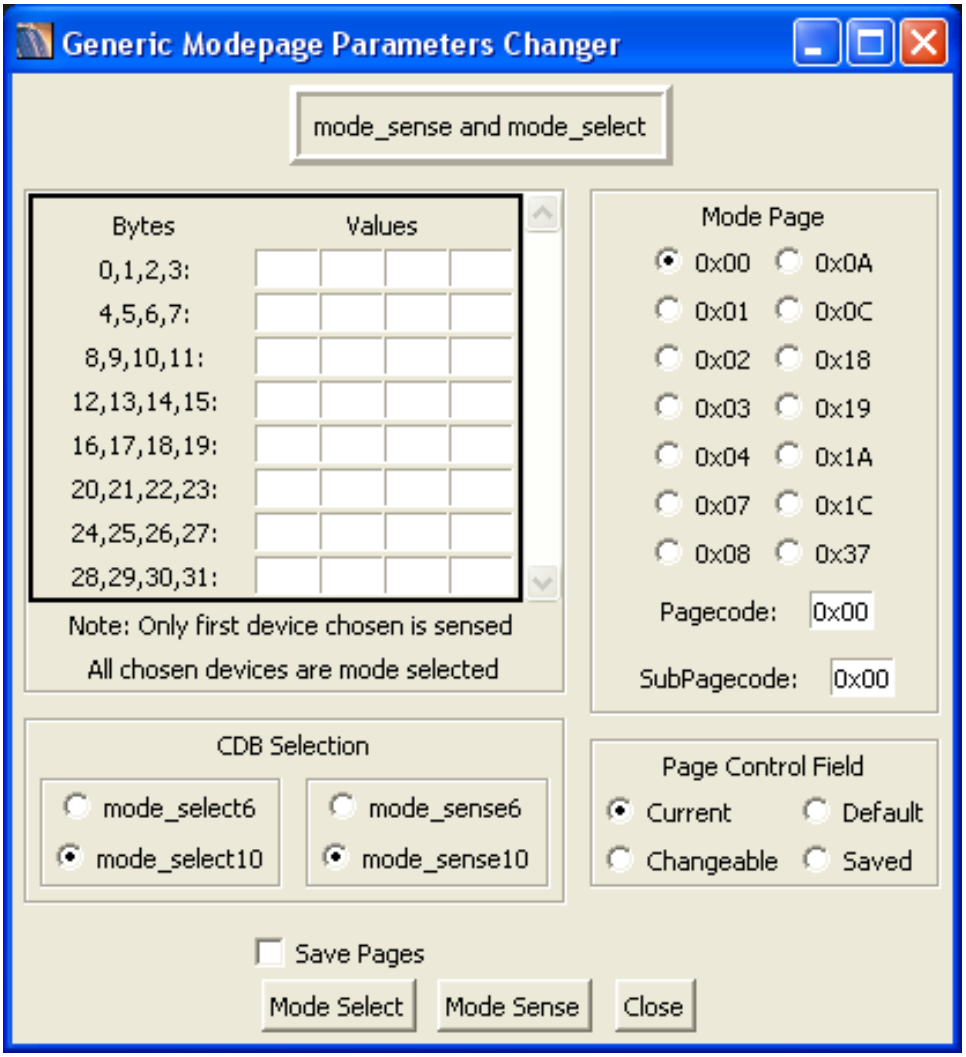


Figure 2.10: Mode Sense Select

2.10 Super CSO

Super CSO is a hard drive workload simulator that is designed to be usable for beginners, highly configurable by intermediate users, and extendable for advanced users. Figure 2.11 shows the main Super CSO GUI. See the Super CSO Tutorial, located in Super CSO's Help menu, for a complete explanation of Super CSO's features.

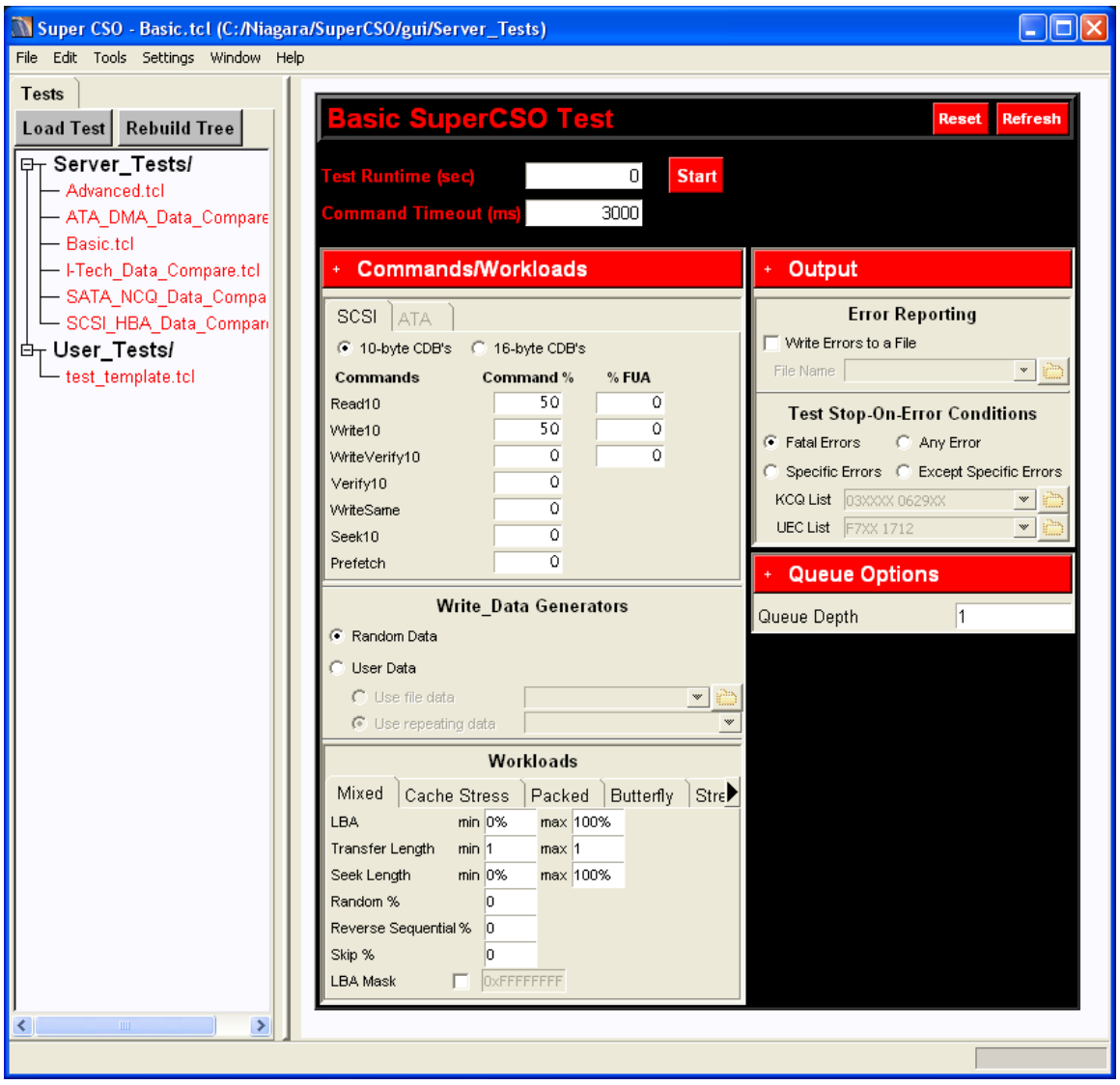


Figure 2.11: Super CSO

Chapter 3

Basic Use Of The Command Line Interface

3.1 Introduction

This chapter explains the basics of using the command line interface. The command line provides a shell like interface to a device and provides more powerful features (such as scripting) than are provided through the Graphical User Interface alone. The drawback to this is that the command interface requires a commitment from the user to learn it (and lacks the intuitive nature of the graphical user interface). Once learned, however, the command interface will likely become the interface of choice for many users.

We will begin by looking at the format for drive (CDB) commands followed by commands to control buffers and other features of the CIL. Note that in some cases, we will be using a bit of TCL code to explain a command in more detail. Feel free to reference chapter 5 on page 74 (which gives an introduction to TCL) whenever you are unclear about what an example is doing.

3.2 Basic Command Entry

To start, we will issue a simple inquiry command. Start the CIL and type the following:

```
inq
```

This will perform a device inquiry and output the results, just as it would from the graphical user interface. You could have also typed:

```
inquiry
```

IMPORTANT: The current device in the command line is completely independent of what is selected in the GUI. Look at the command prompt to see what the current device is. An example command prompt would be `'(uil=0:sg)(dev=0)>'`. This command prompt tells us that `sg` is our current uil and device #0 is our current device. See section 3.4.1 on page 54 for information on how to select devices. Also the command line interface only allows you to interface a single drive at a time. The GUI allow you to select multiple targets.

Most drive commands have 2 forms, a descriptive form (inquiry) and a short form (inq). To see a list of all of the drive commands available type:

```
get_cdb_list
```

This will display a list containing all available device commands. Note that only the long form of the commands are printed. In addition to the commands shown by `get_cdb_list`, you also have access to all of the built in TCL commands (such as `set`, `if`, and `puts`). Through TkCon[1], you also have access to all of the OS commands provided by your shell[2]. TkCon also gives you the option of using the interface as a calculator, for example:

```
expr 5+5
```

[1] TkCon is a free command line extension program, written in TCL by Jeffery Hobbs. For information on the latest version, go to http://tkcon.sourceforge.net

[2] In Linux this includes commands such as ls, cp, and rm. In Windows this Includes commands such as dir, copy, and del

One thing to note is that, if you do not include a decimal point in your calculations, you will get integer math. Fix this by post-fixing `.0` at the end of one of your numbers. Here is an example:

```
> expr 1/3
0

> expr 1/3.0
0.3333333333

> expr 1.0/3.0
0.3333333333

> expr 1.0/3
0.3333333333
```

3.2.1 Getting Help

One useful way to see help is through the tab key. Type the first letter of a command and hit tab. The system will respond by printing all of the possibilities. You can hit tab at any time during command entry. When you hit tab, the following rules are used:

1. If there are multiple possible completions for the command as you have typed it thus far, these completions are shown.
2. If there is only one completion for the command as you typed it thus far, the command will be completed for you.

As an example of the second rule, enter `inqu` and type tab. The rest of the command will be typed for you. To see the short form of a drive command, type the command name followed by -help[3]. For example:

```
inquiry -help
```

This example will output the following data:

```
**************************************

Command Name(s): inquiry,inq

Description: Performs a device inquiry.

Default Parm Order: pagecode, alloc

Buffer Data Sent: <None>
Buffer Data Received: <alloc> Bytes

Parameters:

Name          Range           Default    | Description
--------------------------------------+---------------------------------
-cmddt        (0 or 1)        0x0        | Include Command Support Data
```

[3]This only works with CIL commands, generic TCL commands do not support the -help option

```
-evpd          (0 or 1)         0x0         | Enable Vendor Product Data
-pagecode      (0-0xFF)         0x0         | Page Code
-alloc         (0-0xFFFF)       0x100       | Allocation Length in Bytes
-control_byte  (0-0xFF)         0x0         | NACA | FLAG | LINK
-uil           (0-?)            <current>   | Temporary UIL override
-dev           (0-?)            <current>   | Temporary device index override
-ri            (0-?)            <current>   | Temporary receive buffer override
-cmd_timeout   (0-?)            0           | Persistent timeout override (0=no
                                            |  override)
-dummy         (0-1)            0           | Don't actually send the command
```

Looking at other data returned by `inq -help`, we also see the each parameter has a range and a default value. Any parameters you exclude when calling a command use this default value. With this in mind you can see what the default operation of any command is by using the `-help` option and examining the default value for each parameter.

In the data returned by `inq -help` there is also a section describing the range of each parameter. What happens if you go outside of this range? Basically your result is bit truncated to fit in the desired range. As an example, if a parameter has a range of `0-0xFF` and you enter `0x123456` as a value, the low order 8 bits are used so you will actually be sending 0x56.

3.2.2 Command Options

This brings us to our next topic, command line options. As you probably already know, there is more than one way to call the inquiry CDB. As the GUI allows you to customize the format of the inquiry you will receive, the command line offers the same functionality. Note the parameter list given in the help. Here we see a `-pagecode` option. Let's try using this option to do an inquiry on page 0x80[4]:

```
inq -pagecode 0x80 -evpd 1
```

Note that you need to prefix 0x to all hexadecimal numbers. Any number without a preceding 0x is considered to be in decimal form. Using the above pattern, we can specify any of the options available to the inquiry command. The only problem is that it can be tedious to type in all of the parameter descriptors. Because of this, there is a shortcut to the above form:

```
inq 0x80 -evpd 1
```

This works because `-pagecode` is defined as the first parameter in the `Default Parm Order`[5]. Here is the `Default Parm Order` for the `inquiry` command:

```
Default Parm Order: pagecode, alloc
```

This tells us that:

```
inq 0x80 200 -evpd 1
```

is the same as:

```
inq -pagecode 0x80 -alloc 200 -evpd 1
```

and:

[4] Calling this command as shown in the example will likely throw a check condition from the drive, generally you need to turn on the evpd bit to read this page

[5] To see the Default Parm Order for a CDB command, type the command's name with the -help option

```
inq -alloc 200 -pagecode 0x80 -evpd 1
```

Note above that when you explicitly use the `-pagecode` and `-alloc` options, the ordering is not important. When you exclude the `-pagecode` or `-alloc` option, however, the ordering of the arguments must match those specified in the `Default Parm Order`. You can also mix default parameters with options as shown below:

```
inq 0x80 -evpd 1
```

This is the equivalent to:

```
inq -pagecode 0x80 -evpd 1
```

Once you start using - options, you can not use the shorter form for the remainder of the line. The following, for example, is not allowed:

```
inq -evpd 1 0x80
```

Because we entered the `-evpd` option, the interpreter does not know what option `0x80` relates to. The correct form (assuming you want `0x80` to represent the page code) is `inq 0x80 -evpd 1`. In short, look at the `Default Parm Order` in a command's `-help` description to see which parameters can be entered without using a − option in front. As a final note, because this is a programming language, you can use variables and functions in place of parameters. See the chapter on TCL programming (page 74). Here are some examples:

```
>set page 0x80
>inq $page -evpd 1

>set blocksize 512
>read10 0 [expr 10 * $blocksize]
```

3.2.3 Using Keywords In Place Of Numbers

In place of parameters, you have the option of using keywords. These three commands, for example, do the same thing:

```
inq 0x80 -evpd 1
inq 0x80 -evpd on
inq 0x80 -evpd true
```

The interpreter above works by substituting certain keywords for values. Here is a table of possible substitutions:

Keyword	Substitution
start	1
stop	0
true	1
false	0
on	1
off	0
send	[buff get si]
recv	[buff get ri]

3.3 Table Of CDB Commands

The `get_cdb_list` command can be use to get a list of current commands. However, for you convenience, the list is also given here. Note that this list might not be completely up to date (`get_cdb_list` always is):

Command Name	Short Form	Description
change_definition	chdef	Changes Drive Definition (see page 120)
close_zone	*None*	Performs one or more reset write pointer operations (see page 120)
e6	log_dump	Retrieve internal drive logs. (see page 121)
finish_zone	*None*	Performs one or more reset write pointer operations (see page 122)
format_unit	fmt	Performs a physical format of the drive media. (see page 123)
get_physical_element_status	*None*	The device return status information for physical elements within the device. (see page 123)
inquiry	inq	Performs a device inquiry. (see page 124)
io10	*None*	Send a Generic 10 byte CDB (see page 125)
io12	*None*	Send a Generic 12 byte CDB (see page 126)
io16	*None*	Send a Generic 16 byte CDB (see page 127)
io32	*None*	Send a Generic 32 byte CDB (see page 128)
io6	*None*	Send a Generic 6 byte CDB (see page 130)
log_select	lgsel	Clears statistical information. (see page 130)
log_sense	lgsns	Retrieves statistical data about the drive (see page 131)
logical_depop_fmt_unit	*None*	Initiates spc format to exclude specified heads (see page 132)
logical_depop_inq	*None*	Allows initiator to retrieve info on heads depoped (see page 132)
mode_select10	mdsl10	Specifies device parameters to the target. (see page 133)
mode_select6	mdsl6	Specifies device parameters to the target. (see page 134)
mode_sense10	mdsn10	Reports various device parameters. (see page 134)
mode_sense6	mdsn6	Reports various device parameters. (see page 135)
open_zone	*None*	Performs one or more reset write pointer operations (see page 136)
persistent_reserve_in	pri	Obtains info about persistent reservations. (see page 137)
persistent_reserve_out	pro	Reserves drive for a particular initiator. (see page 137)

continued on next page

continued from previous page

Command Name	Short Form	Description
prefetch	pref	Requests that the drive transfer data to the cache. (see page 138)
prefetch16	pref16	Requests that the drive transfer data to the cache. (see page 138)
read10	r10	Reads blocks of memory from the disk. (see page 139)
read12	r12	Reads blocks of memory from the disk. (see page 140)
read16	r16	Reads blocks of memory from the disk. (see page 141)
read32	r32	Reads blocks of memory from the disk. (see page 142)
read6	r6	Reads blocks of memory from the disk. (see page 143)
read_buffer	rdbuf	agnostic function for memory test. (see page 144)
read_buffer32	rdbuf32	agnostic function for memory test. (see page 145)
read_capacity	rdcap	Returns info regarding the capacity of the drive. (see page 145)
read_capacity16	rdcap16	Returns info regarding the capacity of the drive. (see page 146)
read_defect_data10	rdmap10	Requests that the target transfer medium defect data. (see page 146)
read_defect_data12	rdd12	Requests that the target transfer medium defect data. (see page 147)
read_initialization_pattern	read_init_patt	Requests that device server transfer a single logical block including protection bytes (DIF) of currently Data Pattern set for UNMAPPED LBA to the Data-IN buffer. (see page 148)
read_long	rdlong	ve transfers one block of data to initiator. (see page 149)
read_long16	rdlong16	ve transfers one block of data to intiator. (see page 149)
reassign_blocks	reas	Reassigns specified logical blocks. (see page 150)
receive_diagnostic_results	rcvdg	Sends analysis data to initiator. (see page 151)
release10	rel10	Releases a previously reserved LUN. (see page 151)
release6	rel6	Releases a previously reserved LUN. (see page 152)

continued on next page

continued from previous page

Command Name	Short Form	Description
`remove_element_and_truncate`	*None*	Performs depopulating a storage element and truncating the reported capacity of the media. (see page 152)
`report_dev_id`	`rdi`	Requests that the device server send device information important to the application client (see page 153)
`report_lun`	`rlun`	Returns the known Logical Unit Numbers to the initiator. (see page 154)
`report_provisioning_init_patt`	`rpip`	Requests that the device server transfer the provisioning initialization pattern (see page 154)
`report_supported_opcodes`	`repsupops`	Returns a list of all opcodes and service actions. (see page 155)
`report_supported_tmf`	`repsuptmf`	Returns information on supported TMFs. (see page 156)
`report_timestamp`	`rts`	Requests that the device server return the current value of a device clock. (see page 157)
`report_zones`	*None*	Return the zone structure of the zoned block device. (see page 157)
`report_zones_old`	*None*	Return the zone structure of the zoned block device. (see page 158)
`request_sense`	`sns`	Returns the target's sense data to the initiator. (see page 159)
`reserve10`	`res10`	Used to reserve a LUN for an initiator. (see page 160)
`reserve6`	`res6`	Used to reserve a LUN for an initiator. (see page 160)
`reset_write_pointer`	*None*	Performs one or more reset write pointer operations (see page 161)
`reset_write_pointer_old`	*None*	Performs one or more reset write pointer operations (see page 161)
`rezero_unit`	`rezero`	Requests that the target seek to LBA 0. (see page 162)
`sanitize`	*None*	Performs a sanitize operation. (see page 163)
`security_protocol_in_block`	`sec_in_blk`	Retrieve security protocol information from logical unit. (see page 163)
`security_protocol_in_byte`	`sec_in_byte`	Retrieve security protocol information from logical unit. (see page 164)
`security_protocol_out_block`	`sec_out_blk`	Send security protocol information to logical unit. (see page 165)
`security_protocol_out_byte`	`sec_out_byte`	Send security protocol information to logical unit. (see page 166)
`seek10`	`sk10`	Requests that the drive seek to the specified LBA. (see page 167)

continued on next page

continued from previous page

Command Name	Short Form	Description
`seek10_64lba`	`sk10_64`	Requests that the drive seek to the specified LBA. (see page 167)
`seek6`	`sk6`	Requests that the drive seek to the specified LBA. (see page 168)
`send_diagnostic`	`sndd`	Requests the drive perform a diagnostic. (see page 168)
`set_dev_id`	`sdi`	Requests that the device server send device information important to the application client (see page 169)
`set_initialization_pattern`	`set_init_patt`	Requests that device server transfer a single logical block including protection bytes (DIF) from Data-out buffer. (see page 170)
`set_timestamp`	`sts`	Requests that the device server initialize a device clock (see page 170)
`start_stop_unit`	`ssu`	Starts or stops unit. (see page 171)
`synchronize_cache`	`sync`	Ensures that logical blocks in the cache have their most recent data value recorded on the media (see page 172)
`synchronize_cache16`	`sync16`	Ensures that logical blocks in the cache have their most recent data value recorded on the media (see page 173)
`test_unit_ready`	`tstrdy`	Tests to see if the device is ready. (see page 173)
`unmap`	`um`	Invalidates user data on the disk. (see page 174)
`verify`	`ver`	Asks drive to verify data written on media. If bytechk=0, trans_length is # of blocks to verify internally. If bytechk=1, trans_length is also blocks being sent to drive, so send_length must be made the same as trans_length. (see page 174)
`verify12`	`ver12`	Asks drive to verify data written on media. If bytechk=0, trans_length is # of blocks to verify internally. If bytechk=1, trans_length is also blocks being sent to drive, so send_length must be made the same as trans_length. (see page 175)
`verify16`	`ver16`	Asks drive to verify data written on media. If bytechk=0, trans_length is # of blocks to verify internally. If bytechk=1, trans_length is also blocks being sent to drive, so send_length must be made the same as trans_length. (see page 176)

continued on next page

continued from previous page

Command Name	Short Form	Description
verify32	ver32	Asks drive to verify data written on media. If bytechk=0, trans_length is # of blocks to verify internally. If bytechk=1, trans_length is also blocks being sent to drive, so send_length must be made the same as trans_length. (see page 177)
vu_commit_verify	*None*	Upon receipt of commit verify command, drive updates verify pointer (see page 178)
vu_define_band_type	*None*	fine Band Type (see page 179)
vu_query_band_information	*None*	Returns information associated with bands (see page 179)
vu_query_last_verify_error	*None*	Verify from last verified lba of the band through appropriate EOT (see page 180)
vu_reset_write_pointer	*None*	Reset write pointer for the designated band (see page 180)
vu_set_write_pointer	*None*	Move the write pointer position to start of track of given logical block address (see page 181)
vu_verify_squeezed_blocks	*None*	Verify from last verified lba of the band through appropriate EOT (see page 182)
write10	w10	Writes blocks of memory to the disk. (see page 182)
write12	w12	Writes blocks of memory to the disk. (see page 183)
write16	w16	Reads blocks of memory from the disk. (see page 184)
write32	w32	Writes blocks of memory to the disk. (see page 184)
write6	w6	Writes blocks of memory to the disk. (see page 185)
write_and_verify	wrv	Requests the drive write data and then check it. (see page 186)
write_and_verify12	wrv12	Requests the drive write data and then check it. (see page 186)
write_and_verify16	wrv16	Requests the drive write data and then check it. (see page 187)
write_and_verify32	wrv32	Requests the drive write data and then check it. (see page 188)
write_atomic16	wa16	Atomic Write of blocks of memory to the disk. (see page 189)
write_atomic32	wa32	Atomic writes of blocks of memory to the disk. (see page 190)
write_buffer	writebuf	Used with read_buffer to test drive's memory. (see page 191)

continued on next page

continued from previous page

Command Name	Short Form	Description
`write_buffer32`	`writebuf32`	Used with read_buffer to test drive's memory. (see page 192)
`write_long`	`wrlong`	Requests that the drive write one block of data. (see page 193)
`write_long16`	`wrlong16`	Requests that the drive write one block of data. (see page 194)
`write_same`	`wrs`	Writes one block of data to a number of logical blocks. (see page 195)
`write_same16`	`wrs16`	Writes one block of data to a number of logical blocks. (see page 196)
`write_same32`	`wrs32`	Requests the drive write data and then check it. (see page 197)

3.4 Commands Specific To The CIL

In this section we will explore the different TCL commands that are specific to the CIL. The commands are arranged hierarchically, meaning there are a few base commands that have many options. The basic structure of a CIL command (non CDB) is:

```
<category noun> <action verb> [<subject noun>] ?options?
```

For example: `buff fill zero 0 512` fills a buffer with zeros while `buff dump 0` dumps the contents of buffer #0 to the screen. Below we give a table the commands and a brief description of each. In the sections that follow, we look at each command in more detail. Commands in `type` are accepted commands. Commands in *italics* are not full commands and will prompt you with additional options when entered as-is:

Command Name	Short Form	Description
`ata get`	*None*	Retrieves the info on the ata device (see page 199)
`buff adlerchksum`	*None*	Compute checksum of buffer data using Adler32 algorithm with base (see page 199)
`buff checksum`	*None*	Computes 32-bit checksum of buffer data (see page 200)
`buff compare`	*None*	Compares contents of two buffers. Returns 0 on match, -1 or 1 (see page 200)
`buff copy`	*None*	Copies data from one buffer to another (see page 201)
`buff crc`	*None*	Computes 32-bit CRC of buffer data (see page 202)
`buff diff`	*None*	Diffs contents of two buffers. Returns 0 on match, -1 or 1 (see page 202)
`buff dump`	`bd`	Dumps contents of a buffer in hex (see page 203)

continued on next page

continued from previous page

Command Name	Short Form	Description
`buff e2e`	*None*	Calculates End-To-End protection guard from buffer data. (see page 204)
`buff exists`	*None*	Checks whether a buffer exists for the given index. (see page 205)
`buff fill byte`	`bfb`	Fill a portion of a buffer with a byte value (see page 206)
`buff fill float`	`bff`	Fill buffer with a 4 byte float or 8 byte double precision float (see page 206)
`buff fill int`	`bfi`	Fill 4 buffer bytes with an int value (see page 207)
`buff fill int64`	*None*	Fill 8 buffer bytes with an int value (see page 207)
`buff fill one`	`bfo`	Write 0xff bytes to a buffer (see page 208)
`buff fill patt`	`bfp`	Fill a buffer with a pattern of bytes (see page 208)
`buff fill rand`	`bfr`	Write random bytes to a buffer (see page 209)
`buff fill seq`	`bfs`	Write a sequence of bytes to a buffer (see page 209)
`buff fill short`	`bfsh`	Fill 2 buffer bytes with an 16-bit 'short' value (see page 210)
`buff fill string`	`bfstr`	Write a string to a buffer (see page 211)
`buff fill zero`	`bfz`	Write zeros to a buffer (see page 211)
`buff find`	*None*	Searches for occurances of a data pattern in a buffer (see page 212)
`buff findstr`	*None*	Searches for occurances of a string in a buffer (see page 212)
`buff format`	`bf`	Extracts formatted information from a buffer (see page 213)
`buff get address`	*None*	Returns the address of the buffer. (see page 214)
`buff get count`	*None*	Gets the current buffer count (see page 214)
`buff get dsize`	*None*	Gets the default buffer initial size (see page 214)
`buff get last_si`	*None*	Gets the last send buffer index (see page 215)
`buff get ri`	*None*	Gets the current receive buffer index (see page 215)
`buff get si`	*None*	Gets the current send buffer index (see page 215)
`buff get size`	*None*	Returns the amount of space that is allocated to a specified buffer (see page 216)
`buff gets`	*None*	Extract string(s) from a buffer (see page 216)

continued on next page

continued from previous page

Command Name	Short Form	Description
`buff load`	*None*	Fills a buffer with the contents of a file (see page 217)
`buff peek`	*None*	Get a byte from the buffer (see page 217)
`buff poke`	*None*	Put a single byte into the buffer (see page 218)
`buff print sgl`	*None*	Prints Sgl of the specified buffer index (see page 218)
`buff reset`	*None*	Clears All Buffers (see page 219)
`buff rsa keygen`	*None*	Create an RSA public/private key pair (see page 219)
`buff rsa sign`	*None*	Create an RSA signature using the provided message data and private (see page 220)
`buff rsa verify`	*None*	Verify an RSA signature using the provided message data, public key (see page 220)
`buff save`	*None*	Saves buffer contents into a file (see page 221)
`buff set count`	*None*	Changes the number of available buffers (see page 221)
`buff set dsize`	*None*	Sets the minimum buffer allocation size (see page 222)
`buff set ri`	`bri`	Sets the current receive buffer index (see page 222)
`buff set sgl`	*None*	Sets an SGL (see page 223)
`buff set si`	`bsi`	Sets the current send buffer index (see page 223)
`buff set size`	*None*	Allows the user to set the size of a specified buffer in memory (see page 224)
`console_sync`	*None*	Set whether the Niagara Console and Command Window are (see page 224)
`device count`	*None*	Returns number of devices (see page 224)
`device create`	*None*	Manually adds a device to the device list (see page 225)
`device get allow_set_when_locked`	*None*	Returns the allow_set_when_locked flag of the current device (see page 225)
`device get callback create`	*None*	Returns the callback mapped to the specified device command, if any (see page 226)
`device get callback lock`	*None*	Returns the callback mapped to the specified device command, if any (see page 226)
`device get callback remove`	*None*	Returns the callback mapped to the specified device command, if any (see page 226)
`device get callback rescan`	*None*	Returns the callback mapped to the specified device command, if any (see page 227)
`device get callback set index`	*None*	Returns the callback mapped to the specified device command, if any (see page 227)

continued on next page

continued from previous page

Command Name	Short Form	Description
`device get callback unlock`	*None*	Returns the callback mapped to the specified device command, if any (see page 227)
`device get index`	*None*	Returns the index of the current device (see page 227)
`device get interface`	*None*	Returns the current interface type of the device, if known (see page 228)
`device get last_cmd`	*None*	Returns the last command to be executed (see page 228)
`device get last_cmd_time`	*None*	Returns the command execution time for the last cmd in microseconds (see page 228)
`device get read_xfer`	*None*	Returns the current state of read xfers (see page 229)
`device get receive_count`	*None*	Returns the actual number of data bytes returned by the device (see page 229)
`device get reserved`	*None*	Get the reserve status of the device at index (see page 229)
`device get send_count`	*None*	Returns the actual number of data bytes sent by the device (see page 230)
`device get timeout`	*None*	Returns the timeout of the current device, in milliseconds (see page 230)
`device get xfer_mode`	*None*	Returns the current transfer mode for the current driver. (see page 231)
`device hbareset`	*None*	Resets the HBA driver and rescans the bus (see page 231)
`device info`	*None*	Returns various information about a device (see page 231)
`device info blocksize`	*None*	Returns the block size for a device (see page 232)
`device info channel`	*None*	Returns the channel id for a device (see page 232)
`device info codelevel`	*None*	Returns the code level for a device (see page 233)
`device info host`	*None*	Returns the host id for a device (see page 233)
`device info inq_pages`	*None*	Returns whether certain inquiry pages are valid for a device (see page 234)
`device info lun`	*None*	Returns the lun id for a device (see page 234)
`device info markersize`	*None*	Returns the marker size for a device (see page 235)
`device info maxlba`	*None*	Returns the maximum LBA for a device (see page 235)
`device info mdata_inline`	*None*	Returns whether the device has inline metadata (see page 235)

continued on next page

continued from previous page

Command Name	Short Form	Description
`device info mdata_size`	*None*	Returns the metadata size of a device (see page 236)
`device info name`	*None*	Returns the name of the current device (see page 236)
`device info peripheral`	*None*	Returns the peripheral type of a device (see page 237)
`device info phy_blocksize`	*None*	Returns the physical block size for a device (see page 237)
`device info productid`	*None*	Returns the product id for a device (see page 238)
`device info project`	*None*	Returns the project of a device. (see page 238)
`device info protection`	*None*	Returns true if protection enabled for a device (see page 239)
`device info protection_location`	*None*	Returns the protection location of a device (see page 239)
`device info protection_type`	*None*	Returns the protection type for a device (see page 239)
`device info protocol`	*None*	Returns the protocol of a device (see page 240)
`device info rto`	*None*	Returns true if rto enabled for a device (see page 240)
`device info scsiid`	*None*	Returns the SCSIID for a device (see page 241)
`device info serial`	*None*	Returns the serial id for a device (see page 241)
`device info serial_asic_version`	*None*	Returns the ASIC version of a device connected over serial (see page 242)
`device info target`	*None*	Returns the target id for a device (see page 242)
`device info vendor`	*None*	Returns the vendor id for a device (see page 242)
`device info wwid`	*None*	Returns the wwid for a device (see page 243)
`device islocked`	*None*	Returns 1 if a device is locked, 0 otherwise (see page 243)
`device list`	*None*	Returns a summary list of connected devices (see page 244)
`device lock`	*None*	Locks A Device (Prevents Commands From Being Sent) (see page 244)
`device lock serial`	*None*	Locks a device based on the device serial number (see page 245)
`device remove`	*None*	Removes a device from the device list (see page 245)

continued on next page

continued from previous page

Command Name	Short Form	Description
`device rescan`	*None*	Rescans the SCSI bus or FCAL loop (see page 246)
`device set allow_set_when_locked`	*None*	Allow or disallow a device to be set when its locked (see page 246)
`device set blocksize`	*None*	Manually overides the blocksize for a device (see page 246)
`device set callback create`	*None*	Sets a callback for a command (see page 247)
`device set callback lock`	*None*	Sets a callback for a command (see page 247)
`device set callback remove`	*None*	Sets a callback for a command (see page 248)
`device set callback rescan`	*None*	Sets a callback for a command (see page 248)
`device set callback set index`	*None*	Sets a callback for a command (see page 249)
`device set callback unlock`	*None*	Sets a callback for a command (see page 249)
`device set index`	`dsi`	Sets the current device (see page 250)
`device set markersize`	*None*	Manually overides the HA Marker Size (MRKSZ) for a device (see page 251)
`device set maxlba`	*None*	Manually overides the maxlba for a device (see page 251)
`device set phy_blocksize`	*None*	Manually overides the physical blocksize for a device (see page 251)
`device set project`	*None*	Change the project settings of the drive (see page 252)
`device set protocol`	*None*	Manually overides the protocol (see page 252)
`device set read_xfer`	*None*	Used to turn off data transfers to buffer to improve performance (see page 253)
`device set reserved`	*None*	Set the reserve/release status of the device at index (see page 253)
`device set serial`	*None*	Change the serial number of a drive (see page 254)
`device set timeout`	*None*	Sets the timeout value for the current device (see page 254)
`device set xfer_mode`	*None*	Sets the transfer mode for the current driver (see page 255)
`device unlock`	*None*	Unlocks A Device (see page 255)
`device unlock serial`	*None*	Unlocks a device based on the drive serial number (see page 256)
`disable_niagara_log`	*None*	Disable the Niagara Log (see page 256)
`enable_niagara_log`	*None*	Enable the Niagara Log (see page 256)

continued on next page

continued from previous page

Command Name	Short Form	Description
`encode`	*None*	Encrypts a TCL file (see page 257)
`eparse`	*None*	Parses an encrypted UIL specfile, adding command definitions (see page 257)
`err_str`	*None*	Converts an ec error code to a string (see page 257)
`esource`	*None*	A modified version of source that can execute .stc files (see page 258)
`fcal abort_task_set`	*None*	Sends an abort_task_set frame to the device (see page 258)
`fcal abts`	*None*	Sends an abort sequence frame to the device (see page 259)
`fcal clear_aca`	*None*	Clears autocontingent allegiance condition (see page 259)
`fcal clear_task_set`	*None*	Sends an clear_task_set frame to the device (see page 259)
`fcal lip_reset`	*None*	Sends a LIP followed by a port and process login to all devices (see page 259)
`fcal port_login`	*None*	Sends an PORT_LOGIN frame to the device (see page 260)
`fcal process_login`	*None*	Sends an process_login frame to the device (see page 260)
`fcal reset`	*None*	Sends a reset followed by a port and process login to all devices (see page 260)
`fcal target_reset`	*None*	Clears command queue for all initiators and returns Unit Attention (see page 261)
`fcal term_task`	*None*	Terminates task (see page 261)
`feedback asynccqe`	*None*	Turn NVMe Async CQEs on and off (see page 261)
`feedback color`	*None*	Turns embedded color codes on and off (see page 262)
`feedback default`	*None*	Sets feedback to the default level (see page 262)
`feedback maxlen`	*None*	Changes number of buffer bytes returned by a command (see page 262)
`feedback min`	*None*	Sets feedback to the minimum level (see page 263)
`feedback pop`	*None*	Pop feedback state off the feedback stack (see page 263)
`feedback push`	*None*	Pushes feedback state onto the feedback stack (see page 263)
`feedback showatafis`	*None*	Turn ATA return FIS verbose on and off (see page 264)
`feedback showcdb`	*None*	Turn CDB/ATA command verbose on and off (see page 264)

continued on next page

continued from previous page

Command Name	Short Form	Description
`feedback showcqe`	*None*	Turn NVMe CQ Entry verbose on and off (see page 264)
`get_cil_list`	*None*	Get all CIL commands. (see page 265)
`get_cq_str`	*None*	Returns error string for a specified cq. (see page 265)
`get_kcq_str`	*None*	Returns error string for a specified kcq. (see page 265)
`init`	*None*	No Description Given (see page 266)
`niagara_log_puts`	*None*	Log a message to the Niagara log file (see page 266)
`nvme dump_cid`	*None*	Dumps a specific command (see page 267)
`nvme dump_cq`	*None*	Dumps a completion queue (see page 267)
`nvme dump_sq`	*None*	Dumps a completion queue (see page 268)
`nvme get callback reset`	*None*	Returns the callback mapped to the specified device command, if any (see page 268)
`nvme get controller`	*None*	Returns a controller's ID (see page 268)
`nvme get cq_ids`	*None*	Returns a list of completion queues (see page 269)
`nvme get cq_size`	*None*	Return size of completion queue (see page 269)
`nvme get device_index`	*None*	Returns a controller's device index (see page 270)
`nvme get drain_cq`	*None*	Return whether the completion queue will automatically receive (see page 270)
`nvme get last_cid`	*None*	Returns the last CID (see page 270)
`nvme get last_dword`	*None*	Returns the last completion queue entry's dword (see page 271)
`nvme get last_dword0`	*None*	Returns the last completion queue entry's dword0 (see page 271)
`nvme get last_dword1`	*None*	Returns the last completion queue entry's dword1 (see page 272)
`nvme get last_err_logpage`	*None*	Returns the last error log page (see page 272)
`nvme get last_status`	*None*	Returns the status fields of the last completion queue entry (see page 272)
`nvme get page_size`	*None*	Returns the page size. (see page 273)
`nvme get register`	*None*	Reads an NVMe register (see page 273)
`nvme get sq_ids`	*None*	Returns a list of submission queues (see page 274)
`nvme get sq_size`	*None*	Return size of completion queue (see page 274)
`nvme reset`	*None*	Performs NVMe resets (see page 274)
`nvme reset queue`	*None*	Performs Queue Level reset (see page 275)
`nvme set callback reset`	*None*	Sets a callback for a command (see page 276)

continued on next page

continued from previous page

Command Name	Short Form	Description
`nvme set drain_cq`	*None*	Set whether the completion queue will automatically receive (see page 276)
`nvme set page_size`	*None*	Set the page size (see page 276)
`nvme set register`	*None*	Writes an NVMe register (see page 277)
`parse`	*None*	Parses a UIL specfile, adding command definition (see page 277)
`pcie get config`	*None*	Reads PCIe config space (see page 278)
`pcie get driver`	*None*	Reports active drivers per drive (see page 278)
`pcie get speed`	*None*	Reports a drive's speed (see page 279)
`pcie get status`	*None*	Reports if a drive is up or down (see page 279)
`pcie get width`	*None*	Reports a drive's width (see page 280)
`pcie set config`	*None*	Writes PCIe config space (see page 280)
`perfcnt clicks`	*None*	Gives time with high-resolution timer normalized to us. (see page 281)
`perfcnt count`	*None*	Gives system clock ticks with high-resolution timer. (see page 281)
`perfcnt delay`	*None*	Delays the system a given amount of microseconds. (see page 282)
`perfcnt freq`	*None*	Gives the system's resolution in ticks per second. (see page 282)
`pqi dump_iq`	*None*	Dumps a inbound queue (see page 282)
`pqi dump_oq`	*None*	Dumps an outbound queue (see page 283)
`pqi get register`	*None*	Reads a PQI Register (see page 283)
`pqi set register`	*None*	Writes a PQI register (see page 283)
`qctl get auto_incr`	*None*	Returns 1 if auto tag increment is enabled, 0 otherwise. (see page 284)
`qctl get ignore_queue_full`	*None*	Returns the current ignore_queue_full status (see page 284)
`qctl get max_depth`	*None*	Returns current setting for the maximum queue depth. (see page 284)
`qctl get num_queued`	*None*	Returns the number of commands that have been issued, but haven't (see page 285)
`qctl get num_waiting`	*None*	Returns the number of queued commands that have returned status. (see page 285)
`qctl get tag_type`	*None*	Returns current tag type. Types include simple, ordered and head. (see page 286)
`qctl idx_info`	*None*	Returns status information for the specified queue index. (see page 286)
`qctl recv`	*None*	Waits for and retrieves the next available command from the device. (see page 286)
`qctl recv all`	*None*	Waits for and retrieves all outstanding commands from the device. (see page 287)

continued on next page

continued from previous page

Command Name	Short Form	Description
`qctl recv tag`	*None*	Waits for and retrieves commands from the current device until the (see page 287)
`qctl send`	*None*	Sends internal table of commands built up in stacked queuing mode (see page 288)
`qctl set auto_incr`	*None*	Turns the auto increment of tag ids on/off. (see page 288)
`qctl set ignore_queue_full`	*None*	Sets queue full on or off (see page 289)
`qctl set max_depth`	*None*	Sets the maximum queuing depth. (see page 289)
`qctl set next_tag`	*None*	Sets the tag id of the next command to be sent. (see page 290)
`qctl set tag_type`	*None*	Sets tag type of next command. (see page 290)
`qctl table_info`	*None*	Returns status information for all received commands. (see page 291)
`qctl tag_info`	*None*	Returns status information for the specified queue tag. (see page 291)
`qmode concurrent`	*None*	Puts current UIL in concurrent queuing mode (see page 292)
`qmode disable`	*None*	Puts current UIL in non-queued mode (see page 293)
`qmode info`	*None*	Returns Niagara's current queuing mode (see page 293)
`qmode pcie`	*None*	Puts current UIL in pcie queuing mode (see page 293)
`qmode stacked`	*None*	Puts current UIL in stacked queuing mode (see page 294)
`rand`	*None*	Generates a random unsigned float (see page 295)
`rand addhist`	*None*	Adds a histogram constraint to a random number generator (see page 295)
`rand close`	*None*	Closes a random source (see page 296)
`rand float`	*None*	Generates a random unsigned float (see page 296)
`rand frange`	*None*	Generates a random floating point between two floating points (see page 297)
`rand int`	*None*	Generates a random unsigned integer (see page 297)
`rand open`	*None*	Creates a new random source (see page 298)
`rand range`	*None*	Generates a random unsigned integer between two integers (see page 299)
`rand seed`	*None*	Seeds a random channel (see page 299)
`rand showhist`	*None*	Displays histogram settings for a random channel (see page 300)

continued on next page

continued from previous page

Command Name	Short Form	Description
`randlba`	*None*	Returns a number between zero and maxlba - blksize (see page 300)
`sas abort_task_set`	*None*	Sends an abort_task_set frame to the device (see page 301)
`sas clear_aca`	*None*	Clears autocontingent allegiance condition (see page 301)
`sas clear_task_set`	*None*	Sends an clear_task_set frame to the device (see page 302)
`sas get_pod_address`	*None*	Gets the sas pod address (see page 302)
`sas get_sas_address`	*None*	Gets the sas device address (see page 302)
`sas get_speed`	*None*	Returns the current speed of the SAS bus (see page 302)
`sas link_reset`	*None*	Sends the notify SAS primitive to allow unit start (see page 303)
`sas lun_reset`	*None*	Logical Unit Reset (see page 303)
`sas nexus_reset`	*None*	Performs a SAS I-T Nexus reset sequence (see page 303)
`sas notify`	*None*	Send NOTIFY Primitive to allow auto unit start (see page 304)
`sas notify_epow`	*None*	Sends a SAS EPOW Notify (see page 304)
`sas phy_disable_offline`	*None*	Disables SAS Phy ports that are offline (see page 304)
`sas phy_enable_disabled`	*None*	Enable SAS Phy ports that are disabled (see page 304)
`sas phy_pulse_disable`	*None*	Pulses the Phy disable bit (see page 305)
`sas phy_reset`	*None*	Performs a SAS phy reset sequence (see page 305)
`sas power_manage`	*None*	Sets the SAS Power Management for the current device (see page 305)
`sas query_async_event`	*None*	Sends a Query Asynchronous Event TMF (see page 306)
`sas query_task_set`	*None*	Sends a Query Task Set TMF (see page 306)
`sas reset`	*None*	This performs a SAS hard reset sequence (see page 306)
`sas set_sas_address`	*None*	Sets the SAS Address for the current device (see page 306)
`sas set_speed`	*None*	Sets the interface speed (see page 307)
`sata comreset`	*None*	Issues a COM reset (see page 307)
`sata fis`	*None*	Access to the value of D2H register on current AHCI port (see page 308)
`sata fis get`	*None*	Access to the value of D2H register on current AHCI port (see page 308)

continued on next page

continued from previous page

Command Name	Short Form	Description
`sata fis get count`	*None*	Access to the value of D2H register on current AHCI port (see page 309)
`sata fis get device`	*None*	Access to the value of D2H register on current AHCI port (see page 309)
`sata fis get error`	*None*	Access to the value of D2H register on current AHCI port (see page 310)
`sata fis get lba`	*None*	Access to the value of D2H register on current AHCI port (see page 310)
`sata fis get lba_ext`	*None*	Access to the value of D2H register on current AHCI port (see page 311)
`sata fis get status`	*None*	Access to the value of D2H register on current AHCI port (see page 311)
`sata get`	*None*	Retrieves the info on the target sata device (see page 312)
`sata get active`	*None*	Retrieves the info on the target sata device (see page 312)
`sata get control`	*None*	Retrieves the info on the target sata device (see page 313)
`sata get error`	*None*	Retrieves the info on the target sata device (see page 313)
`sata get status`	*None*	Retrieves the info on the target sata device (see page 314)
`sata get_auto_tags`	*None*	Gets the state of automatic tags (see page 314)
`sata get_clear_ncq_err`	*None*	Gets the state of automatic tags (see page 315)
`sata get_cmd_fencing`	*None*	Gets the state of fencing ncq and non-NCQ commands (see page 315)
`sata get_connected_drive_mask`	*None*	Gets the information that which port in target AHCI ID(Default (see page 315)
`sata get_ncq_sequencing`	*None*	Gets the state of sequencing NCQ commands (see page 316)
`sata get_preserve_tags`	*None*	Gets the state of driver command tag assignment (see page 316)
`sata get_selected_port`	*None*	Gets selected drive's port number (see page 316)
`sata get_signature`	*None*	Gets the PxSIG value. PxSIG contains value of first count register (see page 317)
`sata get_speed`	*None*	Gets the speed of Drive (see page 317)
`sata get_starting_tfd`	*None*	Gets the TFD(Task File Data) value when spin up.TFD contains value (see page 317)
`sata phy_disable_offline`	*None*	Disables SATA Phy ports they are offline (see page 317)

continued on next page

continued from previous page

Command Name	Short Form	Description
`sata phy_enable_disabled`	*None*	Enable SATA Phy ports that are disabled (see page 318)
`sata pm`	*None*	This command is used to changed the interface sleep state (see page 318)
`sata pm aggressive`	*None*	This Command is used to change the hba's aggressive sleep state (see page 318)
`sata read_port_regs`	*None*	Reads the port registers for the target device (see page 319)
`sata set_auto_tags`	*None*	Sets the state of automatic tags (see page 319)
`sata set_clear_ncq_err`	*None*	Sets the state of automatic ncq error clearing (see page 320)
`sata set_cmd_fencing`	*None*	Sets the state of fencing ncq and non-NCQ commands (see page 320)
`sata set_ncq_sequencing`	*None*	Sets the state of sequencing NCQ commands (see page 320)
`sata set_preserve_tags`	*None*	Sets the state of driver command tag assignment (see page 321)
`sata set_speed`	*None*	sets the speed of Drive (see page 321)
`sata soft_reset`	*None*	Sends a soft reset to the device (see page 322)
`sata srst`	*None*	Sends a soft reset to the device (see page 322)
`scsi abort`	*None*	Sends an abort message to the device (see page 322)
`scsi abort_tag`	*None*	Sends an abort tag message to the device (see page 322)
`scsi clear_queue`	*None*	Clears the current queue. (see page 323)
`scsi device_reset`	*None*	Sends a SCSI device reset message to the current target (see page 323)
`scsi id_mode`	*None*	Changes the identify message the initiator uses (see page 323)
`scsi ppr_mode`	*None*	Changes when PPR negotiations will occur (see page 324)
`scsi ppr_mode_parms`	*None*	Sets the desired Parallel Protocol Request (PPR) parameters (see page 324)
`scsi reset`	*None*	Preforms a SCSI bus reset (see page 325)
`scsi sync_mode`	*None*	Changes when synchronous negotiations will occur. (see page 325)
`scsi sync_mode_parms`	*None*	Sets the desired synchronous period and offset. (see page 325)
`scsi wide_mode`	*None*	Changes when width negotiations will occur. (see page 326)

continued on next page

continued from previous page

Command Name	Short Form	Description
`scsi wide_mode_parms`	*None*	Sets the desired data width for use in all subsequent wide (see page 326)
`sop get cdb_iu_type`	*None*	Gets the default IU wrapper for CDBs (see page 327)
`sop set cdb_iu_type`	*None*	Sets the default IU wrapper for CDBs (see page 327)
`transport_cdb get always_on`	*None*	Gets whether the transport_cdb flag is always on. (see page 327)
`transport_cdb get desc_format`	*None*	Gets whether descriptor format is enabled for transport cmds. (see page 328)
`transport_cdb get pad_boundary`	*None*	Gets the transport_cdb padding boundary. (see page 328)
`transport_cdb get padding`	*None*	Gets whether the transport_cdb padding flag is enabled. (see page 328)
`transport_cdb get protocol`	*None*	Gets the transport_cdb APT protocol. (see page 328)
`transport_cdb set always_on`	*None*	Sets whether the transport_cdb flag is always enabled. (see page 329)
`transport_cdb set desc_format`	*None*	Enable descriptor format when issuing the sns option of a transport (see page 329)
`transport_cdb set pad_boundary`	*None*	Sets padding boundary for commands that use the transport_cdb flag. (see page 329)
`transport_cdb set padding`	*None*	Enables padding of ATA/APT DMA cmds that use the transport_cdb (see page 330)
`transport_cdb set protocol`	*None*	Sets the transport_cdb APT protocol. (see page 330)
`uil count`	*None*	Returns the number of available UIL drivers (see page 331)
`uil create`	*None*	Creates (and initializes) a new UIL driver (see page 331)
`uil get autosense`	*None*	Returns 1 if autosense is active, 0 otherwise (see page 331)
`uil get buffsize`	*None*	Returns the driver's internal data buffer size (see page 332)
`uil get callback create`	*None*	Returns the callback mapped to the specified uil command, if any (see page 332)
`uil get callback remove`	*None*	Returns the callback mapped to the specified uil command, if any (see page 332)
`uil get callback set index`	*None*	Returns the callback mapped to the specified uil command, if any (see page 332)
`uil get err_info`	*None*	Returns err info specific to the UIL driver (see page 333)
`uil get filter`	*None*	Returns the filter flag of the current driver (see page 333)

continued on next page

continued from previous page

Command Name	Short Form	Description
`uil get index`	*None*	Gets the index of the current UIL driver (see page 333)
`uil get max_xfer_len`	*None*	Gets the driver's max transfer length (see page 334)
`uil get mpio enabled`	*None*	Gets mpio enabled setting (see page 334)
`uil get mpio path_select`	*None*	Gets mpio path_select setting (see page 334)
`uil get mpio tm_path_select`	*None*	Gets mpio tm_path_select setting (see page 335)
`uil get speed`	*None*	Gets interface speed (see page 335)
`uil get timeout`	*None*	Gets the default timeout value for this uil driver (see page 335)
`uil get version`	*None*	Gets interface version (see page 335)
`uil info`	*None*	Returns a brief description about the current uil driver (see page 336)
`uil list`	*None*	Returns a list of active uil drivers (see page 336)
`uil load`	*None*	Loads and initializes a UIL/TCL extension (see page 336)
`uil message`	*None*	Sends a string message to a UIL and expects a string reply (see page 337)
`uil name`	*None*	Returns the name of the current uil driver (see page 337)
`uil remove`	*None*	Removes (and uninitializes) a UIL driver (see page 338)
`uil set autosense`	*None*	Activate / Deactivate Sense For The Current Driver (see page 338)
`uil set callback create`	*None*	Sets a callback for a command (see page 339)
`uil set callback remove`	*None*	Sets a callback for a command (see page 339)
`uil set callback set index`	*None*	Sets a callback for a command (see page 340)
`uil set index`	`usi`	Sets the current UIL driver (see page 340)
`uil set loglevel`	*None*	Sets the logging level for the current UIL driver (see page 341)
`uil set mpio enabled`	*None*	Sets mpio enabled setting (see page 342)
`uil set mpio path_select`	*None*	Sets mpio path_select setting (see page 342)
`uil set mpio tm_path_select`	*None*	Sets mpio tm_path_select setting (see page 342)
`uil set speed`	*None*	Sets interface speed (see page 343)
`uil set timeout`	*None*	Sets the timeout value for the all devices (see page 343)

continued on next page

continued from previous page		
Command Name	Short Form	Description
`validate_commands`	*None*	Validate target command definition file or validate command (see page 344)

3.4.1 The device command

The device command offers two basic features:

- Manage multiple devices
- Obtain Basic Information About A Device (Or Devices)

Device Information

There are 2 ways to get device information, `device info`, and `device list`. Typing `device info` returns a detailed list about the currently connected device. Here is an example:

```
> device info

Information For Device #0
VendorID:       HP      CD-Writer+ 8290 1.3C
Serial#:
CodeLevel:
HostID:         0
SCSIChannel:    0
DeviceID:       0
LUN:            0
BlockSize:      2048
MaxLBA:         296398
```

You also have the option of looking at these parameters individually. For example:

```
> device info blocksize
2048
```

This can be useful in scripts. Options available are `blocksize`, `channel`, `codelevel`, `host`, `lun`, `maxlba`, `serial`, `target`, and `vendor`.

The `device list` command lists all of the connected devices and provides some information about each one. Here is an example:

```
> device list

Indx| Vendor Info                    | hst chn id  lun Max LBA  Blk Size
----+--------------------------------+---------------------------------
0   | HP      CD-Writer+ 8290 1.3C   | 0   0   0   0   296398   2048
1   | SanDisk ImageMate II    1.30   | 1   0   0   0   15680    512
----+--------------------------------+---------------------------------
```

Changing The Target Device

The `device count` command returns the number of devices as an integer. The final two commands having to do with devices are `device get index` and `device set index`. These commands are used to determine and set the current device. All cdb commands (such as `inq`) are directed at the current device. Here is an example that sets to device #1 (the Sandisk ImageMate in the list given in the last section):

```
device set index 1
```

SHORTCUT: The format given above is descriptive but a pain to type. Because of this, you can also use the shortcut given below:

```
dsi 1
```

Hot swapping Devices

If your driver supports it, you can use the `device rescan` command to scan for "Hot" loop or bus changes. If your current UIL driver does not support dynamic bus rescans, you will get an error message.

Note: If you are unsure if a specific uil driver supports rescans, try to rescan *before* changing the loop or bus. Then if the driver supports rescans, go ahead and change the loop, re-scanning again after the change.

IMPORTANT: If you are using the `sg` driver in Linux, you should do a bus rescan *immediately after changing the loop*. Changing the loop and exiting the CIL without re-scanning the loop can confuse the SCSI subsystem in Linux (requiring a reboot to fix it).

Locking A Device

Sometimes a CIL testing workstation has its internal drives visible to the CIL. This is a potential disaster in that a careless `write` command can corrupt data on the host computer. Because of this, the CIL allows you the option of locking out a device. To lock a device, use the following command:

```
device lock 0
```

Where `'0'` is the index of the device you wish to lock. Use the `device list` command to get the index for a particular device. Once a device is locked, the CIL will not allow you to send any CDB's to it. For example the following will give and error:

```
#lock the device
device set index 0
device lock 0

#this gives an error
inq

#unlock the device
device unlock 0

#now inquiry works
inq
```

Above we used the `device unlock` command. This command can be used to unlock a device that has been locked. The final command having to do with device locks is `device islocked`. The `device islocked <index>` command will return a 1 if `index` is locked, otherwise it will return a zero. Here is an example:

```
device islocked 1
```

IMPORTANT: If an internal device on your machine is visible to the CIL, what you should really do is lock the drive in the CIL's startup script (see x for details on the CIL's startup script). This will ensure that the device is locked by default.

3.4.2 The buff Command

Buffer Basics

Many of the CDB commands you execute involve the transfer of data to or from the device. The `read10` CDB, for example, transfers from the device to the computer while `write10` transfers data from the computer to the device. To see if a CDB command transfers data, use the `-help` option:

```
> read10 -help

**************************************

Command Name(s): read10,r10,rd10

Description: Reads blocks of memory from the disk.

Default Parm Order: lba, translen, rdprotect

Buffer Data Sent: <None>
Buffer Data Received: <translen> Blocks

Parameters:

    Name           Range            Default     | Description
    ---------------------------------------------+---------------------------------
    -rdprotect     (0-7)            0x0         | EndToEnd RdProtect field
    -dpo           (0 or 1)         0x0         | Disable Page Out
    -fua           (0 or 1)         0x0         | Force Unit Access
    -reladr        (0 or 1)         0x0         | Relative Block Address
    -lba           (0-0xFFFFFFFF)   0x0         | Logical Block Address
    -translen      (0-0xFFFF)       0x1         | Transfer Length in Blocks
    -control_byte  (0-0xFF)         0x0         | NACA | FLAG | LINK
    -uil           (0-?)            <current>   | Temporary UIL override
    -dev           (0-?)            <current>   | Temporary device index override
    -ri            (0-?)            <current>   | Temporary receive buffer override
    -cmd_timeout   (0-?)            0           | Persistent timeout override (0=no
                                                |  override)
    -dummy         (0-1)            0           | Don't actually send the command
```

If we look at "Buffer Data Received" above, we see that a buffer (the receive buffer) is receiving data from the drive. Where does this data go? The data goes into a special buffer that is managed by the CIL software. The `buff` command provides access to both this buffer and the send buffer that is used with commands such as `write10`[6]

Let us look at the contents of a buffer. Start the CIL (start it from scratch to make sure the Buffers are in their default condition) and type the following:

```
inq
```

You will see some data dumped to the screen. This data that you see is also stored in a buffer. To see the buffer, enter the following:

[6]There are actually more than 2 buffers managed by the CIL. We will cover this feature later...

```
buff dump 1
```

The buffer should match that returned by the `inq` command (except that `buff dump 1` displays more data than `inq` did).

The `buff dump` command's format is:

```
buff dump <buffer #> ?offset? ?length?
```

The `offset` and `length` parameters are both optional and are used to get more control over what part of the buffer you see. The default offset is zero bytes. The default length is 255 bytes. The default send buffer index is zero. The default receive buffer index is 1. You can also say:

```
buff dump send
buff dump recv
```

This is the equivalent to:

```
buff dump [buff get si]
buff dump [buff get ri]
```

SHORTCUT: You can use the `bd` keyword in place of `buff dump`.

Although the ability to dump the hex data for a buffer is often sufficient, sometimes it is valuable to parse out specific data of a buffer into specific numbers and strings. This is where the `buff format` command comes into use. The generic syntax of the command is:

```
buff format <buffer #> <format string> ?range1? ?range2? ?...?
```

This command uses the same type of format string used by `printf` in C. In case you are not familiar with this, here is an example:

```
#this example parses Inquiry data
inquiry
buff format 1 "The Vendor ID Is: %s" {8 8}
```

This command first does an inquiry to fill the receive buffer (which we are assuming is index 1) with inquiry data. After the buffer contains data, the `buff format` command is used to extract some information. Specifically, the `%s` tag identifies that we should substitute a string. The next field, `{8 8}`, specifies the offset and length within the buffer to use for constructing the string. In addition to strings, you can also specify decimal integers (`%d` and `%u`), hexadecimal integers (`%x` and `%X`) and others. Look up use of the `printf` C function for details. You can also have multiple substitutions per line. Below we use the `read_capacity` cdb with `buff format` to print out the max lba and block size returned by the command:

```
read_capacity
buff format 1 "Max LBA: %u (0x%x hex)" {0 4} {0 4}
buff format 1 "Transfer Length: %u (0x%x hex)" {4 4} {4 4}
```

Generating Buffer Data

In the section above we discussed some ways to examine buffer data. This is fine for `inq`, `read10`, and other commands that transfer data from the drive. For commands that transfer data to the drive, such as `write10` and `mode_select` the `buff fill` command is provided. The `buff fill` command can fill a buffer with many different types of data. The first type of fill is to fill the buffer with all zeros. Here is an example:

```
buff fill zero 1 0 512
```

This command fills buffer 1 with zeros, starting at an index of zero and continuing for 512 bytes. To see the changes you made to a buffer, use the `buff dump` command:

```
buff fill rand 2 0 1024
buff dump 2
```

In this example we used the `buff fill rand` command to write 1024 random bytes to buffer 2. In the line after this we look at the contents of what we wrote. In addition to `buff fill zero` and `buff fill rand`, there is `buff fill one` to fill the buffer with all `0xFF` bytes. We can also fill the buffer with any other byte using the `buff fill byte` command:

```
#format for this command is buff fill byte <index> <offset> <length> <byte>
buff fill byte 0 50 1000 0xCC
buff dump 0 1050
```

This example fills the buffer 0 with 1000 `0xCC` bytes, starting at an offset of 50. The `buff dump` command below it shows that the first 50 bytes of the buffer were left undisturbed. We can also fill the buffer with sequential bytes:

```
#here we use the default of 0 - 0xFF
buff fill seq 4 0 512
#here we choose to use a range of 10 - 50
buff fill seq 4 0 512 10 50
```

The final two fill commands are useful for writing a set of bytes to the buffer. The `buff fill patt` command writes a repeating pattern of bytes to the send buffer. For example to fill buffer 6 with the pattern `0x0A`, `0x0B`, `0x0C`, `0x01`, `0x02`, `0x03`, we would use the command:

```
buff fill patt 6 512 0xa 0xb 0xc 1 2 3
```

This command can also be useful for setting a string of hex digits that repeat once, such as in setting mode pages. For this operation, simply set the length to the same as the number of bytes you specify. Here is an example that sets 5 bytes:

```
buff fill patt 0 0 5 0x00 0x00 0x80 0xFF 0x50
```

Our final fill command, `buff fill string`, is used to write an arbitrary string to the drive. Here is the format:

```
buff fill string <index> <offset> <string>
```

Here we write a simple message to buffer 0:

```
buff fill string 0 0 "Hello!"
```

We can also send the results of a TCL command. Here is an example that fills a buffer with random data, then encodes the first few bytes of the data with the current time and date:

```
buff fill rand 0 0 512
buff fill string 0 0 [clock format [clock seconds]]
```

Another convenient function is `buff fill int`. This function can be used to quickly fill in a buffer parameter that is 4 bytes wide. Here is an example:

```
buff fill int 0 0 $val
```

Note that the bytes above are stored in "big endian" format (the same as the SCSI standard). If you want to store in "little endian" you can use the `buff fill byte` in combination with shift and mask. Here is an example procedure that implements little endian int fill:

```
proc fill_int_le {bi addr val} {
  #"little endian" format
  #bfb is the short form of "buff fill byte"

  bfb $bi $addr            1 [expr $val & 0xff]
  bfb $bi [expr $addr + 1] 1 [expr ($val >> 8) & 0xff]
  bfb $bi [expr $addr + 2] 1 [expr ($val >> 16) & 0xff]
  bfb $bi [expr $addr + 3] 1 [expr $val >> 24]

  return "4 bytes written to buffer: $bi"
}
```

You can also use the "shift and mask" trick as shown above to fill 2 and 3 byte values into the buffer.

For our final note, we can use `send` and `recv` in place of our buffer index. Here is an example:

```
#this command
buff fill rand send 0 512
#is the same as
buff fill rand [buff get si] 0 512
```

SHORTCUT: You can use the following commands in place of their longer winded versions:

Command Name	Shortcut
`buff fill byte`	`bfb`
`buff fill int`	`bfi`
`buff fill one`	`bfo`
`buff fill patt`	`bfp`
`buff fill rand`	`bfr`
`buff fill seq`	`bfs`
`buff fill string`	`bfstr`
`buff fill zero`	`bfz`

Loading And Saving Buffer Data

In addition to generating data on the fly, we also have the option of loading and saving buffers to a disk file. The format for these two commands is:

```
buff load ?filename? [?index?] [?offset?]
buff save ?filename? ?index? ?offset? ?length?
```

The `buff load` commands loads a disk file into memory. The `?offset?` variable is optional and specifies an offset into the buffer other than the default of zero. Here is an example that loads a buffer with a file named `README.txt` and then writes the first 512 bytes of the buffer to block 100 of our current device[7].

[7] Assuming our device's blocksize is 512

```
buff load README.txt send
buff dump send ;# Optional: Look at the buffer quickly
write10 100 1
```

Note that the buffer data is loaded into the send buffer. We could have used a numerical index (such as 0) in place of `send` above. The `buff save` command is performs the reverse of the `buff load` command. Here is an example that gets an inquiry and saves the results to a file named `inqdata.bin`:

```
inq 0x0 96
buff save inqdata.bin 1 0 96
```

The `buff save` command saves the data from the current receive buffer (the default receive buffer is buffer #1). Because the `inq` command puts its data in the same buffer, saving the information is straight forward.

Comparing buffers

Often it is useful to compare the contents of one buffer with that of another. In the next section we will see how to change the send and receive buffers to allow for elaborate data compares. In this section, however, we will stick to the tried and true setup of the send buffer being buffer 0 and the receive buffer being buffer 1. To compare two buffers for equality, use the `buff compare` command. This command has the following syntax:

```
buff compare <buff index a> <buff index b> <amount>
```

The compare command, as described above, compares buffer `a` and `b` for `amount` bytes. If the buffers compare equally, a 0 is returned, otherwise a 1 is returned. Below is an example that writes random data to a device, then reads in the data and compares:

```
for {set i 0} {$i < 50000} {incr i} {
  #create random data and write it
  buff fill rand 0 0 512
  w10 $i 1

  #read back the data and compare
  r10 $i 1
  if {[buff compare 0 1 512]} {
    puts "miscompare at $i"
  }
}
```

Using More Than Two Buffers

The CIL supports a number of buffers. There are 10 buffers by default. Additionally you can specify support for to up to 4 billion buffers if needed (although you will run out of memory first). The CIL's startup mode is to point the send buffer to buffer 0 and the receive buffer to buffer 1. We are not limited to this setup, however. Each of these 10 (default) buffers is actually a generic buffer. This means that each of these buffers can be a send or receive buffer or both. When is this useful? One example is when you want to hold information from multiple CDB commands in memory at once. A specific example would be comparing information between two drives. Here is an example of this in action:

```
for {set i 0} {$i < 1000000} {incr i 100} {
  device set index 0
  buff set ri 1
  r10 $i 100
```

```
  device set index 1
  buff set ri 2
  r10 $i 100

  if {[buff compare 1 2 51200]} {
    puts "miscompare at $i"
  }

}
```

Above we see the use of the `buff set ri` command. This command sets the receive buffer index. By setting it to 2 different values for each drive, we set ourselves up for a convenient buffer compare. Another useful command is `buff set si`. This sets the buffer used for the send index. This command is useful for writing data to a drive that we read with a different command. Here we modify the above script to swap the first million blocks between two drives (note how we avoid any memory copying in this example):

```
for {set i 0} {$i < 1000000} {incr i 100} {

  #read data
  device set index 0
  buff set ri 1
  r10 $i 100

  device set index 1
  buff set ri 2
  r10 $i 100

  #write data to opposite devices
  device set index 0
  buff set si 2
  w10 $i 100

  device set index 1
  buff set si 1
  w10 $i 100
}
```

With a little creativity, you can probably come up with more useful applications for multiple buffers. In some cases it might also be useful to copy a buffer, perhaps to store a master copy of data. The command for this is `buff copy`. The command has 2 formats. The first format is straight forward:

```
buff copy 1 0
```

The command above would copy the information stored in buffer 1 to buffer 0. The original contents of buffer 0 are overwritten. The second format is more flexible:

```
buff copy source_index dest_index ?source_offset? ?dest_offset? ?length?
```

Say you wanted to copy bytes 60-100 of buffer 0 to bytes 200-240 of buffer 1. This would be the command:

```
buff copy 0 1 60 200 40
```

The above says: "Copy buffer 0 to buffer 1, start at offset 60 in buffer 0, start at offset 200 in buffer 1, copy 40 bytes".

SHORTCUT: Because you might want to change buffers often from the command line and the `buffer set` command involves a lot of keystrokes two shortcuts are provided. Instead of typing `buff set ri`, you can type `bri`. Also, instead of typing `buff set si`, you can type `bsi`. Here is an example:

```
bsi 0
```

3.4.3 The uil command

The CIL is built with a layered architecture. This means that when you issue a command such as `read10`, you are actually sending down a formatted block of data to a lower level driver which handles your request. Often this will be a driver for the SCSI or FCAL devices you are testing. The software is not limited to this setup however. Other possibilities for the "driver" include:

- A Device "Simulator" which is meant to emulate future prototype hardware
- A fake "trace" device to help you debug your scripts
- A network driver to allow you to test and control drives remotely
- A driver for a different device type, such as the serial port, loop analysers, or other testing equipment

Having all of these device types available under the same commands means that your same scripts can potentially work with all of these driver types. This "driver plug in" capability allows for a flexible testing environment. The command that allows you to control this environment is `uil`.

The acronym `uil` stands for *Universal Interface Layer*. It is the mechanism that allows the capabilities described above. Using the `uil` command you can:

- Obtain a list of loaded drivers
- Switch Between Drivers
- Load A New Driver
- Load a TCL Extension
- Unload A Driver

First we will look at how to see what drivers are loaded. Enter the command:

```
uil list
```

This will show the names and index for each driver loaded as well as the number of devices recognized by the driver. Note that different drivers can recognize the same device (Although they may do so to different capacities). To find out the driver you are currently using type:

```
uil info
```

This tells you the driver that commands are currently being directed at. To change this driver, use the `uil set index` command. For an example, let us assume that we have a driver named `test` (which we probably do) and that it's index is 1. Type the following:

```
uil set index 1
inq
```

Because we set ourselves to use the test driver, the inquiry was never performed on an actual device. Use `uil set index 0` to return to the default driver.

SHORTCUT: The `uil set index` command, although descriptive, can be a nuisance to type. Because of this, you can also use the `usi` shortcut for the same effect. For example:

```
usi 1
```

Other useful uil commands are beyond the scope of this manual. Here is a brief description of what they are:

- `uil create`: This command loads a new driver into memory and initializes it. This is also generally done by the startup scripts although experienced users can also use the command to load drivers as they are needed.
- `uil remove`: This command removes a driver added by `buff create`.
- `uil load`: This command is used to add C extensions to the CIL interface. These extension are generally in the form of high performance TCL commands.
- `uil count`: This command simply returns the number of drivers currently loaded.

3.4.4 The feedback command

When you type a command such as `inquiry` you generally see a hex dump of what the command returned. You also see the CDB that was sent to the drive and whether the command was successful. Normally this amount of information is sufficient. Sometimes, however, you will want to customize the feedback returned to you. This is accomplished through the `feedback` commands.

One good reason for using the `feedback` command is to improve the performance of your TCL scripts. When you perform an inquiry or a read, the hex dump returned to you takes a bit of time to create. Not requiring the computer to generate this hex dump can improve performance. The command to take feedback to a minimum level is:

```
feedback min
```

This will set the feedback to the minimum amount. Basically, after executing a `feedback min` command, commands such as `read10` will only indicate if they were successful. You can still see the buffer contents using the `buff dump` command, but these contents will no longer be printed automatically. To return the feedback to default type:

```
feedback default
```

This will return the level of feedback to its default setting. Another feedback command is:

```
feedback maxlen
```

This command sets the maximum number of buffer bytes that are printed. When a command returns data the command will either dump out the number of bytes returned or `maxlen`, whichever is smaller. Here are some examples to clarify:

```
feedback maxlen 10
inq 0 96 ;# prints 10 bytes, because that is the value of maxlen
inq 0 150 ;# prints 10 bytes, because that is the value of maxlen

feedback maxlen 100
inq 0 96 ;# prints 96 bytes, because that is all inq returns here
```

```
inq 0 150 ;# prints 100 bytes, because that is the value of maxlen

feedback maxlen 1000
inq 0 96 ;# prints 96 bytes, because that is all inq returns here
inq 0 150 ;# prints 150 bytes, because that is all inq returns here
```

The default value for `maxlen` is 255. Another parameter you can customize is the display of the CDB that was sent. There are two options for this:

```
feedback showcdb 1 ;# show the CDB
feedback showcdb 0 ;# don't show it
```

Another command is `feedback color`. If you are using a terminal that accepts color codes, `feedback color 1` will color the text differently for you, depending on what type of message it is (success messages are green, the CDB is shown in purple, etc.). If you are not using a terminal that supports color codes, `feedback color 1` will print what looks like garbage around your messages. If this is happening, turn off "color" with `feedback color 0`.

Often times when running a script it is useful to call `feedback min` to speed up the execution of the script. The problem is that calling `feedback default` when the script is over might not restore the feedback in a way that the user wants. Perhaps the user prefers feedback set up in a custom way... This problem is handled by the `feedback push` and `feedback pop` commands. The `feedback push` command saves the current feedback settings on a "feedback stack". The `feedback pop` command can then be used to retrieve those values. Here is an example of how a script should use these commands to restore user settings:

```
proc seq_reads {} {

  feedback push ;# save settings
  feedback min ;# boost performance

  for {set i 0} {$i < 50000} {incr i} {
    r10 $i 1
  }

  feedback pop ;# restore original settings

}
```

3.4.5 The randlba Command

The `randlba` command exists for convenience and speed. The command format is simple:

```
randlba ?maxblk? ?channel?
```

This command returns a random number between zero and the maximum `lba` available on the device. The `maxblk` is optional (and has a default value of 1). The purpose of the `maxblk` argument is for when you want to read multiple blocks. Setting `maxblk` to a value greater than one assures that `randlba` will not return a value that causes the disk to read outside of the disk. The basic formula for `randlba` is:

$$random(maxlba - maxblk)$$

Here is an example script that does random reads:

```
for {set i 0} {$i < 50000} {incr i} { r10 [randlba] }
```

Chapter 4

Using The Serial Extension

4.1 Introduction

The serial extension provides a means for the CIL to communicate with a drive via the serial interface. This extension is meant to integrate Serial Debugger functionality into the CIL both for the sake of convenience and the ability to write scripts that make use of the serial interface and the drives standard interface (SCSI, FCAL, and SATA) together. Another advantage of the merger is that many added functions and bug fixes can benefit both the serial and standard interface. The TCL scripting language and true buffers also make it possible to create more powerful serial scripts.

The serial/serial3 drivers provide a means for the serial extension to communicate with the drive itself. The serial driver supports the UART2 protocol where the serial 3 driver supports the UART3 protocol. These drivers handle all of the behind-the-scene serial protocols. The two protocols are not compatible with each other.

4.1.1 Basic Architecture

The serial extension is based on a 2 level architecture. The lowest level is written in C++ as a CIL driver. This provides the advantage of having all of the CIL features provided for free and letting it deal with the protocol. The other layer is the TCL interface layer. The result is more powerful command line and scripting functionality. This gives users the ability to write scripts which are supported in both Windows and Linux.

4.1.2 Integratability

Another big advantage of the CIL's serial extension is that one can write scripts that take advantage of the serial extension and the drives more traditional interface (SCSI or FCAL) in the same script. This allows for more creative use of the serial interface in debugging applications.

4.1.3 Buffers

The Serial Debugger has no real concept of buffers. One capability that is provided was the ability to create a "virtual drive" from a "dump file". This file could then be acted on with regular serial debugger commands.

Because it is a part of the CIL, the serial extension has extensive support for buffers. As with other CIL commands, reading from the serial port automatically stores its results in the receive buffer. Writing to the serial port also comes from the send buffer, although there is an option for filling data within the `swrite` command (for convenience).

This support for buffers also introduces the other following changes:

- "Save to file" is no longer provided by the serial read function (`sread`). This functionality is provided by `buff save`.
- "Write from file" is also no longer provided by `swrite`. The `buff load` command is used instead.
- There is no longer a concept of a "virtual device". Equivalent functionality is provided by the `buff dump` command. Simply load the dump file into a buffer using the provided utility and use `buff dump` instead of `sread`. The reasoning behind this change is that it makes the function of `sread` and `swrite` less ambiguous (they ALWAYS read from the serial port now).

4.1.4 Numbers and Variables

In order to be consistent with the rest of TCL and the CIL, `sread` and `swrite` inputs are always assumed to be in decimal form, unless they are preceded by "0x".

4.1.5 CIL support

Because the serial driver is a part of the CIL, users benefit with support for all CIL features. These include buffers, device selection, feedback indicators, and sending CDBs all work as expected. This means that `device list` will work when the serial/serial3 UIL is selected. Another added feature is that the serial driver performs a "device scan" on start-up so it will recognize your drives, regardless of which serial port you have them plugged into.

One important thing to understand is that, because a person using the serial extension will probably want to use it with the drives traditional interface (SCSI, ATA, or FCAL) without having to switch UILs every time, the serial specific commands (such as `sread`) point to a UIL independently from the other CIL commands. What this boils down to is that you do not have to switch to the serial UIL for the `secho`, `sread`, `swrite`, etc. commands but you will need to for any other CIL commands that depend on the UIL being used (such as the `device` commands). You can also point the serial specific commands to other UILs using the `suil` command.

4.2 Connecting Niagara to a drive

In this section, we will show examples on how to connect Niagara to the drive using a serial debugger. To connect Niagara to the drive one can either use `serial::connect` which will put the drive in UART2 mode or `serial3::connect speed` which will put the drive in UART3 mode. See Code & UART Speed table on page **??** for valid values of speed. The `serial3::connect speed` command will also automatically set the Serial3 drivers speed based off of the given speed parameter.

4.3 Commands

In this section, we will look at the basic commands provided by the serial extension. These commands are supported in both UART2 and UART3. More on the differences later.

`secho`

This command simply asks the device to return it's vendor id and serial number. This command is used to confirm a connection with the drive.

`swrite <address> <length> [-dw|-dd] [data...]`

This command writes information from the send buffer to the drive. As a convenient shortcut, data can be specified in the command line. This data is then written to the send buffer before the command sends the send buffer to the drive. The `-dw` and `-dd` options specify the format that the command line data is in. These options only make sense to use when data is entered as part of the command. The data itself must be separated by spaces and must start with "0x" whenever hex data is used.

`sread <address> <length> [-dw|-dd]`

This command is used for reading memory in the drive. Note that, because this is a TCL command, you must precede all hex addresses with "0x". The `-dw` option displays the buffer content in 16-bit word (little endian) format. The `-dd` option displays the buffer contents in 32-bit (little endian) format. Note that the `-dw` and `-dd` options have no effect on how the contents are stored in the buffer. The buffer is always stored in byte format. The `-dw` and `-dd` options can also be used with the `buff dump` commands to view the buffer in the same format as returned by `sread`.

`sreadsp`

This command takes no arguments and returns the current value of the drive's stack pointer.

`suart_level`

This command returns the current UART version of the device over serial. This is done by sending a sync packet in UART2 and if that doesn't respond within a few milliseconds it will try it over UART3. If UART3 also times out then no devices are attached.

`suart2`

This command sets the serial communication link (using UART3 protocol) to the UART2 protocol.

`suart3 [speed]`

This command sets the serial communication link (using UART2 protocol) to the UART3 protocol. The `speed` value determine the link speed in UART3. See the table on UART3 Line Speed Coding.

Code	UART Speed
0x0000	115,200 bps (Legacy UART - Enterprise)
0x0001	78,800 bps
0x0002	57,600 bps
0x0003	38,400 bps
0x0004	28,800 bps
0x0005	19,200 bps
0x0006	14,400 bps
0x0007	9,600 bps
0x0008	1,843,200 pbs (Legacy UART - C&C)
0x0009	1,228,800 bps
0x000A	921,600 bps
0x000B	614,400 bps
0x000C	460,800 bps
0x000D	307,200 bps
0x000E	230,400 bps
0x000F	153,600 bps
0x0010	2.083... Mbps (16.6666Mbps/8)
0x0011	2.380... Mbps (16.6666Mbps/7)
0x0012	2.777... Mbps (16.6666Mbps/6)
0x0013	3.333... Mbps (16.6666Mbps/5)
0x0014	4.166... Mbps (16.6666Mbps/4)
0x0015	5.555... Mbps (16.6666Mbps/3)
0x0016	8.333... Mbps (16.6666Mbps/2)
0x0017	11.11... Mbps (16.6666Mbps/1.5)
0x0018	16.66... Mbps (16.6666Mbps/1)

`get_serial_list`

This command returns a list of available serial commands.

4.4 UART

4.4.1 UART 2

With the Indy/Monza line of hardware, a new UART protocol was introduced: UART2. UART 2 is backwards compatible with the original UART specification, so all of the existing CIL serial commands will continue to function correctly. In addition, UART 2 adds several new serial commands as well as support for sending CDB commands through the serial interface.

4.4.2 UART 2 CDB Support

With UART 2 comes the ability to send regular CDB commands to the drive through the serial interface. There are 2 steps that must be taken before a CDB can be sent through the serial interface.

First, it must be verified that the drive supports UART 2–this is done by issuing an sversion command. If the response is 2, then the CDB commands (as well as all other UART 2 functionality) will be enabled.

Second, the UIL index must be changed so that it points to the serial driver. If there is only one driver loaded, then the UIL index should already point to the correct driver. However, in most cases the serial index (set using the "suil" command) points to the serial driver while the UIL index (set using the "uil set index" or "usi" command)points to the main SCSI or F-CAL driver. All serial commands are sent to the serial index, but CDB's are sent to the uil index. So, the "uil set index" command must be changed to point to the serial driver before any serial CDB commands may be issued. The following screen shot demonstrates this:

```
(uil=0:test)(dev=0)> #check the list of UIL drivers loaded
(uil=0:test)(dev=0)> uil list
0: test:  v0.7.23: DeviceCount=20
1: spti:  v0.7.8: DeviceCount=1
2: serial:  v0.7.40: DeviceCount=1

(uil=0:test)(dev=0)> #check the serial index
(uil=0:test)(dev=0)> suil
2
(uil=0:test)(dev=0)> #check the UIL index
(uil=0:test)(dev=0)> uil get index
0
(uil=0:test)(dev=0)> #change the UIL index to point to the serial driver
(uil=0:test)(dev=0)> uil set index 2
UIL Set To: 2: serial:  v0.7.40: DeviceCount=1

(uil=2:serial)(dev=0)> #notice that the command prompt changes
(uil=2:serial)(dev=0)> #to now display "serial" as the current driver
```

Once the sversion command has been issued and the uil index has been changed to point to the serial driver, CDB commands may now be sent through the serial interface. To send a CDB, follow the same procedure as with the regular SCSI or F-CAL interface. The serial driver handles all the behind-the-scene serial protocol and returns the expected data. All of the same CDB parameters are available–just issue the CDB command with all desired parameters, and the CIL will construct the CDB command and pass it to the serial driver. Then, the serial driver will wrap the CDB in the correct serial frame and perform all of the querying and data xfer stages. Finally, the serial driver will receive the CDB response wrapped in a serial response block; the serial driver will strip the response block header information and return to the CIL only the CDB response data. If an error occurs during the communication, then the serial driver will attempt to correct the error, and upon giving up it will attempt to request status. If the CDB command returns an error, then the serial driver will request status and return it to the CIL.

For debugging purposes, it is not useful to have the serial driver perform all of the querying and data xfer phases of the serial CDB command. Because of this, the individual serial commands are available that send the CDB, query the drive, and transfer data. *Note, however, that the UART 2 protocol must be followed for these commands to be successful–the user is responsible for sending the squery commands and issuing the sxfer commands at the correct times!* Please see scdb (page 69), squery (page 69), sxfer (page 69), and sabortCDB (page 69) for more information.

4.4.3 UART 3

With the Formidible line of hardware, a new UART protocol was introduced: UART3. UART3 is not backwards compatible with the UART2 specification. However most of the UART2 commands have an equivalent UART3 command. This required that a new driver, the Serial3 Driver, be created to handle all of the behind-the-scene serial protocols, which was designed to act just like the original Serial Driver. Users can still send the same CDBs to the drive just as they did with the Serial Driver. The Serial Extension did require an update as well. It was modified so it will send the appropriate sread/swrite/etc commands to either the UART2 (serial) driver or UART3 (serial3) driver. The suil driver is the UIL that receives the serial command. This allows scripts that were written before the UART3 specification was created to still work without any changes to it.

The Formidible line also supports UART2. Because of this, we needed to create additional commands in order for the user to switch to and from UART3. Hence `suart2` and `suart3 speed` were created. See below for more information on these commands.

4.4.4 UART2 & UART3 Commands

The following commands remain the same in UART 3 as they were in UART 2.

sversion

This command takes no arguments, and it will return two bytes specifying the version of UART that the current drive supports. Note the return value doesn't mean that it supports UART2 or UART3 protocols.

Here is an example of what the the sversion command will look like an a UART 2 drive:

```
(uil=0:serial)(dev=0)> sversion

Success

OFFSET  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  0123456789abcdef
------  -----------------------------------------------  ----------------
000000  00 05                                            ..
```

sversion reference entry: page 350

sreset

The sreset command will initiate a reset on the drive. The drive will send a response block back indicating that the command is acknowledged and then call the appropriate reset function. There are two parameters, ?command flags? and ?drive type?. The drive type field is one byte long; however, currently only bit zero is used to specify whether or not the drive is an Indy/Monza drive. If the drive is an Indy/Monza drive, then ?drive type? should be set to 1. The ?command flags? parameter is a series of bits that specify what type of reset is to be performed.

The current Indy/Monza command flags are defined as:

```
1. Show Stop Flag (bit 0): If Ş1Ť then call Reset.ShowStop();
2. POR Reset Flag (bit 1): If Ş1Ť then initiate a Power On Reset
3. Hard Reset Flag (bit 2): If Ş1Ť then initiate a Hard Reset
4. Soft Reset Flag (bit 3): If Ş1Ť then initiate a Soft Reset
5. TmfShutdown Flag (bit 4): If Ş1Ť then initiate a Tmf Shutdown Reset
6. TmfSpindown Flag (bit 5): If Ş1Ť then initiate a Tmf Spindown Reset
```

sreset reference entry: page 348

scdb

The scdb command is one of the essential commands used for sending a CDB command to the drive through the UART interface. To be successful, this command *must* be used in conjunction with squery, sxfer, and possibly sabortCDB. The scdb command takes one parameter, ?length?, which specifies how many bytes to use from the send buffer. Then a serial CDB command block is constructed, using the first ?length? bytes from the send buffer as the CDB command to send. *Note: the CDB length is checked to verify it is 6, 10, 12, or 16 bytes long; however, the actual bytes are not checked to verify that they form a valid CDB.* If there are no errors during the communication, then the drive will return a busy response block, signifying that the drive received the command successfully and is busy processing it. After sending a successful scdb command, the next phase should be the query phase. scdb reference entry: page 345

sabortCDB

The sabortCDB command is one of the essential commands used for sending a CDB command to the drive through the UART interface. In most cases, this command will be used in conjunction with scdb, squery, and sxfer. The sabortCDB command tells the drive to abort any current UART CDB processing and return the UART CDB handling to an initial state. This command takes no parameters, and it will return the entire header and response block information received from the drive. This command is used when a CDB has been sent, but the entire process of querying and data transfer cannot be completed. In this case, an sabortCDB command should be issued to clear the state of the UART, and then the entire CDB phase must be started from the beginning. This command may also be used as a last resort in effort to clear the state when an initial scdb command returns an error. sabortCDB reference entry: page 345

4.4.5 UART2 Only Commands

sdelay

The sdelay command will tell the drive to change the delay between data receive and data send. The default is for the drive to wait 250 micro seconds after receiving data before sending response data. The one required parameter is an integer that specifies the new delay in micro seconds. On some older systems, 250 micro seconds may not be enough time for the computer to be ready to receive response data–a communication sync problem will occur. In this case, increasing the delay may correct this problem. sdelay reference entry: page 346

slip

The slip command will (on a Fibre Channel drive only) cause the drive to request LIP. There is one parameter, ?LIP type?, is a one byte value that will specify the type of LIP; it may be 0xF7 or 0xF8. The UART device will return status acknowledging that it received the slip command, and then it will perform the appropriate tasks to request LIP. slip reference entry: page 347

squery

The squery command is one of the essential commands used for sending a CDB command to the drive through the UART interface. To be successful, this command *must* be used in conjunction with scdb, sxfer, and possibly sabortCDB. The squery command takes no parameters; when issued, it will send a query frame to the drive, asking for status on whether it is ready or not. This command is meant to be used to poll the drive after sending a CDB frame and between data xfer stages. The squery will return 8 bytes of data (the entire busy, ready, or fault response block). After issuing a successful scdb command, the squery command should be issued until a ready frame is received. Also, after the data xfer phase is complete, the squery command should be issued until a ready response block is received before a status data xfer command is sent. squery reference entry: page 347

sxfer

The sxfer command is one of the essential commands used for sending a CDB command to the drive through the UART interface. To be successful, this command *must* be used in conjunction with scdb, squery, and possibly sabortCDB. The sxfer command will send a data xfer command block to the drive. There are three basic types of

data xfer commands that may be sent: inbound data, outbound data, and command status. There are 4 required parameters that specify what type of data xfer to send. The ?Transfer Bit? parameter should be a 0 or 1 that specifies which direction the data is going to be transferred (0 for inbound, 1 for outbound). The ?Status Bit? parameter should be a 0 or 1 that specifies whether or not this data xfer frame should be used for transferring status. If the status bit is set, then this command will request the return status of the CDB that was sent; this should only be issued when the inbound/outbound data xfer stage is complete, or when the drive, after a query command, returns a fault response block instead of a ready or busy response block. The last two parameters of sxfer are ?Send Length? and ?Recv Length?. The send length will only be used if the command is an inbound data transfer to the drive, and the receive length will only be used if the command is an outbound data transfer from the drive. Since the actual bytes received during a data xfer command is read in on the header of the data xfer response block, the ?Recv Length? parameter will be overridden. As long as the ?Recv Length? is greater than zero, the correct amount of data will be read in and returned. sxfer reference entry: page 351

suart3

This command is new in conjunction with the UART3 Interface Specification. Note: This command is *only supported* if the driver returns a value of 5 or higher for the UART Version command. This command requires a coded speed value for the new speed for the interface. See Code & UART Speed Table for valid values. The `suart3 speed` command causes the drive to switch to the UART3 protocol specification for this interface. The drive will send a response block to the host in the UART2 protocol and current speed indicating the command is acknowledged. If the command was valid the change to the new protocol and speed will occur immediately after the response is sent.

suart3 reference entry: page 350

4.4.6 UART3 Only Commands

sstatus

This command is the Get Drive State request which can be used by the host to check on the status of a long running SCSI command (such as a manufacturing command). The drive will respond to the request with the ack Data Available and will return a Get Drive State response followed by the state data.

```
Byte:   Description:
0-1     Drive state version (0x0001)
2-3     Operating State (Equivalent to the Operating State field of Inq page 3.)
4-5     Functional Mode (Equivalent to the Functional Mode field of Inq page 3.)
6-7     Degraded Reason (Equivalent to the Degraded Reason field of Inq page 3.)
8-9     Broken Reason (Equivalent to the Borken Reason field of Inq page 3.)
10-11   Reset Cause (Value definition is beyond the scope of this document.)
12-13   Showstop state:
        0x0000 - No Showstop has occurred.
        0x0001 - Showstop has occurred and drive is 'stopped'.
        0x0002 - Showstop has occurred and the drive is attempting to reset itself.
14-15   Showstop reason (Value definition is beyond the scope of this document.)
16-19   Showstop value (Value definition is beyond the scope of this document.)
```

sstatus reference entry: page 349

suart2

The `suart2` command requests that the UART Interface gets set back to the UART2 protocol. If a valid request is received the drive will respond with the ack Ready and will immediately switch to UART2 mode at the default speed.

suart2 reference entry: page 349

4.4.7 Additional Helper Serial Commands

The following commands were created to help the user save time by automating steps for them.

suart_level

This command is used to get the current UART protocol. It does so by issuing an `secho` in both UART2 and UART3 protocols. Which ever one responds then that is what is returned. For example, if a drive is using the UART2 protocol then this command will return 2.

suart_level reference entry: page **??**

srescan

This command performs a device rescan on the suil driver.

srescan reference entry: page 348

sspeed

This command gets/sets the speed of the suil driver. If no argument is given then the current speed is returned. If one is given, it must be in bits per second (bps).

sspeed reference entry: page 349

4.5 Advanced UART Commands

This section will focus on specific advanced functionality of the serial driver, such as Indy/Monza simulator support, manual port selection, and low level byte reading/writing.

4.5.1 Driver Parameters

Manual port selection

After the serial driver is loaded, the first serial command sent will cause a bus re-scan in an attempt to detect all connected serial devices. This means that an echo command block will be sent to each available port, and a device will be created for each port that responds. However, during drive debugging, this automatic scan is not desirable. Using the "port" parameter while loading the UIL serial driver allows the auto scan to be disabled. The format of this parameter is as follows:

```
uil create serial port ?PortName?
```

The correct port-name (COM1, COM2, etc) should be substituted for ?PortName?. This parameter will cause the CIL to assume that a drive is connected to that port, and it will create the device without actually scanning the bus. The advantage of this is that no information will cross the bus until the user manually sends a command.

4.5.2 Supported UIL Messages

The "UIL message" command (page 337) allows specific commands to be defined for each driver. The serial driver has several messages that it recognizes.

get_delay and set_delay

The "get_delay" message takes no parameters; it returns the current delay (in ms) that the driver waits between packets. The "set_delay" message takes on parameter, an integer. It changes the post packet delay to the new value specified in ms.

get_retry_count and set_retry_count

The "get_retry_count" message takes no parameters; it returns the current number of times a serial command block will be retried after a failed attempt. The default value is 5. Every command will be attempted at least once, and then the retry count specifies how many more times it will be tried. The "set_retry_count" takes one integer parameter that changes the retry count.

get_poll_count and set_poll_count

The "get_poll_count" message takes no parameters; it returns the current number of times a drive will be polled during a CDB send before giving up. If the drive returns busy after "poll count" polls, then the driver will stop polling and return the error of the last query command. The default poll count is 5. The "set_poll_count" message takes one integer parameter that changes the poll count.

get_stats and reset_stats

The serial driver has built in error logging. A count will be kept of bytes sent, bytes received, commands sent, protocol errors, and timeout errors. Also, every response byte will be logged in a table with a count of how many times it has been seen. These counters are all set to zero when the driver is loaded as well as every time the `uil message "reset_stats"` command is called. The `uil message "get_stats"` command will print out report of the current logging statistics.

4.5.3 sio

For debugging situations, it may be useful to have direct control over what bytes are sent over the serial interface. The sio provides a way for doing this; this command is very low level, sending and receiving only the number of bytes specified. The sio command takes 2 arguments; ?send length? and ?recv length?. When issued, the driver will open the serial port, raise the RTS line, and send ?send length? number of bytes from the send buffer. Then, the RTS line will be dropped and the driver will attempt to read in ?recv length? number of bytes into the receive buffer. If all zeros are returned, then most likely the driver was not able to read any data. After both the send and receive stages are complete, the serial port is closed and the data (if any) is returned to the CIL. It is very difficult, but possible, to send a valid serial command using the sio command. Each packet must be individually sent two bytes at a time with a 1 byte receive length, then the response byte must be checked for errors. *Because of this, it is recommended that sio be used only as a debugging tool to verify that serial communication exists.*

4.6 Tips and Tricks

The UART protocol is based upon timing; if the drive and CIL are not in sync, then communication will not occur. Therefore, accurate timing is imperative! Because timing within the serial driver is implemented in software, accuracy is difficult to achieve. Every time the serial driver is loaded a calibration process is called; this process will attempt to choose correct timing parameters based on the speed of the machine it is running on. *To achieve an accurate calibration, it is recommended that the CPU load be at a minimum while loading the serial driver!*

Another implication of a software based timing scheme is variability. Just because the calibration is accurate doesn't mean that the timing length will be exactly the same every command. In some cases, one or more bytes may be lost if the timing loop runs longer than it should; this is especially common when the CPU load is at a maximum during serial command processing! *Because of this, it is recommended that the CPU load be at a minimum while executing serial commands; especially lengthy procedures, such as a big memory dump!* However, the UART protocol does have error recovery built in and the serial driver has a retry count for failed commands, so usually small timing errors are corrected before they are returned to the user.

If the UART communication seems unreliable, there are a few things that can be done. First issue a "get_stats" (page 72) message to the serial driver to display the error logging statistics. If there are a lot of timeout errors or protocol errors, then a sync problem may exist. Also, if there are large counts of ack bytes other than 0x80 (the ack byte for good status), then this may also indicate problems. Here are some possible remedies to sync problems:

- The "sdelay" command (page 69) may be used to change the delay that the drive uses between the read and write phase.
- The "set_delay" message (page 71) may be used to change the time that the serial driver waits between sending packets.

- The “device set timeout” command (page 254) may be used to change the amount of time, in milliseconds, that the serial driver will listen before returning a timeout.

Chapter 5

Brief Introduction To TCL

To this point we have been concentrating on the Niagara extensions to TCL. In this section, we will look at the basics of TCL syntax. Although we will be touching on the basics of TCL in this chapter, there is simply not enough room in this users guide to go into depth on TCL's features. If you want more information the official TCL manual can be found at http://www.tcl.tk/man/tcl8.5/. In this chapter I will use the > character to represent command input. Lines without the > character represents the text that will be returned by TCL when the command above is typed. Now we will try a basic TCL command, available in all versions of TCL:

```
> puts "hello world"
hello world
```

The `puts` command is TCL's version of "print to screen". Another useful command is the `expr` command. This command is used to perform math calculations:

```
> expr 5 + 5
10
```

5.1 TCL Variables

All TCL variables start with a $. To set a variable use the following command:

```
> set varname value
value
```

Note that the `set` command does not expect (or want) the $ in front of the variable name you are defining. There are two ways to check the value of a variable, `set` and `puts`. The following two commands have the same effect:

```
> set varname
value
> puts $varname
value
```

Notice how you need to include the $ operator for the `puts` command but not the `set` one. Once a variable is defined, it can be used in place of a command argument, for example:

```
> set lba 100
100
> set tl 2
2
> r10 $lba $tl
```

is equivalent to `r10 100 2`.

5.2 Table Of TCL Commands

Below is a list of commands included in TCL. Note that this list does not contain any of the CIL specific commands. See section 3.4 on page 38 for CIL specific commands. The list also does not include any of the Tk extension commands. Here is the list:

Command Name	Description

5.3 TCL Syntax

The execution model that TCL uses can be somewhat strange for beginning programmers. This is because TCL works as a string parser, which is a little different from other languages. Here we give some rules on how TCL parses a command that is typed.

1. Commands are broken into tokens. Each token separates other tokens by one or more spaces or tabs.
2. The first token of a command is the command type. All other tokens on the same line are considered arguments to this command type. Ending a line with `{` or \ tells the interpreter that the command continues on the next line.
3. Enclosing characters in quotes `("")` tells the interpreter to treat everything between the quotes as a single token (spaces and tabs are a part of the token). However, variables (words that start with $) and bracket expressions are still substituted.
4. Enclosing characters in braces `({})` also tells the interpreter to treat everything between the braces as a single token (spaces and tabs are a part of the token). Variables and bracket expressions are not substituted.
5. Enclosing characters in brackets `([])` causes the interpreter to evaluate the characters in between the brackets as a separate command, then insert the return value where the brackets used to be.

Now for some examples to clarify the rules given above. Let us start with the program lines:

```
> set animal dog
> set number 3
```

In the example

```
> puts dog
dog
```

The command type is `puts`. The command has one argument, dog. Here is another example:

```
> puts the dog is 3
wrong # args: should be "puts ?-nonewline? ?channelId? string"
```

The command type is `puts`. The command has 4 arguments: the, dog, is and 3. This is too many arguments for puts (you will get an error). Here is another example that uses quotes to fix the problem:

```
> puts "the dog is 3"
the dog is 3
```

Here we used quotes to represent “the dog is 3” as a single token. Here is an example with variables:

```
> puts "the $animal is $number"
the dog is 3
```

Because we used quotes, variables are expanded. If we do not want variable substitution, we use {}, as shown here:

```
> puts {the $animal is $number}
the $animal is $number
```

Note that in many TCL commands (such as `if`, `for` and `proc`) using {} is a useful way to pass segments of code. For example:

```
> set i hello
> for {set i 0} {$i < 10} {incr i} "puts $i"
<prints hello 10 times>
```

Here we used "" for our command block, this causes the i variable to be substituted with hello before the `for` command ever sees it. Switching to braces:

```
> set i hello
> for {set i 0} {$i < 10} {incr i} {puts $i}
<prints 0 through 9>
```

Now the setting of `$i` is defferred to when the for command calls it. This is probably the effect we wanted. Notice that, unlike C, we are not *forced* to use a { after the `for` command, it just makes more sense do to so most of the time. Lastly we look at the bracket notation:

```
> expr 2 + 3
5
> puts "8 * 7 = [expr 8 * 7]"
8 * 7 = 56
```

Here the interpreter calls the command `expr 8 * 7` as a separate command then places the result in the string.

5.4 Running TCL Commands From A File

To run TCL commands in a file, use the `source` command. For example:

```
source test.tcl
```

The `source` command executes the commands it finds in the specified file immediately. The `source` command can either be used to quickly execute a series of commands or to define procedures.

5.5 Multiple Statements Per Line

You can declare multiple statements per line by placing a semicolon `(;)` between commands. As an example:

```
> set i 0
> while {$i < 5} {
    puts $i
    incr i
  }
```

can be stated on one line using a semicolon:

```
> set i 0; while {$i < 5} { puts $i; incr i }
```

5.6 Comments

A comment in TCL is declared by a # sign. Note that the pound sign must begin a line. You can get around this, however, by putting a semicolon directly before the pound sign. Here are some examples:

```
> # This is a comment
> set i 0 # This is not allowed
> set i 0 ;# A semicolon is a workaround, however
```

5.7 Control Flow Commands

TCL's control flow commands enable you to loop commands repeatedly or process commands conditionally. Control flow commands work exactly like other TCL commands in that the first token is treated as the command type and other tokens are treated as arguments to the command. This means that all of the commands in a `for` loop are treated as a single argument! It also means that, if you are extending a conditional statement to multiple lines, the first line must contain an open brace. Putting the brace on the next line confuses TCL into thinking that a new command is about to start. Here are some examples:

```
if {$a > 10} {

  #everything from the brace above
  puts $a
  #to the brace below
  #is treated as a single argument to the if command

}

if {$a > 10}
{
  #putting the open brace on
  #the next line  is not allowed
  #because tcl thinks that
  #you started a new command
}
```

5.7.1 The if Command

The format for the `if` command is as follows:

```
if <condition> <commands>
```

Here is an example:

```
if {$lba > $maxlba} { set lba 0 }
```

You can also use the `else` and `elseif` tokens to create a conditional chain. Here is an example:

```
if {$age < 13} {
  set type child
} elseif {$age < 20} {
  set type teen
} else {
  set type adult
}
```

5.7.2 The for Command

The `for` command is a convienent way to set up a loop. The basic syntax is:

```
for <initial condition> <ending condition> <loop iterator> <statement block>
```

Here is an example that performs 10 block sequential reads:

```
for {set i 0} {$i < 1000000} {incr i 10} {
  read10 $i 10
}
```

5.7.3 The while Command

The `while` command is similar to the `for` command. The difference is that the `while` command excludes the initial condition and the loop iterator. The syntax is:

```
while <ending condition> <statement block>
```

The `while` command will iterate until the ending condition is met. Here is an example which emulates the `for` example above:

```
set i 0
while {$i < 1000000} {
  read10 $i 10
  incr i 10
}
```

5.7.4 The foreach Command

The `foreach` command is useful for iterating through a list of items. The syntax is:

```
foreach <variable name> <list> <statement block>
```

Here is an example:

```
foreach i {1 2 3 4 5} {
  puts $i
}
```

Here is another example that uses the output of `get_cdb_list` to print out documentation for all available CDB commands:

```
set cmd_list [lsort [get_cdb_list]]
foreach cmd $cmd_list {
  puts [eval "$cmd -help"]
}
```

5.7.5 The switch Command

The `switch` command is used to select between a list of choices. The basic syntax is:

```
switch <options> -- <variable> <list>

where <list> contains:

<<keyword1> <action1> <keyword2> <action2> ...>
```

Here is an example:

```
switch -exact -- $name {
  rover {set type dog}
  cuddles {set type cat}
  poly {set type bird}
  default {set type unknown}
}
```

5.8 Defining Procedures (Functions)

Like most programming languages, TCL allows you to define functions that can later be called by name. In TCL these are called procedures. Here we define a simple procedure:

```
proc add {a b} {
  set c [expr $a + $b]
  return $c
}
```

No we can use `add` as if it were built into TCL:

```
> add 2 3
5
```

Note that variables declared inside a `proc` statement are considered to be *local*. This means that if we are using the variable `c` outside the add prodecure above, it will not be overwritten. For example:

```
> set c 111
111
> add 3 4
7
> puts $c
111
```

In some cases we want to have a variable change inside a procedure effect a global variable. To accomplish this we identify the variable with the `global` statement. Here is the procedure above rewritten with the `global` statement used:

```
> proc add {a b} {
>   global c
>   set c [expr $a + $b]
>   return $c
> }
```

```
> set c 111
111
> add 3 4
7
> puts $c
7
```

It is also possible to have a variable number of arguments passed to a procedure. See a book on TCL for a more detailed description of `proc`.

5.9 Arrays

In TCL, an array is a variable with a string index in parathenses. Below are some examples of legal arrays:

```
set index(5) 26
set address(zip_code) 55906
set address(street_name) "Maple St"
```

5.10 Lists

TCL also provides support for lists. To declare a list, use the `list` command:

```
> list nathan matt jason luke
nathan matt jason luke
```

Here are various operations that can be performed on a list:

```
> set names [list nathan matt jason luke]
nathan matt jason luke
> lsort $names
jason luke matt nathan
> lappend names ben
nathan matt jason luke ben
> linsert $names 2 chad
nathan matt chad jason luke ben
```

There are also other operations that can be performed on lists such as `lindex`, `llength`, `lrange`, `lsearch`, `lreplace`, and others. The website www.tcl.tk is a great resource for more information on lists.

5.11 String Manipulation

Being a string based language, TCL has a number of provisions for manipulaing string expressions. Some examples include join, split, append, string, reqexp and regsub. A full description of string functions can be found in a dedicated TCL book. Below I give some simple examples that illustrate some of the possibilities:

```
> set str hello
hello
> append str " " there
hello there
> join $str ","
hello,there
> string match "*there" $str
1
```

```
> string match "there" $str
0
> string length $str
11
> string toupper $str
HELLO THERE
> string trim "      spaces      "
spaces
```

5.12 File Operations

File operations in TCL are straight forward. Use the `open` command to get a file handle on a file. The use `puts`, `gets`, `read`, and `write` (and more...). To access the file. Here is an example that numbers the lines in a file and prints them to the screen.

```
proc dump_lines {filename} {
  set line_num 1
  set fd [open "$filename" r]

  while { [gets $fd line] != -1 } {
    puts "$line_num ) $line"
    incr line_num
  }

  close $fd
}
```

Here we extend the function above to write it's output to a file:

```
proc dump_copy_line {infile outfile} {
  set line_num 1
  set in [open "$infile" r]
  set out [open "$outfile" w]


  while { [gets $in line] != -1 } {
    puts $out "$line_num ) $line"
    incr line_num
  }

  close $in
  close $out
}
```

If you want to read binary data, use the `read` command. The format for this command is `read <file id> <length>`. An example of it's use is:

```
set buffer [read $fd 2048]
```

5.13 Introduction To The TK gui extension

Like TCL itself, the TK extensions to TCL are too numerous to fully explain here. Again www.tcl.tk is a great resource for a full explanation. Below I give a simple example of how to create a window in TCL and attach an "Inquiry" and "Test Unit Ready" button to it. First we will create the window. The command for this is:

```
> toplevel .mywindow
```

Where `.mywindow` is the name of the window we are creating. This command will bring up a new empty window. Now to add a couple of buttons to it:

```
> button .mywindow.inq -text "Inquiry" -command {puts [inq]}
.mywindow.inq
> button .mywindow.tur -text "Test Unit Ready" -command {puts [tur]}
.mywindow.tur
> pack .mywindow.inq .mywindow.tur
```

At this point you will have two new buttons in your window. Clicking on them performs the commands specified in the `-command` argument of each button. Like I said, this is just the tip of the iceburg in what you can do in TK. A dedicated book on TCL/TK will go into more detail.

Chapter 6
Error Handling Techniques And Variables

6.1 Introduction

Niagara has several special variables that are set and referred to during device error situations. These variables allow program code to quickly discern an error type and properties. Alternatively, you can use TCL's built in `catch` command as a method of trapping just about any error condition. This section discusses both methods of error handling.

6.1.1 General and CDB Errors

The basic difference between general and CDB errors is as follows:

- Device errors relate only to CDB commands. Examples include `CHECK_CONDITION` and `TIMEOUT`. And example of error types that do not apply are syntax errors, out of memory, and file not found.
- General errors include all errors, including all listed in the previous bullet.

To handle errors, the following techniques are used:

- Device errors are handled with the `$ec`, `$sns`, `$err` and `$cdberr` variables.
- Generic errors can be handled with the `catch` statement.

Why Two Error Types?

There are three reasons we distinguish CDB errors from general errors:

- CDB errors can be tedious to deal with using the `catch` statement
- Because `catch` also catches generic errors, it will also trigger on syntax errors and other non drive related errors. Discerning these two types of errors when using `catch` is the responsibility of the user.
- There are times when a lot of drive errors are expected (such as in drive testing). Special CDB error handling allows these situations to be handled in a cleaner fashion.

6.2 Global Variables

The global variables described in this section are related to CDB errors only. They are invalid in any other error context. Note that in order to use these variables within a procedure, you need to declare them `global`. For example:

```
proc test {} {
  global ec
  global sns

  if {[catch {inq 1}]} {
    puts "ec = $ec"
    puts "sense data = $sns"
  }
}
```

Why Are These Variables Global?

These variables are global for two reasons:

- If a test program ends, these variables can be used interactively to get more information. If these variables were local they would be lost as soon as the procedure exited.
- Many of these variables are mapped into TCL in a way that makes access very fast. This technique requires the variables be global.

6.2.1 ec

This variable returns a CIL error code. See appendix B on page 117 for a list of possible codes. The `$ec` variable can be used to easily check if an error is of a certain type (such as a timeout or check-condition).

6.2.2 err

This variable is set to the string value of the last error that occurred. It is most useful when used with `$cdberr`. Note that `$err` is only overwritten when an error occurs. This means that you should not check `$err` to see *if* an error occurred on the last command, use `$ec` for that.

6.2.3 sns

This variable contains a list of the sense data that was returned by the last command. Note that, for performance reasons, this data is only valid updates when `$ec` = -16 (check condition). Always check `$ec` before trusting the contents of this variable.

To access the individual bytes of the sense data use `lindex`. For example:

```
inq 1
set byte2 [lindex $sns 2]
puts "byte 2 of the sense data is $byte2"
```

6.3 The Local Variable: cdberr

Currently, the only local variable is `$cdberr`. This variable can be used to assign custom error handling code to an cdb error event. The way the variable works is that, if the variable is set, a CDB error will execute the contents of the variable (instead of the default operation of returning an error). We will start with a simple example to show how this works:

```
proc test1 {} {
  puts [inq 4]
  puts [inq 5]
}

test1
```

In this example the first command will fail and return an error. The second command will never be executed. The results from this script will look something like:

```
CMD: 12 00 04 01 00 00


<E> Check Condition At :

70 00 05 00 00 00 00 18 00 00 00 00 24 00 00 C0
```

```
00 02 00 00 F8 23 00 00 00 00 00 00 00 00 00 00

Translation: Illegal Request.  Invalid Field in CDB.
```

Now we will override this default functionality with our own cdb error handling. If we modify the function as shown:

```
proc test2 {} {

  global ec

  set cdberr {
    puts "The function had an error code of $ec"
  }

  puts [inq 4]
  puts [inq 5]

}

test2
```

The output will now be:

```
The function had an error code of -16

The function had an error code of -16
```

In the example above, we override the default operation of the cdb error handling with new functionality by setting `cdberr`. Note that in the example above *no errors are generated*. This is why the second inquiry in the code was executed. Alternatively we may want an error to be generated but reported in an alternate way. We simply change the script to generate an error in `$cdberr`:

```
proc test3 {} {

  global ec

  set cdberr {
    error "The function had an error code of $ec"
  }

  puts [inq 4]
  puts [inq 5]

}

test3
```

6.3.1 Restoring default behavior

To restore default error handling, unset `$cdberr` as follows:

```
unset cdberr
```

6.3.2 Why is cdberr local?

There are two reasons that `$cdberr` is local:

- To protect procedures from changing command line behavior
- To protect procedures from effecting each others error handling behavior

By keeping `$cdberr` local, a script can assume that it is in a certain error state upon startup. The command line error state will also remain unchanged (unless you explicitly set `$cdberr` at the global level).

6.3.3 Some Examples

To return the errors as normal with some extra information, we use the `$err` variable:

```
set cdberr { error "$err \n The function had a return code of $ec" }
```

When defining this in a procedure, don't forget to make `$err` (and in the above case `$ec`) global. If we simply want to print errors, we use `puts` above in place of `error`. We can also log errors to a file like this:

```
set cdberr {
  set logfile [open "logfile.txt" "a"]
  puts $logfile "--------------------------------"
  puts $logfile "At [clock format [clock seconds]]:\n"
  puts $logfile $err
  flush $logfile
  close $logfile
  error $err
}
```

Again remember to make `$err` global when defining `$cdberr` in a procedure and we can use `puts` in place of `error` to log cdb errors without generating TCL errors.

6.4 Using catch

An alternative to using `$cdberr` above is the `catch` statement. Catch is a more general handling method that has the following syntax:

```
catch { commands } var
```

When `catch` is run as above, it will execute "commands" and put the result in `var`. The return for `catch` is 0 if the command completed without error and 1 if an error occurred. Using this information, we can set up an example:

```
set errflag [catch { inq 5 } var]
if {$errflag} {
  puts "An error occurred:"
  puts $var
} else {
  puts "The command completed successfully:"
  puts $var
}
```

You can also use `$err`, `$ec`, and `$sns` as they are used in previous examples:

```
proc test {} {
  global ec

  set errflag [catch { inq 5 } var]
  if {$errflag} {
    puts "An error occurred:"
    puts "ec = $ec"
  } else {
    puts "The command completed successfully:"
    puts $var
  }
}

test
```

6.4.1 Choosing Between cdberr and catch

In cases where you are testing for errors not related to sending CDB's, `catch` is your only option. When dealing with CDB commands however either can be used. Much of this choice depends on which approach fits best with your programming style. There are also cases where it makes sense to combine `catch` and `$cdberr`. For example, if you want errors to be reported in a special way, except in certain cases, such as the following code:

```
proc test {} {

  global err
  global ec

  #We would like to see the error code as well
  set cdberr {
    error "$err \n ec = $ec"
  }

  # supported by generic cdberr above
  inq

  #if we fail here, do additional processing
  if {[catch {r10 [randlba]} var]} {
     #additional error processing here...
  }

}
```

This code handles errors in a special way through `$cdberr` but also uses `catch` to do additional processing when the read CDB fails. Note that when used in this context, `$cdberr` must return an error of the `catch` statement will never be flagged.

Chapter 7
Random Number Generation

7.1 Introduction

To compliment TCL's built in `expr rand()` function, Niagara includes a suite of random commands. These commands extend the functionality of TCL's random number generation capability, adding the following features:

- Creation of up to 1024 independent random number generation channels (`rand open`, `rand close`)
- Each channel can be seeded independently (`rand seed`)
- Each channel can contain optional histogram constraints (`rand addhist`, `rand showhist`)
- Each channel can produce integers or integer ranges (`rand int`, `rand range`)
- Each channel can produce floats or float ranges (`rand float`, `rand frange`)

These features allow for more robust control over what types of random numbers are generated and improves the reproducibility of these numbers (by offering independently seeded random number channels).

7.2 Basic Use

This section explains basic random commands that can be used without being concerned with random channels or histograms. The most basic commands are:

```
rand int
rand float
rand range <min> <max>
rand frange <min> <max>
```

The `rand int` command returns an integer between `0` and `0xFFFFFFFF`. The `rand float` command returns a floating point number: $0.0 \leq n < 1.0$. The `rand range` command returns an integer: $min \leq n \leq max$. The `rand frange` command returns a floating point number: $min \leq n < max$. Here are some examples of the commands in use:

```
rand int
rand float
rand range 0 255
rand frange 0.0 6.28319
```

To reseed the random number generator, use the `rand seed` command. This command takes an integer values as an input. Here are some examples:

```
rand seed 1234
rand seed [clock seconds]
```

7.3 Using Channels

Channels allow you to use several independent random channels at the same time. Here is an example that uses a lot of random numbers:

```
rand seed 12345

for {set i 0} {$i < 5} {incr i} {
    set len [rand range 1 64]
    bfr send 0 [expr $len * 512]
    set lba [randlba $len]
    puts "writing $len blocks at lba $lba"
    #just a test
    #w10 $lba $len
}
```

This script will produce the following output:

```
writing 23 blocks at lba 119994285
writing 1 blocks at lba 136511712
writing 2 blocks at lba 39966460
writing 59 blocks at lba 104881024
writing 32 blocks at lba 27510689
```

This script will run fine and will produce identical numbers each time it is run (because we set the seed). This script, however, has a subtle problem. The problem is that all of the random numbers are highly interdependent. For example say you make a subtle change to the blocksize (using 520 instead of 512):

```
rand seed 12345

for {set i 0} {$i < 5} {incr i} {
    set len [rand range 1 64]
    bfr send 0 [expr $len * 520]
    set lba [randlba $len]
    puts "writing $len blocks at lba $lba"
    #just a test
    #w10 $lba $len
}
```

You now get these results:

```
writing 23 blocks at lba 51031861
writing 38 blocks at lba 40581075
writing 49 blocks at lba 143141557
writing 5 blocks at lba 2564612
writing 36 blocks at lba 137440182
```

Completely different numbers! The problem is that `rand range`, `bfr`, and `randlba` are all sharing the same random number generator. Changing the number of random bytes need by `bfr` from 512 to 520, effected the other two commands. These situations are one area where random channels can help. Here is the first script (512 version), modified to use multiple channels:

```
set rlen [rand open 12345]
set rbuff [rand open 23456]
set rlba [rand open 34567]

for {set i 0} {$i < 5} {incr i} {
    set len [rand range 1 64 $rlen]
    bfr send 0 [expr $len * 512] $rbuff
    set lba [randlba $len $rlba]
    puts "writing $len blocks at lba $lba"
    #just a test
    #w10 $lba $len
}

rand close $rlen
rand close $rbuff
rand close $rlba
```

The result of running this script is:

```
writing 23 blocks at lba 74485584
writing 13 blocks at lba 60777058
writing 48 blocks at lba 36466626
writing 60 blocks at lba 24134585
writing 28 blocks at lba 130652177
```

Now we change the blocksize to 520:

```
set rlen [rand open 12345]
set rbuff [rand open 23456]
set rlba [rand open 34567]

for {set i 0} {$i < 5} {incr i} {
    set len [rand range 1 64 $rlen]
    bfr send 0 [expr $len * 520] $rbuff
    set lba [randlba $len $rlba]
    puts "writing $len blocks at lba $lba"
    #just a test
    #w10 $lba $len
}

rand close $rlen
rand close $rbuff
rand close $rlba
```

The result is **still**:

```
writing 23 blocks at lba 74485584
writing 13 blocks at lba 60777058
writing 48 blocks at lba 36466626
writing 60 blocks at lba 24134585
writing 28 blocks at lba 130652177
```

In our new script, we used a separate random channel for our lba, transfer length, and buffer data. Because we are using three separate channels, we can made modifications to an area (such as the block size), without effecting the other two streams. Another advantage to channels is that we can apply histogram constraints to the values produced. Histograms are explained in the next section. To open a new random channel use the command `rand open`. The format for this command is:

```
rand open ?seed?
```

The `?seed?` value is optional. If you do not specify a seed, the current time is used as a seed. Another good source for a seed is the `rand int` command, which will retrieve a random number from channel 0. Opening a new channel returns a channel id. You generally want to save this id so that you can later use it to refer to the channel:

```
set id [rand open]
```

Channel zero is a special random channel in that it is always open (you can not close it), and you can not apply histogram information to it. The `rand open` command will never return a channel id of zero (because it is already open, always). Any random commands that do not have a specified random channel are using channel 0. To close a channel, use the `rand close` command:

```
rand close ?channel?
```

It is a good idea to close random channels when you are finished with them. Niagara can have up to 1024 random channels open at a time.

To use a random channel, you simply refer to the channel id in the appropriate command's argument list. The following commands support channel ids:

```
buff fill rand <index> <offset> <length> ?channel?
rand float ?channel?
rand frange <min> <max> ?channel?
rand int ?channel?
rand range <min> <max> ?channel?
rand seed <seedval> ?channel?
```

Histogram related random commands *require* a channel id. These commands are described in the next section.

7.4 Using Histograms

Histograms allow you to fine tune the type of random numbers a particular random channel generates[1]. When you add histogram information to a channel, you are specifying the likelihood that the value returned falls within a specified range. Examples of this will follow. Here is an example list of some of the things that can be done using histograms:

- Fill a buffer with 10% zeros and 90% 0xFF.
- Test the inner cylinders of a drive 60 percent of the time, the middle cylinders 30 percent of the time and the outer cylinders 10 percent of the time.
- Ensure that a particular transfer length will be used 10% of the time
- Simulate different "3 sigma" ranges during off-track read-write tests

[1] You cannot add a histogram to channel 0

To add histogram data to a channel, use the `rand addhist` command. The format for this command is:

```
rand addhist <channel> <min> <max> <percent>
```

The effect of this command is to tell the channel that you want to see random values between `min` and `max` "`percent`" of the time. When you call `rand addhist` multiple times on the same channel, the results are stacked together, creating a histogram profile. You can stack up to 32 histogram entires per channel. In this example, we are filling a buffer with 0x00 50% of the time, 0x55 25% of the time and 0xAA 25% of the time:

```
set r [rand open]
rand addhist $r 0x00 0x00 50
rand addhist $r 0x55 0x55 25
rand addhist $r 0xAA 0xAA 25

bfr send 0 512 $r
```

In this example, we set a random channel to return numbers that are either 0-2 or 8-10 (but never 3-7):

```
set r [rand open]
rand addhist $r 0 2 50
rand addhist $r 8 10 50

for {set i 0} {$i < 20} {incr i} {
  puts -nonewline "[rand int $r] "
}
puts ""

rand close $r
```

To look at the current histogram status of a channel, you can use the `rand showhist` command. Here is the output for our buffer example above:

```
 > set r [rand open]
 > rand addhist $r 0x00 0x00 50
 > rand addhist $r 0x55 0x55 25
 > rand addhist $r 0xAA 0xAA 25
 > rand showhist $r

   Entry #        Min        Max    Percent
-----------------------------------------
         1   0.000000   0.000000  50.000000
         2  85.000000  85.000000  25.000000
         3 170.000000 170.000000  25.000000
```

What happens if we do not use up 100% of our potential histogram space? Basically the remaining percentage is returned as a default value. Here is the example, modified above:

```
 > set r [rand open]
 > rand addhist $r 0x00 0x00 50
```

```
 > rand addhist $r 0x55 0x55 25
 > rand showhist $r

   Entry #        Min        Max    Percent
-----------------------------------------
         1   0.000000   0.000000  50.000000
         2  85.000000  85.000000  25.000000
         3    Default    Default  25.000000
```

Note that "Default" means different things to different commands. For example, the default range for the `rand int` command is 0-0xFFFFFFFF where the default range for the `rand float` command is 0.0-1.0.

If we define more than 100% for our histogram profile, the less likely entries we defined will never get called. The `rand showhist` command will show this. Here is an example where we use 175% of our histogram space:

```
 > set r [rand open]
 > rand addhist $r 0 1 50
 > rand addhist $r 1 2 30
 > rand addhist $r 2 3 75
 > rand addhist $r 3 4 20
 > rand showhist $r

   Entry #        Min        Max    Percent
-----------------------------------------
         1   2.000000   3.000000  75.000000
         2   0.000000   1.000000  25.000000
         3   1.000000   2.000000   0.000000
         4   3.000000   4.000000   0.000000
```

Here is a breakdown of the results:

- Our 75% entry is the most likely. Niagara puts the most likely histogram entries at the top of the list (for best performance).
- Our next likely entry is our 50% entry. However our first entry has already taken 75% of our histogram space, this leaves only 25% of the space left.
- Now our histogram space is maxed out at 100%, our remaining two entries will never be called as a result.

The real moral of this story is that you should avoid defining more than 100% of your histogram space anyway. Defining more than 100% does not make sense.

Chapter 8
Command Queueing

8.1 Introduction

Using certain drivers, Niagara offers support for command tag queuing. The queuing features of Niagara are as follows:

- The ability to send a series of commands concurrently to a device, without waiting for status from each command.
- The ability to send a sequence of commands to a device in rapid succession. Certain drivers (i-Tech) supports this functionality in hardware while others (Linux) emulate this functionality using a tightly optimized C loop (i.e. without the TCL overhead)
- The ability to specify a certain tag or have tags auto-generated for you
- The ability to wait for any tag to complete, to wait for a specific tag to complete or to wait for all tags to complete.
- Support for simple, ordered or head of queue tagging
- Ability to mix different tag types in the same queue
- The ability to retrieve status and data for each command returned by the device from a list

These functions are implemented through the following commands:

Command Name	Description
`qmode concurrent`	Puts Niagara in concurrent queuing mode. In this mode commands are sent immediately but status on the commands is deferred
`qmode disable`	Puts Niagara in normal (non queued) mode
`qmode stacked`	Puts Niagara in stacked queuing mode. In this mode commands are built up in a table and then executed with a `qctl send` command.
`qctl get auto_incr`	Returns 1 if auto tag increment is enabled, 0 otherwise
`qctl get tag_type`	Returns the current tag queuing mode
`qctl get num_waiting`	Returns the number of commands (if any) that the drive has returned status on
`qctl get max_depth`	Returns the current maximum queue depth
`qctl idx_info <index> ?-list?`	Returns status information for the specified queue index. Call this command after `qctl recv tag` or `qctl recv all` to get more information about command completion status
`qctl recv`	Waits for and retrieves the next available command from the queue, status is given like a normal command
`qctl recv tag <tag_id>`	Waits for and retrieves the commands from the device until tag `<tag_id>` is received. Returns success (and command data for `tag_id`) if all commands up to and including `<tag_id>` were completed successfully or an error if any of the commands did not
`qctl recv all`	Waits for and retrieves all outstanding commands from the device. Returns success if all commands were completed successfully or an error if any of the commands did not.
`qctl send`	Sends a command table built in memory. Only applicable in `qmode stacked` mode.
`qctl set auto_incr <1/0>`	Used to enable or disable automatic tag increments
`qctl set next_tag <val>`	Set the tag id for the next command to be sent
`qctl set tag_type <type>`	Sets a particular queuing mode. Options for `<type>` are `simple`, `ordered`, and `head`
`qctl set max_depth`	Sets the current maximum queue depth
`qctl table_info`	Returns a formatted table indicating status of all received commands. Call this command after `qctl recv tag` or `qctl recv all` to get more information about command completion status.
`qctl tag_info <index> ?-list?`	Returns status information for the specified queue tag. Call this command after `qctl recv tag` or `qctl recv all` to get more information about command completion status.

Note that some or all of these features may not be available, depending on your driver. For example, the SPTI driver does not support queuing at all while the Linux driver doesn't support queuing types other than simple.

8.2 General Usage Stacked Mode

To start we will issue 16 random queued reads to the drive from the command line, we will then look at the returns using various techniques. Stacked queuing mode allows us to build up a list of commands in an internal table and send them to the drive all at once. With some drivers, this mode is even hardware assisted. A benefit of

stacked mode is that it allows us to effectively achieve queue depths interactively from the command line, whereas this would not be possible in concurrent mode. To begin our look at command queuing we will put Niagara in stacked queuing mode:

```
qmode stacked
```

We'll also say that we would like Niagara to automatically increment queue tags for us with an initial tag id of zero:

```
qctl set auto_incr true
qctl set next_tag 0
```

Building A Sequence

Now we will issue a command:

```
r10 [randlba]
```

Note that instead of returning status of the read operation, the `r10` command above returns a number (0). This number is the queue tag id that the command will have when it is sent to the drive. Because we are in stacked mode, the command has not been sent to the drive yet, but placed in an internal table. Next we will add 15 more random reads to the table:

```
do 15 {r10 [randlba]}
```

Receiving Commands

Above we added 15 additional read operations to our internal table, each with an incrementing queue tag. Including our initial `r10` command makes our total table size 16. The table will be sent to the drive when a receive command is called:

```
qctl recv
```

The above receive command returns the first command that the drive completed. Note that, in simple queuing mode, the commands may come back in a different order than they were sent. To get the rest of the commands, we'll use a `do` loop:

```
do 15 {puts [qctl recv]}
```

Other Receive Options

Alternate ways to receive commands are through the `qctl recv all` and `qctl recv tag` operations. The `qctl recv all` operation retrieves all outstanding commands from the device. Here is an example:

```
do 16 {r10 [randlba]}
qctl send
qctl recv all
```

One question that naturally comes up here is "What about the return status for all of these commands". It is often useful to know if all of the commands were executed successfully. The first clue is the return of `qctl recv all`. If this command returns without error, all commands were completed successfully. If one or more commands did not complete successfully, an error is returned. In either case, we can get more information about the commands returned by using the `qctl *info` family of commands. As an example, we will issue 4 commands to the device with on of them (the inquiry) a purposeful error. Here is the setup:

```
qmode set next_tag 0
inq 1
r10 [randlba]
r10 [randlba]
r10 [randlba]
qctl send
qctl recv all
```

In the case of `qctl recv all`, the command should return an error. This is because we did not call inquiry correctly. To see status on all the commands, we use the `qctl table_info` command:

```
> qctl table_info

#1
-------------------------------------------------------
Tag:   0000
CDB:   12 00 00 FF 01 00 .. .. .. .. .. .. .. .. .. ..
EC:    -16 (Check Condition)
Sense: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
       00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

#2
-------------------------------------------------------
Tag:   0002
CDB:   28 00 12 34 56 78 00 00 01 00 .. .. .. .. .. ..
EC:    0 (Success)

#3
-------------------------------------------------------
Tag:   0001
CDB:   28 00 05 7A 52 12 00 00 01 00 .. .. .. .. .. ..
EC:    0 (Success)

#4
-------------------------------------------------------
Tag:   0003
CDB:   28 00 02 92 01 FD 00 00 01 00 .. .. .. .. .. ..
EC:    0 (Success)
```

We can also look at a particular index entry:

```
> qctl idx_info 1

#1
-------------------------------------------------------
Tag:   0000
CDB:   12 00 00 FF 01 00 .. .. .. .. .. .. .. .. .. ..
EC:    -16 (Check Condition)
Sense: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
       00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
> qctl idx_info 1 -list

{1 0 -16 {12 00 00 ff 01 00} {00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00}}

> qctl idx_info 3 -list

{3 1 0 {28 00 05 7A 52 12 00 00 01 00} {}}
```

Or by a particular tag value:

```
> qctl tag_info 1 -list

{3 1 0 {28 00 05 7A 52 12 00 00 01 00} {}}

> qctl tag_info 3

#4
------------------------------------------------------------
Tag:   0003
CDB:   28 00 02 92 01 FD 00 00 01 00 .. .. .. .. .. ..
EC:    0 (Success)
```

Note the order of the lists given above is:

```
{<index> <tag id> <error code> <cdb> <sense data>}
```

To look at a particular field, use the `lindex` command to extract the contents. The `-list` option is useful in this situation. For example:

```
> lindex [qctl idx_info 1 -list] 2

-16

> lindex [qctl idx_info 3 -list] 3

{28 00 05 7A 52 12 00 00 01 00}
```

8.3 Capturing Data

As commands are executed, their data is sent from the current send buffer to the current receive buffer. In this situation you can only see the data from the last command received (because the previous command's data are overwritten). To avoid this situation, specify different buffers for each command as you send them. Note that you might need to use the `buff set count` command to allow for a greater number of buffers. Here is an example that does 32 random reads and reads status of each command into a separate buffer:

```
proc read_32rand {

  qmode seqential
```

```
  #allocate 32 buffers
  if {[buff get count] < 32} {
    buff set count 32
  }

  #set up 32 reads
  for {set i 0} {$i < 32} {incr i} {
    r10 [randlba] -ri $i
  }

  #send the commands out and get status back
  qctl send
  qctl recv all

  #return status and the first data bytes (if command was a success)
  #for each command
  for {set i 0} {$i < 32} {incr i} {
    puts "[qctl idx_info $i]\n"
    if {[lindex [qctl idx_info $i -list] 2] == 0} {
      #successful command, print first 64 bytes of data
      puts "[bd $i 0 64]\n"
    }
  }

}
```

8.4 Concurrent Mode

Although stacked mode is useful for achieving a certain queue depth, it does so in a "one shot" type operation. For cases where we would like to try and maintain a queue depth for an extended period of time, concurrent mode is a better choice. Concurrent mode differs from stacked mode in that it does not wait for you to issue a `qctl send` command before sending commands to the device. Instead commands are sent as soon as you execute them.

This difference allows you to send a command without building a table and without issuing a `qctl send` call, both of which take a bit of overhead to execute. However, it is now your responsibility (if you want to get any kind of a queue depth) to ensure that you get the next command sent to the device as rapidly as possible. Note that all queuing commands except `qctl send` can be used in concurrent mode.

Below is a procedure that will perform random reads from a device to a specified queue depth:

```
proc rand_read_qd {ops depth} {

  #if there is an error, turn off queueing
  global err
  set cdberr {
    qmode disable
    error $err
  }
```

```
  #set queueing parameters
  qmode concurrent
  qctl set max_depth $depth
  qctl set tag_type simple
  qctl set auto_incr 1

  #load up the queue with commands
  for {set i 0} {$i < $depth} {incr i} {
    r10 [randlba]
  }

  #maintain depth for ops operations
  for {set i 0} {$i < $ops} {incr i} {
    #pullor cases a command off
    qctl recv
    #put another on
    r10 [randlba]
  }

  #get the remaining commands
  qtcl recv all

  #turn off queueing
  qmode disable

}
```

Chapter 9

Using Hardware Data Generation / Compare

9.1 Introduction

Certain testing systems (e.g. iTech) offer the capability to generate, collect and compare data in hardware. Consequently, these systems have historically also provided an inefficient path for doing the same in software. Because of this, Niagara offers the ability to access the hardware data generation / comparison features built into cards that support the feature. Currently, only the iTech card offers this support.

This chapter provides an overview of different hardware modes supported by Niagara and discusses the performance / flexibility implications of using each feature.

9.2 Understanding iTech Performance

The iTech architecture provides on card memory buffers for data transfers. This provides a number of advantages, including hardware data generation and comparisons and buffer overrun protection. The buffer overrun protection prevents the computer from crashing when the drive sends more data to the host than is expected. The hardware generation and comparison features are provided because it is (significantly) faster to use the iTech memory directly as opposed to copying the memory from the computer to the iTech card and then executing a command. For some reason transferring data from main memory to the iTech card is quite slow, making hardware generation necessary in many applications. Note that with non-iTech cards that DMA from main memory, this is not an issue and good performance can be achieved without using any hardware generation features.

The default mode of iTech is the (slow) transfer from main memory to card memory (and visa versa) for each CDB execution. This has the effect of making the iTech card perform more slowly than card types. When performance is necessary and iTech is the card that is being used, the `device set xfer_mode` and `device set read_xfer` commands can be used to access the hardware generation capabilities.

9.3 Changing The Transfer Mode

The command that facilitates hardware generation is `device set xfer_mode`. This command has the following format:

```
device set xfer_mode <mode>
```

Available modes are:

Mode	Description
`normal`	Default mode: buffer data is copied from main memory on both read and write. Both reads and write operate in the slower mode.
`hc`	Hardware Compare: The iTech card's read and write buffers are compared on read, does not speed up writes.
`random`	The iTech card randomly generates data for write commands. Speeds up writes.
`random_hc`	The iTech card randomly generates data for write commands and expects this same data during read commands. Speeds up writes. In most cases you should use `random_seed` or `random_seed_keyed` instead of this mode because they handle compares more robustly (i.e. `random_seed_keyed` will correctly compare any block that has been written, where `random_hc` is much more restrictive and will generally only compare a sequence correctly if it is the same as the sequence that was written)
`random_seed`	Works the same as `random_hc` except that the number generator is seeded with LBA and with block information. This allows large transfers (that disconnect and reconnect) to be compared properly.
`random_seed_keyed`	Works the same as `random_seed` with the LBA coded at the beginning of each block
`keyed`	Uses current buffer contents but overlays LBA keys, does not speed up writes
`keyed_hc`	Uses current buffer contents but overlays LBA keys, performs hardware compare on read, does not speed up writes
`inc`	Writes an incrementing pattern to the drive. Speeds up writes
`inc_hc`	Writes an incrementing pattern to the drive. Performs hardware compare on reads. Speeds up writes
`repeat`	The first write CDB executed after this command will copy data from the PC's memory to the iTech card (slow). All subsequent write calls will then reuse this same data (no buffer xfer) for improved performance.
`repeat_hc`	The same as `repeat` with hardware compare on read.
`repeat_read_hc`	The fist read CDB executed after this command will copy data from iTech's read buffer to its write buffer. Subsequent reads will compare the new read buffer with the write buffer.

Note that for **all** of these modes, the data is still copied from the card to main memory on reads (another time consuming process). To disable this feature, you can use the `device set read_xfer` command to suppress this for a performance gain (see the next section for details).

I'll say it again because its important: **Reads will run slow regardless of transfer mode unless you use** `device set read_xfer 0`

Mode Effect

The hardware generation modes only are used with the following CDBs:

- read6
- read10

- write6
- write10

For all other CDBs, “normal” mode is used.

9.4 Suppressing Card to Memory Transfers

None of the transfer modes above will speed up read operations on an iTech card. The reason is because, by default, read data is copied from the iTech card to main memory so that you can see the data that was sent. You can suppress this behavior using `device set read_xfer 0`. In this mode, reads will be fast at a cost of seeing the data. At this point, you can use iTech hardware compare features if you need to check data integrity. If a miscompare occurs, the iTech driver will transfer data into main memory, regardless of the `read_xfer` setting. This way you can examine data on a miscompare.

9.5 Returning to a Default State

There are two ways to return to a default state, the first is to call `device set xfer_mode normal` and `device set read_xfer 1` as appropriate. Another is to call `init`. The `init` function will return Niagara to a default state for queuing (off), hardware generation (off) and feedback (default).

9.6 Example

Below are some examples of hardware compare. Note the use of checking for cross driver compatibility. This code will run correctly whether hardware compare is available or not.If Hardware compare is not available, the code below uses software compare instead.

```
proc h_compare {count} {

    #do $count writes followed by $count reads
    #use hardware compare if possible, software otherwise

    #create some random channels for data generation
    set lba_seed [rand int]
    puts "lba_seed: $lba_seed"

    set lba_rand [rand open $lba_seed]
    set data_rand [rand open]

    #**************************************
    #try to set hardware mode
    #**************************************

    if {[catch { device set xfer_mode random_seed_keyed }]} {
        #software mode
        puts "***Hardware Mode Not Available, Using Software Mode***"
        set smode 1
    } else {
        #hardware mode
        puts "*** Using Hardware Compare ***"
        set smode 0
        #our compare will handle the reads
        device set read_xfer 0
```

```
    }

    #**************************************
    #do 1000 writes
    #**************************************

    puts "Doing $count Random writes"
    for {set i 0} {$i < $count} {incr i} {
        set lba [randlba 1 $lba_rand]
        if {$smode} { rand seed $lba $data_rand; bfr 0 0 512 $data_rand }
        w10 $lba
    }

    #reset our seeds
    rand seed $lba_seed $lba_rand

    #**************************************
    #do 1000 read/compares
    #**************************************

    puts "Doing $count Random Read/Compares"
    for {set i 0} {$i < $count} {incr i} {
        set lba [randlba 1 $lba_rand]
        if {$smode} { rand seed $lba $data_rand; bfr 0 0 512 $data_rand }
        r10 $lba
        if {$smode && [buff compare 0 1 512]} {
            error "Data Miscompare"
        }
    }


    #**************************************
    #clean up
    #**************************************

    init
    rand close $lba_rand
    rand close $data_rand

    return "Done"

}
```

Chapter 10
Startup Scripts

Startup scripts are tcl scripts that are executed when Niagara is initially started. These scripts are located in `startup`, `startup_ata`, `startup_hdc`, `startup_scsi`, and `startup_user` directories where Niagara is installed. You can see which scripts are executed by watching Niagara's console window on startup. To include your own scripts, simply add them to the `startup_user` directory.

10.1 Included Startup Scripts

This section contains an overview of the included startup scripts. It is recommended that you also take a look at the actual code in the `.tcl` files to understand exactly what each script is doing...

10.1.1 checksum

The `checksum` script adds many usefull checksum utilities.

```
checksum::calc_And_Fill <buff> <offset> ?length?
```

The `calc_And_Fill` procedure is used to update the checksum. It takes a buffer, offset and an optional length.

10.1.2 debug_puts

The script `debug_puts.tcl` allows for the use of debug puts. Setting the debug level in a namespace using the `set_Debug_Level` will determine the number of messages that will appear. The command `dputs` works like a normal puts except that you also pass in a debug level that determines when the message should be printed.

```
dputs <dlevel> <message>
set_Debug_Level <debug_level> ?namespace_name?
get_Debug_Level ?namespace_name?
```

Here are some examples of `dputs`.

```
dputs 1 -color blue "The code is at this point"
dputs 3 "The code has failed at line 273"
```

10.1.3 do

The `do.tcl` script adds a single command to TCL called `do`. The `do` command provides an easy way to repeat a command a certain number of times (or forever). An additional bonus to using `do` is that a Tk button appears that allows you to cancel a command sequence before it is finished. The general syntax is:

```
do <iterations> <command>
```

The `<iterations>` parameter can be a number or `fo` for "forever". Here are some examples of `do`:

```
do 5 {puts "hello"}
do 1000 {rd10 [randlba]}
do fo {rd10 [randlba]}
```

10.1.4 device_ops

The device_ops script contains the procedure reset_Device_Info which performs a rdcap for SCSI drives or identify_device for ATA drives, which resets the device info to the correct max LBA or blocksize.

```
reset_Device_Info
```

The procedure reset_Device_Info will return zero if successful and one if not.

10.1.5 drive

The drive script implements drive::connect for consistency with serial::connect, agilent::connect, etc...

```
drive::connect
```

10.1.6 endian

The endian.tcl script adds a command to TCL called endian_switch. The endian_switch command can be used to switch a value between big endian and little endian format. The syntax is:

```
endian_switch <value> <size>
endian <value> <size>
```

The <value> parameter specifies the value you want to be switched. The <size> parameter determines the size in bytes of the <value>. Acceptable values for the <size> argument are 2, 3, and 4. This command can also be run with endian instead of endian_switch.

10.1.7 file

The file script adds the procedure convert_Filename. The procedure convert_Filename will convert a given file path into a path that TCL can recognize. Backslashes are converted to forward slashes. Spaces are replaced with " ". The syntax is:

```
file::convert_Filename <filename> <filter_spaces>
```

Where filename is the file path to convert and filter_spaces can be set to zero if you don't want spaces to be filtered.

10.1.8 hdc

The hdc script is an attempt to consolidate all HDC identification logic to a single place in the code, simplifying the process of updating the tools when we get new HDCs.

```
hdc::generation
```

The hdc::generation procedure will return the HDC's chip ID.

10.1.9 identify

The identify.tcl script adds a command to TCL called identify. The identify command calls a series of inquiry commands in an attempt to identify the drive (Only HGST drives are identified, other return "unknown"). This script is useful for applications which need to be sensitive to which drive they are testing. The syntax for the identify script is:

```
identify ?uil_index? ?device_index?
```

The parameters ?uil_index? and ?device_index? are optional and can be used to specify a specific device. If these parameters are not specified, the current device is targeted.

10.1.10 modefields_cli

The `modefields_cli` script sets up mode page definitions for various devices. The variables set up in this file are used by modefields_cli functions and the Mode Fields GUI tool.

10.1.11 model_number

The `model_number` script contains procedures that can be used to obtain model numbers from drives. The syntax is.

```
model::get_Model_Number ?serial_number? ?project_id? ?alsm_code_filename?
```

All of these parameters are optional but `serial_number` is the default serial number to use if necessary, `project_id` is the project of the selected device, and `alsm_code_filename` will set the allsym file name that's just been created.

10.1.12 serial

Run `serial::connect` to point SUIL to the serial driver. Run `serial::connect 3` to point SUIL to the serial3 driver, so that the UART3 interface can be used.

```
serial::connect
```

10.1.13 sns_tools

The `sns_tools` script contains procedures to allow a user to easily parse data received from the request_sense command

```
get_Code ?sns_data?
```

The `get_Code` procedure returns a hex byte for the code data.

```
get_Descriptor_Data <sns_data> <descriptor>
```

The `get_Descriptor_Data` procedure searches the sns_data for the given descriptor and returns all data within that descriptor block except for the descriptor byte and the additional size byte.

```
get_Information_Bytes ?sns_data?
```

The `get_Information_Bytes` procedure returns an LBA in hex of either 4 bytes (fixed) or 8 bytes (descriptor) depending on the format of the sns_data.

```
get_Key ?sns_data?
get_Progress_Indication ?sns_data?
```

The `get_Progress_Indication` procedure returns in hex the 2 byte data for the progress indication. Note progress indication data is only valid if the sense key is of value NO_SENSE or NOT_READY.

```
get_Qual ?sns_data?
get_Sense_Format
```

The `get_Sense_Format` procedure will return the current sense format (Descriptor [1] or Fixed [0]).

```
get_Sector ?sns_data?
get_Sksv ?sns_data?
get_Uec ?sns_data?
get_Valid ?sns_data?
set_Sense_Format <d_sense>
set_Data <data>
```

10.1.14 uartmode

The `uartmode` script adds UART mode manipulation functionality to Niagara.

`uartmode`

The procedure `uartmode` can be used to set the UART mode by providing an argument or to get the current UART mode by not providing any arguments.

Chapter 11
Expanding The TCL GUI

The scripting power of **TCL/TK** makes it *very* easy to be able to dynamically expand your code. Several parts of the Niagara take advantage of this flexibility and allow you to customize various parts of the GUI. This chapter will explain what you can customize, how the system works in general, and examples of all the changeable parts. Things that can be altered include:

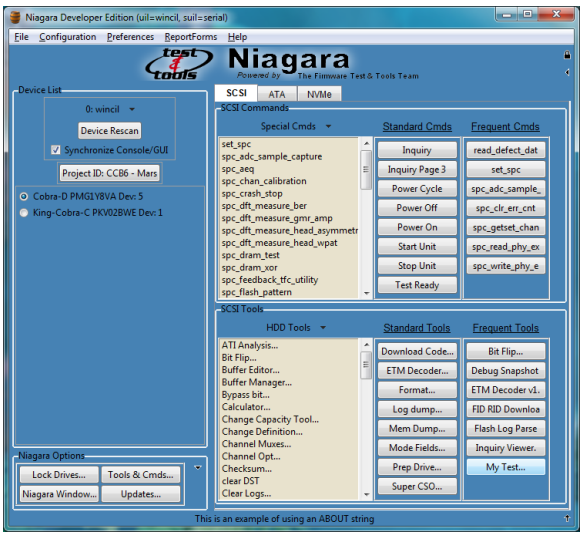
1. Quick Buttons can be added or taken away from the Quick button list.
2. Action types can be added to the action menu
3. New items can be added to the action listbox
4. Personal preference variables can be added.
5. Some default personal preferences may be changed.

11.1 Working with Quick Buttons

Quick buttons are meant as an easy way to access scripts that you use often. These scripts are located within the TkGui/buttons directory of your main Niagara path. (example: c:/Niagara/TkGui_user/buttons for most Windows machines. Buttons may be added or removed by adding them to or removing them from this directory. On the GUI startup for Niagara the buttons directory is examined and any tcl file in this directory is added to the GUI as a button. The file name becomes the button name minus the `.tcl` extension. Anywhere in the name where ppp occurs is replaced by ... and _ are replace by spaces. For example `MyTest.tcl` would show up as the button MyTest, `My_Test_ppp.tcl` would appear as `My Test ...`.

You will notice however that simply placing any TCL script in the buttons directory may create the button, but clicking on the button won't always execute the script correctly. This is because when the button is pressed a procedure is called based upon your filename. The extension .tcl is removed and replaced by `_init`. So when you click on `My Test ...` the procedure `My_Test_ppp_init` is called. Be careful when naming procedures and global variables in your custom scripts as name pollution can occur. If two functions have the same name the most current one sourced in is called which could create undesired behavior. A safe way to name procedures is to prefix it with the name of the file minus the extension. So instead of naming a procedure go, a safer name would be `My_Test_ppp_go`.

You may have noticed on the main GUI that when the mouse is placed over some quick buttons a short description about what the script does occurs in the bottom most information label field. (See figure 11.2 on page 111) You can add this functionality to your script quickly and easily, all you need to do is at the beginning of your script (outside any functions) create a global variable with your filename (minus the extension) appended with `_ABOUT`, and set this to what you want to display. (See figure 11.1)



Code for this button:

```
global My_Test_ppp_ABOUT
set My_Test_ppp_ABOUT "This is an example of using an ABOUT string"
proc My_Test_ppp_init {} {}
```

Figure 11.1: My Test Example

11.2 The Action List

The action list allows you to easily organize different kinds of tests under broad categories. These categories appear in the action button list menu (See figure 11.3). When a category is chosen all of the scripts under this category are placed in the listbox under the action list menu button. To create a new category make a directory inside the actions directory. When the TCL GUI first starts up the Actions directory will be scanned for other directories to be added to the action list menu button. In turn these directories are scanned for TCL files which are placed in the listbox once their category is chosen. If there are no TCL files in a directory then the listbox will appear blank. Naming conventions follow the format described in the section titled **Working with Quick Buttons**. One advantage to using the action list is that a large number of tests can be put under a category which will be displayed in the scrolling listbox. If these tests were made into Quick Buttons the buttons would appear small and unreadable as there would be a large number competing for limited space.

11.3 Preference Variables

Niagara allows you to store custom variables when you close the program and bring them up when you start the program, thanks to the way TCL handles arrays and the way it uses global variables. All of the personal preference variables are stored in prefs.tcl in the main Niagara directory. On startup, this script is run which sets the preference variables to the value they had when the program exited.

Preferences are stored in a global array so any script may have access to the preferences. You can define new prefs variables just by setting them. Make sure that the procedure in which you create a new preference has prefs globalized (global prefs) otherwise a local version will be created then destroyed once the function terminates. If you want to find out if a prefs index is already taken you can use the command `array names prefs` this will list all indices that are currently defined.

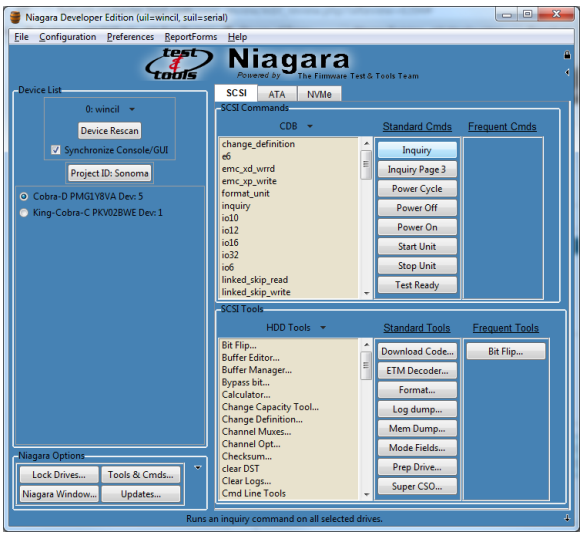


Figure 11.2: Mouse over the Inquiry button

11.3.1 Special Global Variables

The variable niagara_selected_devices has all the devices under the current UIL that has been chosen. The command [uil get index] can provide the current UIL index. Using the dev and uil flags on most commands you can run a script against a bunch of chosen drives. The code example below shows a function that runs an inquiry on page 0 of every selected drive and prints the results to the screen.

```
proc runInquiry {} {
    global niagara_selected_devices

    foreach index $niagara_selected_devices {
        puts [inq -dev $index -uil [uil get index]]
    }
}
```

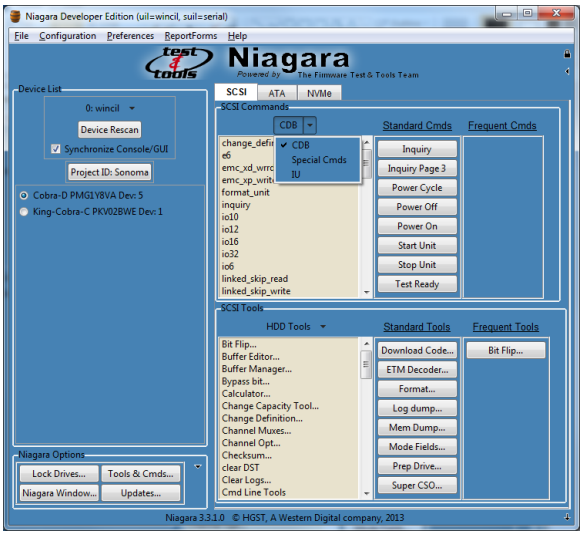


Figure 11.3: Display of action list button menu

Appendix A
TCL Code Examples

In This appendix we will look at several example TCL scripts, starting with simple scripts and extending into more complex ones.

A.1 Random Read/Write/Verify Application

In This section we start with a simple sequential read application and build it until we have a random read application with a tk GUI front end.

A.1.1 Basic Sequential Read Loop

We start with a simple sequential loop.

```
#This Script Performs 10000 Sequential Reads
for {set i 0} {$i < 10000} {incr i} {
    r10 $i 1
}
```

A.1.2 Basic Random Read Loop

Here we use the `randlba` command to read from a random lba instead of sequentially

```
#This Script Performs 10000 Random Reads
for {set i 0} {$i < 10000} {incr i} {
    r10 [randlba] 1
}
```

A.1.3 Creating A Procedure

Next We encapsulate our function into a procedure, we make the number of iterations and block size parameters. Note the `numblocks` parameter used in the `randlba` function. This prevents the `randlba` command from returning an lba that overlaps the maximum lba (due to the numblocks).

```
proc rand_read {iterations numblocks} {
  #This Script Performs Random Reads
  for {set i 0} {$i < $iterations} {incr i} {
    r10 [randlba $numblocks] $numblocks
  }
}
```

A.1.4 Adding LBA Range and Boosting Performance

Next we will add an `lbarange` parameter. If this paraameter is zero, the maximum lba range will be used. We will also add status to the script to help show its progress.

```
proc rand_read {iterations numblocks lbarange} {

  #boost performance
  feedback push; feedback min
```

```
  #figure out the lba range
  if {$lbarange == 0} {
    set lbarange [device info maxlba]
  }

  #trim off the block size
  set lbarange [expr [device info maxlba] - $lbarange + $blocksize]

  #Start the reads
  set i 0
  while {$i < $iterations} {
    #print progress every 1000 iterations
    for {set j 0} {$j < 1000 && $i < $iterations} {incr j} {
      r10 [randlba $lbarange] $numblocks
    }

    puts "$i Reads Completed"
  }

  #restore feedback
  feedback pop
}
%$
```

A.1.5 Adding Writes And Compare

Next we will add a write to the device and a data compare.

```
proc rand_read_write {iterations numblocks lbarange} {

  #boost performance
  feedback push; feedback min

  #make sure our send and receive buffer are setup right
  buff set si 0
  buff set ri 1

  #count miscompares
  set miscompares 0

  #figure out the lba range
  if {$lbarange == 0} {
    set lbarange [device info maxlba]
  }

  #compare size
  set compare_size [expr $numblocks * [device info blocksize]]

  #trim off the number of blocks
  set lbarange [expr [device info maxlba] - $lbarange + $numblocks]
```

```
  #Start the write, read, compare
  set i 0
  while {$i < $iterations} {
    #print progress every 1000 iterations
    for {set j 0} {$j < 1000 && $i < $iterations} {incr j} {
      set $lba [randlba]

      #fill in some data
      buff fill rand 0 0 $compare_size

      w10 $lba $numblocks
      r10 $lba $numblocks

      if {[buff compare 0 1 $compare_size]} {
        puts "***Miscompare At LBA $lba at [clock format [clock seconds]]***"
        incr miscompares
      }
    }

    puts "$i Reads Completed"
  }

  puts "There Were $miscompares miscompares during this test"

  #restore feedback
  feedback pop
}
```

A.1.6 Adding A TK GUI Front End

A.2 Reading Random Blocks From Every Drive On The Loop

This script shows how to switch devices and read randomly from them. Note how the `randlba` command is smart to the maximum lba of the current device.

```
proc rand_read_all {$iterations} {

  #get the number of devices
  set maxdev [device get count]

  for {set i 0} {$i < $iterations} {incr i} {

    #choose a random device
    device set index [expr int(rand() * $maxdev]

    #read a random lba
    r10 [randlba] 1

    #another way to do status updates
    if {[expr i%1000] == 0} {
```

```
      puts "$i Reads Completed"
    }
  }

}
```

Appendix B

EC Error Codes

Error Code	C Symbol	Description
-1	ERR_UNSUPPORTED_DRIVER	Driver is not yet supported
-2	ERR_BAD_DEVICE_KEY	Key for the device is wrong
-3	ERR_UNSUPPORTED_FEATURE	The feature for that particular driver is unsupported
-4	ERR_BUFFER_OVERRUN	Buffer has been over loaded with too much information
-5	ERR_BUFFER_UNDERRUN	Buffer not filled
-6	ERR_BAD_DEVICE_INDEX	Not a valid device index
-7	ERR_DEVICE_INACTIVE	Device Inactive
-8	ERR_SEND_FORMAT	Send command format error
-9	ERR_SEND	General send() failure
-10	ERR_INTERNAL	Problems internal to the code
-11	ERR_ARG_COUNT	Bad Argument Count
-12	ERR_PARAMETER_SET	Parameter Set Failure
-13	ERR_PARAMETER_GET	Parameter Get Failure
-14	ERR_TIMEOUT	Timeout Error
-15	ERR_UNDEFINED	Undefined error
-16	ERR_CHECK_CONDITION	Receive Error
-17	ERR_MISCOMPARE	Hardware Miscompare
-18	ERR_MULTIPLE	Multiple errors (ITech)
-19	ERR_QUEUE_EMPTY	Queue empty
-20	ERR_DEVICE_LOCKED	Device Locked
-21	ERR_RESERVATION_CONFLICT	Another initiator owns the drive that this initiator is talking to
-22	ERR_CONDITION_MET	Condition Met
-23	ERR_QUEUE_FULL	Drive Queue is full
-24	ERR_RESOURCE_BUSY	Resource Is Busy
-25	ERR_COMMAND_TERMINATED	Command Was Terminated
-26	ERR_BAD_TARGET	No Target Response
-27	ERR_ABORTED	Command Aborted
-28	ERR_PARITY	Parity Error
-29	ERR_DRIVER_INTERNAL	Internal UIL Driver error
-30	ERR_RESET	Bus/Loop reset
-31	ERR_INTERRUPT	Interrupt error
-32	ERR_SOFT_ERROR	Soft Error
-33	ERR_MEDIA	Media Error
-34	ERR_HARD_ERROR	Hard Error
-35	ERR_DRIVER_SENSE	Could Not Retrieve Sense Information

continued on next page

continued from previous page

Error Code	C Symbol	Description
-36	ERR_MEMORY_ALLOC	Error Allocating Memory
-37	ERR_NO_SUCH_TAG	Queue ID tag not found
-38	ERR_PROTOCOL	Communication exists, but it does not protocol
-39	ERR_ACA_ACTIVE	ACA Active (needs to be cleared)
ERR_CHECK_CONDITION	ERR_ATA_ERR_BIT_SET	The ATA error bit is set
-40	ERR_COMMANDS_OUTSTANDING	There are outstanding commands in the
-41	ERR_APT_REGISTERS_INVALID	Sent an ATA pass-through command, bu date the shadow registers on completion
-42	ERR_DEVICE_REMOVED	The device was removed
-43	ERR_ATA_INTERNAL	Unable to determine if command succ ATA internal error
-44	ERR_USER_ABORTED	Command Aborted by User
-45	ERR_NO_SUCH_QUEUE	User tried to reference a queue that does
ERR_CHECK_CONDITION	ERR_NVME_NONZERO_SF	NVMe command completed without a 0
-46	ERR_WINDOWS_ERROR	Windows errors returned by GetLastErr
-47	ERR_XFER_LEN_ERROR	Max Xfer len request exceeded
-48	ERR_NO_CMDS_COMPLETE	Tried to receive an NVMe command wh none avaliable
-49	ERR_TAG_IN_USE	Tried to send a command with a tag/ci ready in use
-50	IU_NONZERO_STATUS	A SOP/PQI command completed successful status
-51	ERR_BAD_METADATA	A read completed successfully, but with information
-52	ERR_UNSUITABLE_SGL	A custom SGL couldn't be used with the settings
-53	ERR_TASK_MANAGE_FAILED	A Task Management Request failed
-54	ERR_INTERRUPT_VECTOR	Interrupt vector value out of range
-55	ERR_SLOT_FULL	The slot in target port is full(not same as
-56	ERR_NATIVE_LINUX_LOADER	Linux native driver loader state
-57	ERR_NATIVE_LINUX_UNLOADER	Linux native driver loader state
-58	ERR_NATIVE_LINUX_LOADER_MMAP	Linux native driver loader state
-59	ERR_NATIVE_LINUX_LOADER_MOD_NOT_FOUND	Linux native driver loader state
-60	ERR_DEVICE_DISCONNECTED	The device lost connection
-61	ERR_UNEXPECTED_TAG	Received a different response tag than mand
-101	ERR_PARSE_SYNTAX	Syntax Error
-102	ERR_PARSE_INTERNAL	Internal Parsing Error
-103	ERR_CMD_FORMATION	Problems Forming CDB/ATA bytes
-104	ERR_CMD_INTERNAL	Internal TCL error
-105	ERR_CMD_UNKNOWN_PARM	Unknown Parameter
-106	ERR_UIL_CREATION	Problems creating UIL
-107	ERR_OUT_OF_RANGE	Parameter out of range

continu

continued from previous page

Error Code	C Symbol	Description
-108	ERR_OUT_OF_MEMORY	Out Of Memory
-109	ERR_BUFFER_OVERFLOW	Buffer Overflow
-110	ERR_BAD_UIL	Command sent to bad UIL
-111	ERR_FILE_NOT_FOUND	File not found
-112	ERR_FILE_IO	IO Error reading/writing file
-113	ERR_BAD_COMMAND_TYPE	Command 'packet type' is unknown
-114	ERR_BUFFER_IN_USE	Buffer is in use by a queued command
-115	ERR_COMMAND_NOT_SUPPORTED	The currently selected power toggle device does not support this command
-116	ERR_I2C_SLAVE_NOT_FOUND	I2C slave was not found
-117	ERR_COMMAND_VALIDATION	The same named data in same offset are used in multiple descriptions

Appendix C
SCSI Commands

C.1 change_definition

Command Name(s): `change_definition,chdef`

Description: Changes Drive Definition

Default Parm Order: `def_param, vendor_spec1, vendor_spec2, vendor_spec3, password, par_ls_lngth`

Buffer Data Sent: `<par_ls_lngth>` Bytes

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
`-vendor_spec4`	[0,0x7]	`0x0`	Vendor Specific 4
`-format_4K5XX`	[0,0x3]	`0x0`	<No Description Available>
`-unlock`	[0,1]	`0x0`	unlock
`-hr`	[0,1]	`0x0`	Hard Reset
`-vendor_spec2`	[0,0xF]	`0x1`	Vendor Specific 2
`-def_param`	[0,0xF]	`0x4`	Definition Parameter
`-vendor_spec3`	[0,0x7]	`0x0`	Vendor Specific 3
`-vendor_spec1`	[0,0x1F]	`0x0`	Vendor Specific 1
`-password`	3 bytes	`0x0`	Password
`-par_ls_lngth`	[0,0xFF]	`0x0`	Parameter List Length in Bytes
`-control_byte`	[0,0xFF]	`0x0`	NACA | FLAG | LINK
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-si`	[0,?]	`<current>`	Temporary send buffer override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.2 close_zone

Command Name(s): `close_zone`

Description: Performs one or more reset write pointer operations

Default Parm Order: `zone_id`

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
`-zone_id`	8 bytes	`0x0`	The ZONE ID field specifies the
			lowest LBA of a write pointer
			zone
`-all`	[0,1]	`0x0`	An ALL bit set to one specifies
			that the device server shall
			perform a close zone operation
			on all zones with a zone
			condition of EXPLICIT OPEN or
			IMPLICIT OPEN
`-control_byte`	[0,0xFF]	`0x0`	NACA | FLAG | LINK
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.3 e6

Command Name(s): `e6,log_dump`

Description: Retrieve internal drive logs.

Default Parm Order: `offset, alloc, mode`

Buffer Data Sent: *None*

Buffer Data Received: `<recv_bytes>` Bytes

Parameters:

Name	Range	Default	Description
-block_mode	[0,1]	0x0	<No Description Available>
-mode	[0,0xFF]	0x0	Reserved for future use
-offset	3 bytes	0x0	Byte offset into log data
-alloc	3 bytes	0x0	<No Description Available>
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.4 finish_zone

Command Name(s): finish_zone

Description: Performs one or more reset write pointer operations

Default Parm Order: zone_id

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-zone_id	8 bytes	0x0	The ZONE ID field specifies the
			lowest LBA of a write pointer
			zone
-all	[0,1]	0x0	An ALL bit set to one specifies
			that the device server shall
			perform a finish zone operation
			on all zones with a zone
			condition of EXPLICIT OPEN,
			IMPLICIT OPEN and CLOSED
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.5 format_unit

Command Name(s): `format_unit,fmt`

Description: Performs a physical format of the drive media.

Default Parm Order: `fmt_data, cmp_lst, def_ls_frmt`

Buffer Data Sent: `<send_size>` Bytes

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
`-send_size`	4 bytes	`0x0`	Number of buffer bytes to send
`-fmt_prot_info`	[0,0x3]	`0x0`	<No Description Available>
`-longlist`	[0,1]	`0x0`	Parameter list contains a long
			parameter.
`-fmt_data`	[0,1]	`0x0`	FmtData
`-cmp_lst`	[0,1]	`0x0`	CmpLst
`-def_ls_frmt`	[0,0x7]	`0x0`	Defect List Format
`-vendor_unqu`	[0,0xFF]	`0x0`	Vendor Unique
`-intrleve_fac`	[0,0xFF]	`0x0`	Interleave Factor
`-ffmt`	[0,0x3]	`0x0`	Fast Format Mode
`-control_byte`	[0,0xFF]	`0x0`	NACA | FLAG | LINK
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-si`	[0,?]	`<current>`	Temporary send buffer override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.6 get_physical_element_status

Command Name(s): `get_physical_element_status`

Description: The device return status information for physical elements within the device.

Default Parm Order: `start_elem, page_cnt, filter, report_type`

Buffer Data Sent: *None*

Buffer Data Received: `<page_cnt>` Bytes

Parameters:

Name	Range	Default	Description
-page_cnt	[0,0xFFFF]	0x1	The number of 512-byte data
			blocks to transfer.
-start_elem	6 bytes	0x0	The element identifier of the
			first physical element
			addressed.
-filter	[0,0x3]	0x0	00b : All physical status / 01 :
			Only physical element status.
-report_type	[0,0xF]	0x0	0h : physical elements / 1h :
			storage elements.
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)
-dummy	[0,1]	0	Don't actually send the command
-reserved_area	[0,1]	0	1 if the command is issued to a
			reserved area, 0 if the command

C.7 inquiry

Command Name(s): inquiry,inq

Description: Performs a device inquiry.

Default Parm Order: pagecode, alloc

Buffer Data Sent: *None*

Buffer Data Received: <alloc> Bytes

Parameters:

Name	Range	Default	Description
-cmddt	[0,1]	0x0	Include Command Support Data
-evpd	[0,1]	0x0	Enable Vendor Product Data
-pagecode	[0,0xFF]	0x0	Page Code
-alloc	[0,0xFFFF]	0x100	Allocation Length in Bytes
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.8 io10

Command Name(s): io10

Description: Send a Generic 10 byte CDB

Default Parm Order: send_size, recv_size, b0, b1, b2, b3, b4, b5, b6, b7, b8, b9

Buffer Data Sent: <send_size> Bytes

Buffer Data Received: <recv_size> Bytes

Parameters:

Name	Range	Default	Description
-send_size	4 bytes	0x0	Number of buffer bytes to send
-recv_size	4 bytes	0x0	Number of buffer bytes returned
-b0	[0,0xFF]	0x0	Byte 0
-b1	[0,0xFF]	0x0	Byte 1
-b2	[0,0xFF]	0x0	Byte 2
-b3	[0,0xFF]	0x0	Byte 3
-b4	[0,0xFF]	0x0	Byte 4
-b5	[0,0xFF]	0x0	Byte 5
-b6	[0,0xFF]	0x0	Byte 6
-b7	[0,0xFF]	0x0	Byte 7
-b8	[0,0xFF]	0x0	Byte 8
-b9	[0,0xFF]	0x0	Byte 9
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.9 io12

Command Name(s): io12

Description: Send a Generic 12 byte CDB

Default Parm Order: send_size, recv_size, b0, b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11

Buffer Data Sent: <send_size> Bytes

Buffer Data Received: <recv_size> Bytes

Parameters:

Name	Range	Default	Description
-send_size	4 bytes	0x0	Number of buffer bytes to send
-recv_size	4 bytes	0x0	Number of buffer bytes returned
-b0	[0,0xFF]	0x0	Byte 0
-b1	[0,0xFF]	0x0	Byte 1
-b2	[0,0xFF]	0x0	Byte 2
-b3	[0,0xFF]	0x0	Byte 3
-b4	[0,0xFF]	0x0	Byte 4
-b5	[0,0xFF]	0x0	Byte 5
-b6	[0,0xFF]	0x0	Byte 6
-b7	[0,0xFF]	0x0	Byte 7
-b8	[0,0xFF]	0x0	Byte 8
-b9	[0,0xFF]	0x0	Byte 9
-b10	[0,0xFF]	0x0	Byte 10
-b11	[0,0xFF]	0x0	Byte 11
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.10 io16

Command Name(s): io16

Description: Send a Generic 16 byte CDB

Default Parm Order: send_size, recv_size, b0, b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12, b13, b14, b15

Buffer Data Sent: <send_size> Bytes

Buffer Data Received: <recv_size> Bytes

Parameters:

Name	Range	Default	Description
-send_size	4 bytes	0x0	Number of buffer bytes to send
-recv_size	4 bytes	0x0	Number of buffer bytes returned
-b0	[0,0xFF]	0x0	Byte 0
-b1	[0,0xFF]	0x0	Byte 1
-b2	[0,0xFF]	0x0	Byte 2
-b3	[0,0xFF]	0x0	Byte 3
-b4	[0,0xFF]	0x0	Byte 4
-b5	[0,0xFF]	0x0	Byte 5
-b6	[0,0xFF]	0x0	Byte 6
-b7	[0,0xFF]	0x0	Byte 7
-b8	[0,0xFF]	0x0	Byte 8
-b9	[0,0xFF]	0x0	Byte 9
-b10	[0,0xFF]	0x0	Byte 10
-b11	[0,0xFF]	0x0	Byte 11
-b12	[0,0xFF]	0x0	Byte 12
-b13	[0,0xFF]	0x0	Byte 13
-b14	[0,0xFF]	0x0	Byte 14
-b15	[0,0xFF]	0x0	Byte 15
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.11 io32

Command Name(s): io32

Description: Send a Generic 32 byte CDB

Default Parm Order: send_size, recv_size, b0, b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12, b13, b14, b15, b16, b17, b18, b19, b20, b21, b22, b23, b24, b25, b26, b27, b28, b29, b30, b31

Buffer Data Sent: <send_size> Bytes

Buffer Data Received: <recv_size> Bytes

Parameters:

Name	Range	Default	Description
-send_size	4 bytes	0x0	Number of buffer bytes to send
-recv_size	4 bytes	0x0	Number of buffer bytes returned
-b0	[0,0xFF]	0x0	Byte 0
-b1	[0,0xFF]	0x0	Byte 1
-b2	[0,0xFF]	0x0	Byte 2
-b3	[0,0xFF]	0x0	Byte 3
-b4	[0,0xFF]	0x0	Byte 4
-b5	[0,0xFF]	0x0	Byte 5
-b6	[0,0xFF]	0x0	Byte 6
-b7	[0,0xFF]	0x0	Byte 7
-b8	[0,0xFF]	0x0	Byte 8
-b9	[0,0xFF]	0x0	Byte 9
-b10	[0,0xFF]	0x0	Byte 10
-b11	[0,0xFF]	0x0	Byte 11
-b12	[0,0xFF]	0x0	Byte 12
-b13	[0,0xFF]	0x0	Byte 13
-b14	[0,0xFF]	0x0	Byte 14
-b15	[0,0xFF]	0x0	Byte 15
-b16	[0,0xFF]	0x0	Byte 16
-b17	[0,0xFF]	0x0	Byte 17
-b18	[0,0xFF]	0x0	Byte 18
-b19	[0,0xFF]	0x0	Byte 19
-b20	[0,0xFF]	0x0	Byte 20
-b21	[0,0xFF]	0x0	Byte 21
-b22	[0,0xFF]	0x0	Byte 22
-b23	[0,0xFF]	0x0	Byte 23
-b24	[0,0xFF]	0x0	Byte 24
-b25	[0,0xFF]	0x0	Byte 25
-b26	[0,0xFF]	0x0	Byte 26
-b27	[0,0xFF]	0x0	Byte 27
-b28	[0,0xFF]	0x0	Byte 28
-b29	[0,0xFF]	0x0	Byte 29
-b30	[0,0xFF]	0x0	Byte 30
-b31	[0,0xFF]	0x0	Byte 31
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.12 io6

Command Name(s): `io6`

Description: Send a Generic 6 byte CDB

Default Parm Order: `send_size, recv_size, b0, b1, b2, b3, b4, b5`

Buffer Data Sent: `<send_size>` Bytes

Buffer Data Received: `<recv_size>` Bytes

Parameters:

Name	Range	Default	Description
`-send_size`	4 bytes	`0x0`	Number of buffer bytes to send
`-recv_size`	4 bytes	`0x0`	Number of buffer bytes returned
`-b0`	[0,0xFF]	`0x0`	Byte 0
`-b1`	[0,0xFF]	`0x0`	Byte 1
`-b2`	[0,0xFF]	`0x0`	Byte 2
`-b3`	[0,0xFF]	`0x0`	Byte 3
`-b4`	[0,0xFF]	`0x0`	Byte 4
`-b5`	[0,0xFF]	`0x0`	Byte 5
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-ri`	[0,?]	`<current>`	Temporary receive buffer
			override
`-si`	[0,?]	`<current>`	Temporary send buffer override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.13 log_select

Command Name(s): `log_select,lgsel`

Description: Clears statistical information.

Default Parm Order: `pcr, pc, sp, par_ls_lngth`

Buffer Data Sent: `<par_ls_lngth>` Bytes

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-pcr	[0,1]	0x0	Parameter Code Reset
-sp	[0,1]	0x0	Save Parameters
-pc	[0,0x3]	0x3	Page Control
-page_code	[0,0x3F]	0x0	Page Code
-sub_page_code	[0,0xFF]	0x0	Sub Page Code
-par_ls_lngth	[0,0xFFFF]	0x0	Parameter List Length in Bytes
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.14 log_sense

Command Name(s): log_sense,lgsns

Description: Retrieves statistical data about the drive

Default Parm Order: page_code, alloc, pc, sp, ppc, par_pointer

Buffer Data Sent: *None*

Buffer Data Received: <alloc> Bytes

Parameters:

Name	Range	Default	Description
-ppc	[0,1]	0x0	Parameter Pointer Control
-sp	[0,1]	0x0	Save Parameters
-pc	[0,0x3]	0x1	Page Control
-page_code	[0,0x3F]	0x0	Page Code
-sub_page_code	[0,0xFF]	0x0	Sub Page Code; must be 0 in
			legacy products
-par_pointer	[0,0xFFFF]	0x0	Parameter Pointer
-alloc	[0,0xFFFF]	0xE	Allocation Length in Bytes
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.15 logical_depop_fmt_unit

Command Name(s): logical_depop_fmt_unit

Description: Initiates spc format to exclude specified heads

Default Parm Order: <No Default Parms>

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-feature	[0,0xFF]	0x2	<No Description Available>
-heads	4 bytes	0x0	The heads to exclude
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.16 logical_depop_inq

Command Name(s): logical_depop_inq

Description: Allows initiator to retrieve info on heads depoped

Default Parm Order: `<No Default Parms>`

Buffer Data Sent: *None*

Buffer Data Received: `8` Bytes

Parameters:

Name	Range	Default	Description
`-feature`	[0,0xFF]	`0x1`	<No Description Available>
`-control`	[0,0xFF]	`0x0`	<No Description Available>
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-ri`	[0,?]	`<current>`	Temporary receive buffer
			override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.17 mode_select10

Command Name(s): `mode_select10,mdsl10`

Description: Specifies device parameters to the target.

Default Parm Order: `par_ls_lngth, sp`

Buffer Data Sent: `<par_ls_lngth>` Bytes

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-pf	[0,1]	0x1	Page Format
-sp	[0,1]	0x0	Save Pages
-par_ls_lngth	[0,0xFFFF]	0x20	Parameter List Length in Bytes
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.18 mode_select6

Command Name(s): mode_select6,mdsl6

Description: Specifies device parameters to the target.

Default Parm Order: par_ls_lngth, sp

Buffer Data Sent: <par_ls_lngth> Bytes

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-pf	[0,1]	0x1	Page Format
-sp	[0,1]	0x0	Save Pages
-par_ls_lngth	[0,0xFF]	0x1C	Parameter List Length in Bytes
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.19 mode_sense10

Command Name(s): mode_sense10,mdsn10

Description: Reports various device parameters.

Default Parm Order: `page_code, alloc`

Buffer Data Sent: *None*

Buffer Data Received: `<alloc>` Bytes

Parameters:

Name	Range	Default	Description
`-longlba`	[0,1]	`0x0`	If 1, parameter data can be
			returned with the LONGLBA bit on
`-dbd`	[0,1]	`0x0`	Disable Block Descriptor
`-pcf`	[0,0x3]	`0x0`	Page Control Field
`-page_code`	[0,0x3F]	`0x3F`	Page Code
`-sub_page_code`	[0,0xFF]	`0x0`	Sub Page Code
`-alloc`	[0,0xFFFF]	`0xFF`	Allocation Length in Bytes
`-control_byte`	[0,0xFF]	`0x0`	NACA | FLAG | LINK
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-ri`	[0,?]	`<current>`	Temporary receive buffer
			override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.20 mode_sense6

Command Name(s): `mode_sense6,mdsn6`

Description: Reports various device parameters.

Default Parm Order: `page_code, alloc`

Buffer Data Sent: *None*

Buffer Data Received: `<alloc>` Bytes

Parameters:

Name	Range	Default	Description
-dbd	[0,1]	0x0	Disable Block Descriptor
-pcf	[0,0x3]	0x0	Page Control Field
-page_code	[0,0x3F]	0x3F	Page Code
-sub_page_code	[0,0xFF]	0x0	Sub Page Code
-alloc	[0,0xFF]	0xFF	Allocation Length in Bytes
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.21 open_zone

Command Name(s): open_zone

Description: Performs one or more reset write pointer operations

Default Parm Order: zone_id

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-zone_id	8 bytes	0x0	The ZONE ID field specifies the
			lowest LBA of a write pointer
			zone
-all	[0,1]	0x0	An ALL bit set to one specifies
			that the device server shall
			perform a zone management
			operation and then an open zone
			operation on all CLOSED zones
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.22 persistent_reserve_in

Command Name(s): `persistent_reserve_in,pri`

Description: Obtains info about persistent reservations.

Default Parm Order: `ser_action, alloc`

Buffer Data Sent: *None*

Buffer Data Received: `<alloc>` Bytes

Parameters:

Name	Range	Default	Description
`-ser_action`	[0,0x1F]	`0x0`	Service Action
`-alloc`	[0,0xFFFF]	`0x8`	Allocation Length in Bytes
`-control_byte`	[0,0xFF]	`0x0`	NACA | FLAG | LINK
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-ri`	[0,?]	`<current>`	Temporary receive buffer
			override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.23 persistent_reserve_out

Command Name(s): `persistent_reserve_out,pro`

Description: Reserves drive for a particular initiator.

Default Parm Order: `ser_action, type, par_ls_lngth`

Buffer Data Sent: `<par_ls_lngth>` Bytes

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-ser_action	[0,0x1F]	0x0	Service Action
-scope	[0,0xF]	0x0	Scope Code
-type	[0,0xF]	0x0	Type Code
-par_ls_lngth	[0,0xFFFF]	0x18	Parameter List Length in Bytes
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.24 prefetch

Command Name(s): prefetch,pref

Description: Requests that the drive transfer data to the cache.

Default Parm Order: lba, trans_length

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-immed	[0,1]	0x0	Immediate bit
-rel_adr	[0,1]	0x0	Relative Block Address
-lba	4 bytes	0x0	Logical Block Address
-trans_length	[0,0xFFFF]	0x1	Transfer Length in Blocks
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.25 prefetch16

Command Name(s): prefetch16,pref16

Description: Requests that the drive transfer data to the cache.

Default Parm Order: `lba, trans_len`

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
`-immed`	[0,1]	`0x0`	Immediate bit
`-lba`	8 bytes	`0x0`	Logical Block Address
`-trans_len`	4 bytes	`0x0`	Transfer Length in Blocks
`-grp_num`	[0,0x1F]	`0x0`	Grp which attributes are
			collected
`-control_byte`	[0,0xFF]	`0x0`	NACA | FLAG | LINK
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.26 read10

Command Name(s): `read10,r10,rd10`

Description: Reads blocks of memory from the disk.

Default Parm Order: `lba, translen, rdprotect`

Buffer Data Sent: *None*

Buffer Data Received: `<translen>` Blocks

Parameters:

Name	Range	Default	Description
-rdprotect	[0,0x7]	0x0	EndToEnd RdProtect field
-dpo	[0,1]	0x0	Disable Page Out
-fua	[0,1]	0x0	Force Unit Access
-rarc	[0,1]	0x0	Modifies read recovery behavior
-reladr	[0,1]	0x0	Relative Block Address
-lba	4 bytes	0x0	Logical Block Address
-translen	[0,0xFFFF]	0x1	Transfer Length in Blocks
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.27 read12

Command Name(s): read12,r12,rd12

Description: Reads blocks of memory from the disk.

Default Parm Order: lba, translen, rdprotect

Buffer Data Sent: *None*

Buffer Data Received: <translen> Blocks

Parameters:

Name	Range	Default	Description
-rdprotect	[0,0x7]	0x0	EndToEnd RdProtect field
-dpo	[0,1]	0x0	Disable Page Out
-fua	[0,1]	0x0	Force Unit Access
-rarc	[0,1]	0x0	Modifies read recovery behavior
-fua_nv	[0,1]	0x0	FUA Non-Volatile Cache
-lba	4 bytes	0x0	Logical Block Address
-translen	4 bytes	0x1	Transfer Length in Blocks
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.28 read16

Command Name(s): read16,r16,rd16

Description: Reads blocks of memory from the disk.

Default Parm Order: lba, translen, rdprotect

Buffer Data Sent: *None*

Buffer Data Received: <translen> Blocks

Parameters:

Name	Range	Default	Description
-rdprotect	[0,0x7]	0x0	EndToEnd RdProtect field
-dpo	[0,1]	0x0	Disable Page Out
-fua	[0,1]	0x0	Force Unit Access
-rarc	[0,1]	0x0	Modifies read recovery behavior
-fua_nv	[0,1]	0x0	Force Unit Access Non-Volatile
			Cache
-lba	8 bytes	0x0	Logical Block Address
-translen	4 bytes	0x1	Transfer Length in Blocks
-group_num	[0,0xF]	0x0	Group Number
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.29 read32

Command Name(s): read32,r32,rd32

Description: Reads blocks of memory from the disk.

Default Parm Order: lba, translen

Buffer Data Sent: *None*

Buffer Data Received: <translen> Blocks

Parameters:

Name	Range	Default	Description
-control_byte	[0,0xFF]	0x0	VU | Reserved | FLAG | LINK
-add_cdb_len	[0,0xFF]	0x18	Additional CDB Length
-serv_action	[0,0xFFFF]	0x9	Service Action
-rdprotect	[0,0x7]	0x0	EndToEnd RdProtect field
-dpo	[0,1]	0x0	Disable Page Out
-fua	[0,1]	0x0	Force Unit Access
-rarc	[0,1]	0x0	Modifies read recovery behavior
-fua_nv	[0,1]	0x0	Force Unit Access Non-Volatile
			Cache
-lba	8 bytes	0x0	Logical Block Address
-lbr_tag_msb	[0,0xFFFF]	0xFFFF	Expected Initial Logical Block
			Reference Tag MSB
-lbr_tag_lsb	[0,0xFFFF]	0xFFFF	Expected Initial Logical Block
			Reference Tag LSB
-lba_tag	[0,0xFFFF]	0x0	Logical Block Application Tag
-lba_tag_mask	[0,0xFFFF]	0x0	Logical Block Application Tag
			Mask
-translen	4 bytes	0x1	Transfer Length in Blocks
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.30 read6

Command Name(s): read6,r6,rd6

Description: Reads blocks of memory from the disk.

Default Parm Order: lba, translen

Buffer Data Sent: *None*

Buffer Data Received: <translen> Blocks

Parameters:

Name	Range	Default	Description
-lba	[0,0x1FFFFF]	0x0	Logical Block Address
-translen	[0,0xFF]	0x1	Transfer Length in Blocks
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.31 read_buffer

Command Name(s): read_buffer, rdbuf

Description: Diagnostic function for memory test.

Default Parm Order: buff_offset, alloc, buffer_id, mode

Buffer Data Sent: *None*

Buffer Data Received: <recv_bytes> Bytes

Parameters:

Name	Range	Default	Description
-block_mode	[0,1]	0x0	When enabled, buff_offset &
			alloc are in block units
-mode	[0,0x1F]	0x0	Mode
-buffer_id	[0,0xFF]	0x0	Buffer ID
-buff_offset	3 bytes	0x0	Buffer Offset
-alloc	3 bytes	0x20	Allocation Length in Bytes
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)
-dummy	[0,1]	0	Don't actually send the command
-reserved_area	[0,1]	0	1 if the command is issued to a
			reserved area, 0 if the command

C.32 read_buffer32

Command Name(s): `read_buffer32,rdbuf32`

Description: Diagnostic function for memory test.

Default Parm Order: `buff_offset, alloc, buffer_id, mode`

Buffer Data Sent: *None*

Buffer Data Received: `<recv_bytes>` Bytes

Parameters:

Name	Range	Default	Description
`-control_byte`	[0,0xFF]	`0x0`	NACA | FLAG | LINK
`-block_mode`	[0,1]	`0x0`	When enabled, buff_offset &
			alloc are in block units
`-mode`	[0,0x1F]	`0x0`	Mode
`-add_cdb_len`	[0,0xFF]	`0x18`	<No Description Available>
`-service_actio`	[0,0xFFFF]	`0xFF3C`	<No Description Available>
`n`			
`-buffer_id`	4 bytes	`0x0`	Buffer ID
`-buff_offset`	8 bytes	`0x0`	Buffer Offset
`-alloc`	8 bytes	`0x20`	Allocation Length in Bytes
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-ri`	[0,?]	`<current>`	Temporary receive buffer
			override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.33 read_capacity

Command Name(s): `read_capacity,rdcap`

Description: Returns info regarding the capacity of the drive.

Default Parm Order: `lba, pmi`

Buffer Data Sent: *None*

Buffer Data Received: `8` Bytes

Parameters:

Name	Range	Default	Description
-rel_adr	[0,1]	0x0	Relative Address
-lba	4 bytes	0x0	Logical Block Address
-pmi	[0,1]	0x0	Partial Medium Indicator
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.34 read_capacity16

Command Name(s): read_capacity16,rdcap16

Description: Returns info regarding the capacity of the drive.

Default Parm Order: lba, pmi

Buffer Data Sent: *None*

Buffer Data Received: 32 Bytes

Parameters:

Name	Range	Default	Description
-servact	[0,0xFF]	0x10	<No Description Available>
-lba	8 bytes	0x0	Logical Block Address
-len	4 bytes	0x20	<No Description Available>
-pmi	[0,1]	0x0	Partial Medium Indicator
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.35 read_defect_data10

Command Name(s): read_defect_data10,rdmap10

Description: Requests that the target transfer medium defect data.

Default Parm Order: `def_ls_frmt, alloc`

Buffer Data Sent: *None*

Buffer Data Received: `<alloc>` Bytes

Parameters:

Name	Range	Default	Description
`-p_list`	[0,1]	`0x1`	Primary Defect List
`-g_list`	[0,1]	`0x0`	The Grown Defect List
`-def_ls_frmt`	[0,0x7]	`0x5`	Defect List Format
`-alloc`	[0,0xFFFF]	`0x4`	Allocation Length in Bytes
`-control_byte`	[0,0xFF]	`0x0`	NACA | FLAG | LINK
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-ri`	[0,?]	`<current>`	Temporary receive buffer
			override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.36 read_defect_data12

Command Name(s): `read_defect_data12,rdd12`

Description: Requests that the target transfer medium defect data.

Default Parm Order: `def_ls_frmt, alloc`

Buffer Data Sent: *None*

Buffer Data Received: `<alloc>` Bytes

Parameters:

Name	Range	Default	Description
-p_list	[0,1]	0x1	Primary Defect List
-g_list	[0,1]	0x0	The Grown Defect List
-def_ls_frmt	[0,0x7]	0x5	Defect List Format
-addr_desc_idx	4 bytes	0x0	Address Descriptor Index
-alloc	4 bytes	0x8	Allocation Length in Bytes
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.37 read_initialization_pattern

Command Name(s): read_initialization_pattern,read_init_patt

Description: Requests that device server transfer a single logical block including protection bytes (DIF) of currently Data Pattern set for UNMAPPED LBA to the Data-IN buffer.

Default Parm Order: translen

Buffer Data Sent: *None*

Buffer Data Received: <translen> Bytes

Parameters:

Name	Range	Default	Description
-serv_act	[0,0x7]	0x1	Service Action
-translen	4 bytes	0x200	Transfer Length in Bytes. Should
			be the formatted block size
			(512, 520, 524, or 528).
			Defaults to 512
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.38 read_long

Command Name(s): `read_long,rdlong`

Description: Drive transfers one block of data to initiator.

Default Parm Order: `lba, trans_length`

Buffer Data Sent: *None*

Buffer Data Received: `<trans_length>` Bytes

Parameters:

Name	Range	Default	Description
`-pblock`	[0,1]	`0x0`	Return entire phy block w/ the
			logical block
`-cort`	[0,1]	`0x0`	Corrected Bit
`-rel_adr`	[0,1]	`0x0`	Relative Block Address
`-lba`	4 bytes	`0x0`	Logical Block Address
`-trans_length`	[0,0xFFFF]	`0x228`	Transfer Length in Bytes
`-control_byte`	[0,0xFF]	`0x0`	NACA | FLAG | LINK
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-ri`	[0,?]	`<current>`	Temporary receive buffer
			override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.39 read_long16

Command Name(s): `read_long16,rdlong16`

Description: Drive transfers one block of data to intiator.

Default Parm Order: `lba, trans_len`

Buffer Data Sent: *None*

Buffer Data Received: `<trans_len>` Bytes

Parameters:

Name	Range	Default	Description
-serv_act	[0,0x1F]	0x11	<No Description Available>
-lba	8 bytes	0x0	Logical Block Address
-trans_len	[0,0xFFFF]	0x0	Transfer Length in Bytes
-pblock	[0,1]	0x0	Return entire phy block w/ the
			logical block
-cort	[0,1]	0x0	Corrected Bit
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.40 reassign_blocks

Command Name(s): reassign_blocks,reas

Description: Reassigns specified logical blocks.

Default Parm Order: <No Default Parms>

Buffer Data Sent: <send_length> Bytes

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-send_length	4 bytes	0x0	<No Description Available>
-long_lba	[0,1]	0x0	Turn on to use 8 byte LBAs
-long_list	[0,1]	0x0	Turn on to increase the defect
			list length
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.41 receive_diagnostic_results

Command Name(s): `receive_diagnostic_results,rcvdg`

Description: Sends analysis data to initiator.

Default Parm Order: `page_code, par_ls_lngth`

Buffer Data Sent: *None*

Buffer Data Received: `<par_ls_lngth>` Bytes

Parameters:

Name	Range	Default	Description
`-pcv`	[0,1]	`0x1`	Page Code Valid
`-page_code`	[0,0xFF]	`0x0`	Page Code
`-par_ls_lngth`	[0,0xFFFF]	`0x200`	Parameter List Length in Bytes
`-control_byte`	[0,0xFF]	`0x0`	NACA | FLAG | LINK
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-ri`	[0,?]	`<current>`	Temporary receive buffer
			override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.42 release10

Command Name(s): `release10,rel10`

Description: Releases a previously reserved LUN.

Default Parm Order: `reserv_id`

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-thirdpty	[0,1]	0x0	3rdPty Bit
-ext	[0,1]	0x0	Extent Bit
-td_pty_dv_id	[0,0xFF]	0x0	Third Party Device ID
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.43 release6

Command Name(s): release6,rel6

Description: Releases a previously reserved LUN.

Default Parm Order: reserv_id

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-thirdpty	[0,1]	0x0	3rdPty Bit
-third_pty_id	[0,0x7]	0x0	3rd Party ID
-ext	[0,1]	0x0	Extent Bit
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.44 remove_element_and_truncate

Command Name(s): remove_element_and_truncate

Description: Performs depopulating a storage element and truncating the reported capacity of the media.

Default Parm Order: lba, elem_id

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-elem_id	4 bytes	0x0	The element identifier
			associated with the storage
			element to be depopulated.
-lba	6 bytes	0x0	The maximum LBA after completion
			of this command without error.
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)
-dummy	[0,1]	0	Don't actually send the command
-reserved_area	[0,1]	0	1 if the command is issued to a
			reserved area, 0 if the command

C.45 report_dev_id

Command Name(s): report_dev_id, rdi

Description: Requests that the device server send device information important to the application client

Default Parm Order: alloc

Buffer Data Sent: *None*

Buffer Data Received: <alloc> Bytes

Parameters:

Name	Range	Default	Description
-serv_act	[0,0x1F]	0x5	Service Action
-lun	[0,0xFFFF]	0x0	Logical Unit Number
-alloc	4 bytes	0xFF	Allocation Length
-info_type	[0,0x7F]	0x0	Information Type. Support 0 or
			7F
-control_byte	[0,0xFF]	0x0	FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.46 report_lun

Command Name(s): report_lun,rlun

Description: Returns the known Logical Unit Numbers to the initiator.

Default Parm Order: alloc

Buffer Data Sent: *None*

Buffer Data Received: <alloc> Bytes

Parameters:

Name	Range	Default	Description
-alloc	4 bytes	0x10	Allocation Length in Bytes
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.47 report_provisioning_init_patt

Command Name(s): report_provisioning_init_patt,rpip

Description: Requests that the device server transfer the provisioning initialization pattern

Default Parm Order: `allocation_length, control_byte`

Buffer Data Sent: *None*

Buffer Data Received: `<allocation_length>` Bytes

Parameters:

Name	Range	Default	Description
`-service_actio`	[0,0x1F]	`0x1D`	<No Description Available>
`n`			
`-allocation_le`	4 bytes	`0x0`	<No Description Available>
`ngth`			
`-control_byte`	[0,0xFF]	`0x0`	<No Description Available>
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-ri`	[0,?]	`<current>`	Temporary receive buffer
			override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.48 report_supported_opcodes

Command Name(s): `report_supported_opcodes,repsupops`

Description: Returns a list of all opcodes and service actions.

Default Parm Order: `rep_options, req_opcode, req_serv_act, alloc`

Buffer Data Sent: *None*

Buffer Data Received: `<alloc>` Bytes

Parameters:

Name	Range	Default	Description
-rctd	[0,1]	0x0	Return timeout descriptor bit
-rep_options	[0,0x7]	0x0	Reporting options.
-req_opcode	[0,0xFF]	0x0	Requested opcode.
-req_serv_act	[0,0xFFFF]	0x0	Requested service action
-alloc	4 bytes	0x600	Allocation Length in bytes
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.49 report_supported_tmf

Command Name(s): report_supported_tmf, repsuptmf

Description: Returns information on supported TMFs.

Default Parm Order: alloc

Buffer Data Sent: *None*

Buffer Data Received: <alloc> Bytes

Parameters:

Name	Range	Default	Description
-repd	[0,1]	0x0	Return extended parameter data
			bit.
-alloc	4 bytes	0x16	Allocation Length in bytes
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.50 report_timestamp

Command Name(s): `report_timestamp,rts`

Description: Requests that the device server return the current value of a device clock.

Default Parm Order: `alloc`

Buffer Data Sent: *None*

Buffer Data Received: `<alloc>` Bytes

Parameters:

Name	Range	Default	Description
`-alloc`	4 bytes	`0xC`	<No Description Available>
`-control_byte`	[0,0xFF]	`0x0`	FLAG | LINK
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-ri`	[0,?]	`<current>`	Temporary receive buffer
			override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.51 report_zones

Command Name(s): `report_zones`

Description: Return the zone structure of the zoned block device.

Default Parm Order: `z_start_lba, alloc, reporting_opt`

Buffer Data Sent: *None*

Buffer Data Received: `<alloc>` Bytes

Parameters:

Name	Range	Default	Description
-z_start_lba	8 bytes	0x0	The ZONE START LBA field
			specifies the starting LBA of
			the first zone to be reported.
-alloc	4 bytes	0x1	Allocation Length in Bytes
-partial	[0,1]	0x0	<No Description Available>
-reporting_opt	[0,0x3F]	0x0	The REPORTING OPTIONS field
			specifies the information to be
			returned in the parameter data.
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.52 report_zones_old

Command Name(s): report_zones_old

Description: Return the zone structure of the zoned block device.

Default Parm Order: z_start_lba, alloc, reporting_opt

Buffer Data Sent: *None*

Buffer Data Received: <alloc> Bytes

Parameters:

Name	Range	Default	Description
-z_start_lba	8 bytes	0x0	The ZONE START LBA field
			specifies the starting LBA of
			the first zone to be reported.
-alloc	4 bytes	0x1	Allocation Length in Bytes
-reporting_opt	[0,0xF]	0x0	The REPORTING OPTIONS field
			specifies the information to be
			returned in the parameter data.
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.53 request_sense

Command Name(s): request_sense,sns

Description: Returns the target's sense data to the initiator.

Default Parm Order: alloc

Buffer Data Sent: *None*

Buffer Data Received: <alloc> Bytes

Parameters:

Name	Range	Default	Description
-desc	[0,1]	0x0	Specify the sense data format
-alloc	[0,0xFF]	0xFC	Allocation Length in Bytes
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.54 reserve10

Command Name(s): `reserve10,res10`

Description: Used to reserve a LUN for an initiator.

Default Parm Order: `reserv_id`

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
`-thirdpty`	[0,1]	`0x0`	3rdPty
`-ext`	[0,1]	`0x0`	Extent Bit
`-td_pty_dv_id`	[0,0xFF]	`0x0`	3rd Party Device ID
`-ext_ls_lngth`	[0,0xFFFF]	`0x0`	Extent List Length
`-control_byte`	[0,0xFF]	`0x0`	NACA | FLAG | LINK
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.55 reserve6

Command Name(s): `reserve6,res6`

Description: Used to reserve a LUN for an initiator.

Default Parm Order: `reserv_id`

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-thirdpty	[0,1]	0x0	3rdPty
-third_pty_id	[0,0x7]	0x0	3rd Party ID
-ext	[0,1]	0x0	Extent Bit
-ext_ls_lngth	[0,0xFFFF]	0x0	Extent List Length
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.56 reset_write_pointer

Command Name(s): reset_write_pointer

Description: Performs one or more reset write pointer operations

Default Parm Order: zone_id

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-zone_id	8 bytes	0x0	The ZONE ID field specifies the
			zone start LBA
-reset_all	[0,1]	0x0	If RESET ALL bit set to one,
			perform a reset write pointer
			operation on all OPEN zones and
			FULL zones
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.57 reset_write_pointer_old

Command Name(s): reset_write_pointer_old

Description: Performs one or more reset write pointer operations

Default Parm Order: `zone_id`

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-zone_id	8 bytes	0x0	The ZONE ID field specifies the
			zone start LBA
-reset_all	[0,1]	0x0	If RESET ALL bit set to one,
			perform a reset write pointer
			operation on all OPEN zones and
			FULL zones
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.58 rezero_unit

Command Name(s): `rezero_unit,rezero`

Description: Requests that the target seek to LBA 0.

Default Parm Order: `<No Default Parms>`

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.59 sanitize

Command Name(s): `sanitize`

Description: Performs a sanitize operation.

Default Parm Order: `service_action, par_ls_lngth`

Buffer Data Sent: `<par_ls_lngth>` Bytes

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
`-immed`	[0,1]	`0x1`	Immediately Return Status
`-znr`	[0,1]	`0x0`	Zone No Reset
`-ause`	[0,1]	`0x0`	Allow Unrestricted Sanitize Exit
`-service_actio`	[0,0x1F]	`0x0`	0x01: Overwrite. 0x02: Block
`n`			Erase. 0x03: Cryptographic
			Erase. All Others: reserved
`-par_ls_lngth`	[0,0xFFFF]	`0x0`	Parameter List Length in Bytes
`-control_byte`	[0,0xFF]	`0x0`	NACA | FLAG | LINK
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-si`	[0,?]	`<current>`	Temporary send buffer override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.60 security_protocol_in_block

Command Name(s): `security_protocol_in_block,sec_in_blk`

Description: Retrieve security protocol information from logical unit.

Default Parm Order: `security_protocol, protocol_specific, alloc, control_byte`

Buffer Data Sent: *None*

Buffer Data Received: `<alloc>` Blocks

Parameters:

Name	Range	Default	Description
-security_prot	[0,0xFF]	0x0	0: security protocol
ocol			information. 1-6: defined by
			TCG.
-protocol_spec	[0,0xFFFF]	0x0	Depends on the security_protocol
ific			field
-alloc	4 bytes	0x0	Receive buffer allocation size
			in 512 byte blocks
-control_byte	[0,0xFF]	0x0	<No Description Available>
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.61 security_protocol_in_byte

Command Name(s): security_protocol_in_byte,sec_in_byte

Description: Retrieve security protocol information from logical unit.

Default Parm Order: security_protocol, protocol_specific, alloc, control_byte

Buffer Data Sent: *None*

Buffer Data Received: <alloc> Bytes

Parameters:

Name	Range	Default	Description
-security_prot	[0,0xFF]	0x0	0: security protocol
ocol			information. 1-6: defined by
			TCG.
-protocol_spec	[0,0xFFFF]	0x0	Depends on the security_protocol
ific			field
-alloc	4 bytes	0x0	Receive buffer allocation size
			in bytes
-control_byte	[0,0xFF]	0x0	<No Description Available>
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.62 security_protocol_out_block

Command Name(s): security_protocol_out_block,sec_out_blk

Description: Send security protocol information to logical unit.

Default Parm Order: security_protocol, protocol_specific, alloc, control_byte

Buffer Data Sent: <alloc> Blocks

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-security_prot	[0,0xFF]	0x0	0: security protocol
ocol			information. 1-6: defined by
			TCG.
-protocol_spec	[0,0xFFFF]	0x0	Depends on the security_protocol
ific			field
-alloc	4 bytes	0x0	Amount of data to send from send
			buffer in 512 byte blocks
-control_byte	[0,0xFF]	0x0	<No Description Available>
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.63 security_protocol_out_byte

Command Name(s): security_protocol_out_byte,sec_out_byte

Description: Send security protocol information to logical unit.

Default Parm Order: security_protocol, protocol_specific, alloc, control_byte

Buffer Data Sent: <alloc> Bytes

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-security_prot	[0,0xFF]	0x0	0: security protocol
ocol			information. 1-6: defined by
			TCG.
-protocol_spec	[0,0xFFFF]	0x0	Depends on the security_protocol
ific			field
-alloc	4 bytes	0x0	Amount of data to send from send
			buffer in bytes
-control_byte	[0,0xFF]	0x0	<No Description Available>
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.64 seek10

Command Name(s): seek10,sk10

Description: Requests that the drive seek to the specified LBA.

Default Parm Order: lba

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-lba	4 bytes	0x0	Logical Block Address
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.65 seek10_64lba

Command Name(s): seek10_64lba,sk10_64

Description: Requests that the drive seek to the specified LBA.

Default Parm Order: `lba`

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
`-lba`	8 bytes	`0x0`	Logical Block Address
`-control_byte`	[0,0xFF]	`0x80`	VU | Reserved | FLAG | LINK
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.66 seek6

Command Name(s): `seek6,sk6`

Description: Requests that the drive seek to the specified LBA.

Default Parm Order: `lba`

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
`-lba`	[0,0x1FFFFF]	`0x0`	Logical Block Address
`-control_byte`	[0,0xFF]	`0x0`	NACA | FLAG | LINK
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.67 send_diagnostic

Command Name(s): `send_diagnostic,sndd`

Description: Requests the drive perform a diagnostic.

Default Parm Order: `funct_code, par_ls_lngth`

Buffer Data Sent: <par_ls_lngth> Bytes

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-funct_code	[0,0x7]	0x1	Function Code
-pf	[0,1]	0x1	Page Format
-slftst	[0,1]	0x1	Self Test
-dev0fl	[0,1]	0x0	IGNORED
-unt0fl	[0,1]	0x0	IGNORED
-par_ls_lngth	[0,0xFFFF]	0x0	Parameter List Length in Bytes
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.68 set_dev_id

Command Name(s): `set_dev_id,sdi`

Description: Requests that the device server send device information important to the application client

Default Parm Order: `alloc`

Buffer Data Sent: <alloc> Bytes

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-serv_act	[0,0x1F]	0x6	Service Action
-alloc	4 bytes	0xFF	Allocation Length
-control_byte	[0,0xFF]	0x0	FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.69 set_initialization_pattern

Command Name(s): set_initialization_pattern,set_init_patt

Description: Requests that device server transfer a single logical block including protection bytes (DIF) from Data-out buffer.

Default Parm Order: translen

Buffer Data Sent: <translen> Bytes

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-serv_act	[0,0x7]	0x1	Service Action
-translen	4 bytes	0x200	Transfer Length in Bytes. Should
			be the formatted block size
			(512, 520, 524, or 528).
			Defaults to 512
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.70 set_timestamp

Command Name(s): set_timestamp,sts

Description: Requests that the device server initialize a device clock

Default Parm Order: `par_ls_lngth`

Buffer Data Sent: `<par_ls_lngth>` Bytes

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
`-par_ls_lngth`	4 bytes	`0xC`	Parameter List Length in Bytes
`-control_byte`	[0,0xFF]	`0x0`	FLAG | LINK
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-si`	[0,?]	`<current>`	Temporary send buffer override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.71 start_stop_unit

Command Name(s): `start_stop_unit,ssu,unit`

Description: Starts or stops unit.

Default Parm Order: `start, immed`

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-immed	[0,1]	0x0	Immediately Return Status
-pwr_cond_mod	[0,0xF]	0x0	Places the logical unit into a
			power condition or adjusts a
			timer. Implemented in Sonoma.
-pwr_cond	[0,0xF]	0x0	Places the logical unit into a
			power condition or adjusts a
			timer. Implemented in Sonoma.
-start	[0,1]	0x0	Start(1)/Stop(0) Spindle
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.72 synchronize_cache

Command Name(s): synchronize_cache,sync

Description: Ensures that logical blocks in the cache have their most recent data value recorded on the media

Default Parm Order: lba, num_blocks

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-immed	[0,1]	0x0	Immediate Bit
-rel_adr	[0,1]	0x0	Relative Address
-lba	4 bytes	0x0	Logical Block Address
-num_blocks	[0,0xFFFF]	0x1	Number of Blocks
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.73 synchronize_cache16

Command Name(s): `synchronize_cache16,sync16`

Description: Ensures that logical blocks in the cache have their most recent data value recorded on the media

Default Parm Order: `lba, num_blocks`

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
`-sync_nv`	[0,1]	`0x0`	Synch volatile and non-volatile
`-immed`	[0,1]	`0x0`	Immediate Bit
`-lba`	8 bytes	`0x0`	Logical Block Address
`-num_blocks`	4 bytes	`0x0`	Number of Blocks
`-grp_num`	[0,0x1F]	`0x0`	Grp which attributes are
			collected
`-control_byte`	[0,0xFF]	`0x0`	NACA | FLAG | LINK
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.74 test_unit_ready

Command Name(s): `test_unit_ready,tstrdy,tstr`

Description: Tests to see if the device is ready.

Default Parm Order: `<No Default Parms>`

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.75 unmap

Command Name(s): unmap,um

Description: Invalidates user data on the disk.

Default Parm Order: listlen

Buffer Data Sent: <listlen> Bytes

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-anchor	[0,1]	0x0	Set to 1 to anchor an LBA
-group_num	[0,0x1F]	0x0	<No Description Available>
-listlen	[0,0xFFFF]	0x18	List Length in Bytes
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.76 verify

Command Name(s): verify,ver

Description: Asks drive to verify data written on media. If bytechk=0, trans_length is # of blocks to verify internally. If bytechk=1, trans_length is also blocks being sent to drive, so send_length must be made the same as trans_length.

Default Parm Order: lba, trans_length, vrprotect

Buffer Data Sent: `Buffer` Data

Buffer Data Received:

Parameters:

Name	Range	Default	Description
`-dpo`	[0,1]	`0x0`	Disable Page Out
`-bytechk`	[0,0x3]	`0x0`	Byte Check
`-rel_adr`	[0,1]	`0x0`	Relative Address
`-lba`	4 bytes	`0x0`	Logical Block Address
`-trans_length`	[0,0xFFFF]	`0x1`	Number of blocks to verify
`-control_byte`	[0,0xFF]	`0x0`	NACA | FLAG | LINK
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-si`	[0,?]	`<current>`	Temporary send buffer override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.77 verify12

Command Name(s): `verify12,ver12`

Description: Asks drive to verify data written on media. If bytechk=0, trans_length is # of blocks to verify internally. If bytechk=1, trans_length is also blocks being sent to drive, so send_length must be made the same as trans_length.

Default Parm Order: `lba, trans_length, vrprotect`

Buffer Data Sent: `Buffer` Data

Buffer Data Received:

Parameters:

Name	Range	Default	Description
-dpo	[0,1]	0x0	Disable Page Out
-bytechk	[0,0x3]	0x0	Byte Check
-lba	4 bytes	0x0	Logical Block Address:
			previously was lba_low and
			lba_high
-trans_length	4 bytes	0x0	Number of blocks to verify
-restricted	[0,1]	0x0	<No Description Available>
-group_number	[0,0x1F]	0x0	<No Description Available>
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.78 verify16

Command Name(s): verify16,ver16

Description: Asks drive to verify data written on media. If bytechk=0, trans_length is # of blocks to verify internally. If bytechk=1, trans_length is also blocks being sent to drive, so send_length must be made the same as trans_length.

Default Parm Order: lba, trans_length, vrprotect

Buffer Data Sent: Buffer Data

Buffer Data Received:

Parameters:

Name	Range	Default	Description
-dpo	[0,1]	0x0	Disable Page Out
-bytechk	[0,0x3]	0x0	Byte Check
-lba	8 bytes	0x0	Logical Block Address:
			previously was lba_low and
			lba_high
-trans_length	4 bytes	0x0	Number of blocks to verify
-restricted	[0,1]	0x0	<No Description Available>
-group_number	[0,0x1F]	0x0	<No Description Available>
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.79 verify32

Command Name(s): verify32,ver32

Description: Asks drive to verify data written on media. If bytechk=0, trans_length is # of blocks to verify internally. If bytechk=1, trans_length is also blocks being sent to drive, so send_length must be made the same as trans_length.

Default Parm Order: lba, translen

Buffer Data Sent: Buffer Data

Buffer Data Received:

Parameters:

Name	Range	Default	Description
-add_cdb_len	[0,0xFF]	0x18	Additional CDB Length
-serv_action	[0,0xFFFF]	0xA	Service Action
-vrprotect	[0,0x7]	0x0	EndToEnd VrProtect field
-dpo	[0,1]	0x0	<No Description Available>
-bytechk	[0,0x3]	0x0	<No Description Available>
-lba	8 bytes	0x0	Logical Block Address
-lbr_tag_msb	[0,0xFFFF]	0xFFFF	Expected Initial Logical Block
			Reference Tag MSB
-lbr_tag_lsb	[0,0xFFFF]	0xFFFF	Expected Initial Logical Block
			Reference Tag LSB
-lba_tag	[0,0xFFFF]	0x0	Logical Block Application Tag
-lba_tag_mask	[0,0xFFFF]	0x0	Logical Block Application Tag
			Mask
-translen	4 bytes	0x1	Transfer Length in Blocks
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.80 vu_commit_verify

Command Name(s): vu_commit_verify

Description: Upon receipt of commit verify command, drive updates verify pointer

Default Parm Order: target_band

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-target_band	[0,0xFFFF]	0x0	Target band of commit verify
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.81 vu_define_band_type

Command Name(s): `vu_define_band_type`

Description: Define Band Type

Default Parm Order: `target_band, band_type`

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
`-target_band`	[0,0xFFFF]	`0x0`	Target Band Number
`-band_type`	[0,0x3]	`0x2`	Band Type
`-control_byte`	[0,0xFF]	`0x0`	NACA | FLAG | LINK
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.82 vu_query_band_information

Command Name(s): `vu_query_band_information`

Description: Returns information associated with bands

Default Parm Order: `target_band, alloc`

Buffer Data Sent: *None*

Buffer Data Received: `<alloc>` Bytes

Parameters:

Name	Range	Default	Description
-target_band	[0,0xFFFF]	0x1	Target band number
-alloc	4 bytes	0x60	Allocation Length in Bytes
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.83 vu_query_last_verify_error

Command Name(s): vu_query_last_verify_error

Description: Verify from last verified lba of the band through appropriate EOT

Default Parm Order: drp_level, last_band_num, alloc

Buffer Data Sent: *None*

Buffer Data Received: <alloc> Bytes

Parameters:

Name	Range	Default	Description
-drp_level	[0,0xFF]	0x0	DRP level specifies the relative
			depth of verify DRP
-last_band_num	[0,0xFFFF]	0x0	The last written band number
-alloc	[0,0xFFFF]	0x20	Allocation Length in Bytes
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-ri	[0,?]	<current>	Temporary receive buffer
			override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.84 vu_reset_write_pointer

Command Name(s): vu_reset_write_pointer

Description: Reset write pointer for the designated band

Default Parm Order: `target_band`

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-target_band	[0,0xFFFF]	0x0	Target band must be sequential
			write band with reliable write
			band
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.85 vu_set_write_pointer

Command Name(s): `vu_set_write_pointer`

Description: Move the write pointer position to start of track of given logical block address

Default Parm Order: `w_pointer_lba`

Buffer Data Sent: *None*

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-w_pointer_lba	8 bytes	0x0	Write Pointer LBA
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.86 vu_verify_squeezed_blocks

Command Name(s): `vu_verify_squeezed_blocks`

Description: Verify from last verified lba of the band through appropriate EOT

Default Parm Order: `drp_level, target_band, alloc`

Buffer Data Sent: `<alloc>` Bytes

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
`-drp_level`	[0,0xFF]	`0x0`	DRP level specifies the relative
			depth of verify DRP
`-target_band`	[0,0xFFFF]	`0x0`	Target band must be sequential
			write band
`-alloc`	[0,0xFFFF]	`0x1`	Allocation Length in Bytes
`-control_byte`	[0,0xFF]	`0x0`	NACA | FLAG | LINK
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-si`	[0,?]	`<current>`	Temporary send buffer override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.87 write10

Command Name(s): `write10,w10,wr10`

Description: Writes blocks of memory to the disk.

Default Parm Order: `lba, translen, wrprotect`

Buffer Data Sent: `<translen>` Blocks

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-wrprotect	[0,0x7]	0x0	EndToEnd WrProtect field
-dpo	[0,1]	0x0	Disable Page Out
-fua	[0,1]	0x0	Force Unit Access
-reladr	[0,1]	0x0	Relative Block Address
-lba	4 bytes	0x0	Logical Block Address
-translen	[0,0xFFFF]	0x1	Transfer Length in Blocks
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.88 write12

Command Name(s): write12,w12,wr12

Description: Writes blocks of memory to the disk.

Default Parm Order: lba, translen, wrprotect

Buffer Data Sent: <translen> Blocks

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-wrprotect	[0,0x7]	0x0	EndToEnd WrProtect field
-dpo	[0,1]	0x0	Disable Page Out
-fua	[0,1]	0x0	Force Unit Access
-fua_nv	[0,1]	0x0	FUA Non-Volatile Cache
-lba	4 bytes	0x0	Logical Block Address
-translen	4 bytes	0x1	Transfer Length in Blocks
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.89 write16

Command Name(s): `write16,w16,wr16`

Description: Reads blocks of memory from the disk.

Default Parm Order: `lba, translen, wrprotect`

Buffer Data Sent: `<translen>` Blocks

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
`-wrprotect`	[0,0x7]	`0x0`	EndToEnd WrProtect field
`-dpo`	[0,1]	`0x0`	Disable Page Out
`-fua`	[0,1]	`0x0`	Force Unit Access
`-fua_nv`	[0,1]	`0x0`	Force Unit Access Non-Volatile
			Cache
`-lba`	8 bytes	`0x0`	Logical Block Address
`-translen`	4 bytes	`0x1`	Transfer Length in Blocks
`-control_byte`	[0,0xFF]	`0x0`	NACA | FLAG | LINK
`-transport_cdb`	[0,1]	`<current>`	Convert to 0xC3 transport cmd
`-uil`	[0,?]	`<current>`	Temporary UIL override
`-dev`	[0,?]	`<current>`	Temporary device index override
`-si`	[0,?]	`<current>`	Temporary send buffer override
`-cmd_timeout`	[0,?]	`0`	Single cmd timeout override
			(0=no override)
`-set_timeout`	[0,?]	`0`	Persistent timeout override
			(0=no override)

C.90 write32

Command Name(s): `write32,w32,wr32`

Description: Writes blocks of memory to the disk.

Default Parm Order: `lba, translen`

Buffer Data Sent: `<translen>` Blocks

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-control_byte	[0,0xFF]	0x0	VU | Reserved | FLAG | LINK
-add_cdb_len	[0,0xFF]	0x18	Additional CDB Length
-serv_action	[0,0xFFFF]	0xB	Service Action
-wrprotect	[0,0x7]	0x0	EndToEnd WrProtect field
-dpo	[0,1]	0x0	Disable Page Out
-fua	[0,1]	0x0	Force Unit Access
-fua_nv	[0,1]	0x0	Force Unit Access Non-Volatile
			Cache
-lba	8 bytes	0x0	Logical Block Address
-lbr_tag_msb	[0,0xFFFF]	0xFFFF	Expected Initial Logical Block
			Reference Tag MSB
-lbr_tag_lsb	[0,0xFFFF]	0xFFFF	Expected Initial Logical Block
			Reference Tag LSB
-lba_tag	[0,0xFFFF]	0x0	Logical Block Application Tag
-lba_tag_mask	[0,0xFFFF]	0x0	Logical Block Application Tag
			Mask
-translen	4 bytes	0x1	Transfer Length in Blocks
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.91 write6

Command Name(s): write6,w6,wr6

Description: Writes blocks of memory to the disk.

Default Parm Order: lba, translen

Buffer Data Sent: <translen> Blocks

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-lba	[0,0x1FFFFF]	0x0	Logical Block Address
-translen	[0,0xFF]	0x1	Transfer Length in Blocks
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.92 write_and_verify

Command Name(s): write_and_verify,wrv

Description: Requests the drive write data and then check it.

Default Parm Order: lba, trans_length, wrprotect

Buffer Data Sent: <trans_length> Blocks

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-wrprotect	[0,0x7]	0x0	EndToEnd WrProtect field
-dpo	[0,1]	0x0	Disable Page Out
-bytechk	[0,0x3]	0x0	Byte Check
-rel_adr	[0,1]	0x0	Relative Address
-lba	4 bytes	0x0	Logical Block Address
-trans_length	[0,0xFFFF]	0x1	Transfer Length in Blocks
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.93 write_and_verify12

Command Name(s): write_and_verify12,wrv12

Description: Requests the drive write data and then check it.

Default Parm Order: `lba, trans_length, wrprotect`

Buffer Data Sent: <trans_length> Blocks

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-wrprotect	[0,0x7]	0x0	EndToEnd WrProtect field
-dpo	[0,1]	0x0	Disable Page Out
-bytechk	[0,0x3]	0x0	Byte Check
-RelAdr	[0,1]	0x0	<No Description Available>
-lba	4 bytes	0x0	Logical Block Address
-trans_length	4 bytes	0x0	Transfer Length in Blocks
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.94 write_and_verify16

Command Name(s): `write_and_verify16,wrv16`

Description: Requests the drive write data and then check it.

Default Parm Order: `lba, trans_length, wrprotect`

Buffer Data Sent: <trans_length> Blocks

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-wrprotect	[0,0x7]	0x0	EndToEnd WrProtect field
-dpo	[0,1]	0x0	Disable Page Out
-bytechk	[0,0x3]	0x0	Byte Check
-lba	8 bytes	0x0	Logical Block Address:
			previously was lba_low and
			lba_high
-trans_length	4 bytes	0x0	Transfer Length in Blocks
-restricted	[0,1]	0x0	<No Description Available>
-group_number	[0,0x1F]	0x0	<No Description Available>
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.95 write_and_verify32

Command Name(s): write_and_verify32,wrv32

Description: Requests the drive write data and then check it.

Default Parm Order: lba, translen

Buffer Data Sent: <translen> Blocks

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-control_byte	[0,0xFF]	0x0	VU | Reserved | FLAG | LINK
-add_cdb_len	[0,0xFF]	0x18	Additional CDB Length
-serv_action	[0,0xFFFF]	0xC	Service Action
-wrprotect	[0,0x7]	0x0	EndToEnd WrProtect field
-dpo	[0,1]	0x0	Disable Page Out
-bytechk	[0,0x3]	0x0	Byte Check
-lba	8 bytes	0x0	Logical Block Address
-lbr_tag_msb	[0,0xFFFF]	0xFFFF	Expected Initial Logical Block
			Reference Tag MSB
-lbr_tag_lsb	[0,0xFFFF]	0xFFFF	Expected Initial Logical Block
			Reference Tag LSB
-lba_tag	[0,0xFFFF]	0x0	Logical Block Application Tag
-lba_tag_mask	[0,0xFFFF]	0x0	Logical Block Application Tag
			Mask
-translen	4 bytes	0x1	Transfer Length in Blocks
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.96 write_atomic16

Command Name(s): write_atomic16,wa16,wra16

Description: Atomic Write of blocks of memory to the disk.

Default Parm Order: lba, translen, wrprotect

Buffer Data Sent: <translen> Blocks

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-wrprotect	[0,0x7]	0x0	EndToEnd WrProtect field
-dpo	[0,1]	0x0	Disable Page Out
-fua	[0,1]	0x0	Force Unit Access
-lba	8 bytes	0x0	Logical Block Address
-atomic_bndry	[0,0xFFFF]	0x0	Number of Atomic Operations
-translen	[0,0xFFFF]	0x1	Transfer Length in Blocks
-group_num	[0,0x1F]	0x0	Group Number
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.97 write_atomic32

Command Name(s): write_atomic32,wa32,wra32

Description: Atomic writes of blocks of memory to the disk.

Default Parm Order: lba, translen

Buffer Data Sent: <translen> Blocks

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-control_byte	[0,0xFF]	0x0	VU | Reserved | FLAG | LINK
-atomic_bndry	[0,0xFFFF]	0x0	Number of Atomic Operations
-group_num	[0,0x1F]	0x0	Group Number
-wrprotect	[0,0x7]	0x0	EndToEnd WrProtect field
-dpo	[0,1]	0x0	Disable Page Out
-fua	[0,1]	0x0	Force Unit Access
-lba	8 bytes	0x0	Logical Block Address
-lbr_tag	4 bytes	0xFFFFFFFF	Expected Initial Logical Block
			Reference Tag
-lba_tag	[0,0xFFFF]	0x0	Expected Logical Block
			Application Tag
-lba_tag_mask	[0,0xFFFF]	0x0	Logical Block Application Tag
			Mask
-translen	4 bytes	0x1	Transfer Length in Blocks
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.98 write_buffer

Command Name(s): write_buffer,writebuf

Description: Used with read_buffer to test drive's memory.

Default Parm Order: mode, par_ls_lngth

Buffer Data Sent: <send_bytes> Bytes

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-mode_specific	[0,0x7]	0x0	For all modes except 0xD, 0
			signifies byte mode transfer,
			and 4 signifies block mode
			transfer. For mode 0xD,
			mode_specific can have values of
			1,2,4, for PO_ACT, HR_ACT, and
			VSE_ACT respectively
-mode	[0,0x1F]	0x0	Mode
-buffer_id	[0,0xFF]	0x0	Buffer ID
-buff_offset	3 bytes	0x0	Buffer Offset
-par_ls_lngth	3 bytes	0x4	Parameter List Length in Bytes
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)
-dummy	[0,1]	0	Don't actually send the command
-reserved_area	[0,1]	0	1 if the command is issued to a
			reserved area, 0 if the command

C.99 write_buffer32

Command Name(s): write_buffer32,writebuf32

Description: Used with read_buffer to test drive's memory.

Default Parm Order: mode, par_ls_lngth

Buffer Data Sent: <send_bytes> Bytes

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-mode_specific	[0,0x7]	0x0	For all modes except 0xD, 0
			signifies byte mode transfer,
			and 4 signifies block mode
			transfer. For mode 0xD,
			mode_specific can have values of
			1,2,4, for PO_ACT, HR_ACT, and
			VSE_ACT respectively
-mode	[0,0x1F]	0x0	Mode
-add_cdb_len	[0,0xFF]	0x18	<No Description Available>
-service_actio	[0,0xFFFF]	0xFF3B	<No Description Available>
n			
-buffer_id	4 bytes	0x0	Buffer ID
-buff_offset	8 bytes	0x0	Buffer Offset
-par_ls_lngth	8 bytes	0x20	Parameter List Length in Bytes
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.100 write_long

Command Name(s): write_long,wrlong

Description: Requests that the drive write one block of data.

Default Parm Order: lba, trans_length

Buffer Data Sent: <trans_length> Bytes

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-cor_dis	[0,1]	0x0	<No Description Available>
-wr_uncor	[0,1]	0x0	<No Description Available>
-pblock	[0,1]	0x0	Send entire phy block w/ the
			logical block
-lba	4 bytes	0x0	Logical Block Address
-trans_length	[0,0xFFFF]	0x228	Transfer Length in Bytes
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.101 write_long16

Command Name(s): write_long16,wrlong16

Description: Requests that the drive write one block of data.

Default Parm Order: lba, trans_len

Buffer Data Sent: <trans_len> Bytes

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-cor_dis	[0,1]	0x0	<No Description Available>
-wr_uncor	[0,1]	0x0	<No Description Available>
-pblock	[0,1]	0x0	Send entire phy block w/ the
			logical block
-serv_act	[0,0x1F]	0x11	<No Description Available>
-lba	8 bytes	0x0	Logical Block Address
-trans_len	[0,0xFFFF]	0x0	Transfer Length in Bytes
-cort	[0,1]	0x0	Corrected bit
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.102 write_same

Command Name(s): write_same,wrs

Description: Writes one block of data to a number of logical blocks.

Default Parm Order: lba, num_blocks, wrprotect

Buffer Data Sent: 1 Blocks

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-wrprotect	[0,0x7]	0x0	EndToEnd WrProtect field
-anchor	[0,1]	0x0	Anchor bit
-unmap	[0,1]	0x0	Deallocates each LBA specified
			in command
-lba	4 bytes	0x0	Logical Block Address
-num_blocks	[0,0xFFFF]	0x1	Number of Continuous Blocks to
			be written
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.103 write_same16

Command Name(s): write_same16,wrs16

Description: Writes one block of data to a number of logical blocks.

Default Parm Order: lba, num_blocks, wrprotect

Buffer Data Sent: 1 Blocks

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-wrprotect	[0,0x7]	0x0	EndToEnd WrProtect field
-anchor	[0,1]	0x0	<No Description Available>
-unmap	[0,1]	0x0	Deallocates each LBA specified
			in command
-ndob	[0,1]	0x0	if set, no data out buffer
			should be sent with the command
-lba	8 bytes	0x0	Logical Block Address:
			previously was lbalow and
			lbahigh
-num_blocks	4 bytes	0x1	Number of Continuous Blocks to
			be written
-control_byte	[0,0xFF]	0x0	NACA | FLAG | LINK
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

C.104 write_same32

Command Name(s): write_same32,wrs32

Description: Requests the drive write data and then check it.

Default Parm Order: lba, translen

Buffer Data Sent: 1 Blocks

Buffer Data Received: *None*

Parameters:

Name	Range	Default	Description
-control_byte	[0,0xFF]	0x0	VU | Reserved | FLAG | LINK
-add_cdb_len	[0,0xFF]	0x18	Additional CDB Length
-serv_action	[0,0xFFFF]	0xD	Service Action
-wrprotect	[0,0x7]	0x0	EndToEnd WrProtect field
-anchor	[0,1]	0x0	<No Description Available>
-unmap	[0,1]	0x0	Deallocates each LBA specified
			in command
-ndob	[0,1]	0x0	if set, no data out buffer
			should be sent with the command
-lba	8 bytes	0x0	Logical Block Address
-lbr_tag_msb	[0,0xFFFF]	0xFFFF	Expected Initial Logical Block
			Reference Tag MSB
-lbr_tag_lsb	[0,0xFFFF]	0xFFFF	Expected Initial Logical Block
			Reference Tag LSB
-lba_tag	[0,0xFFFF]	0x0	Logical Block Application Tag
-lba_tag_mask	[0,0xFFFF]	0x0	Logical Block Application Tag
			Mask
-translen	4 bytes	0x1	Transfer Length in Blocks
-transport_cdb	[0,1]	<current>	Convert to 0xC3 transport cmd
-uil	[0,?]	<current>	Temporary UIL override
-dev	[0,?]	<current>	Temporary device index override
-si	[0,?]	<current>	Temporary send buffer override
-cmd_timeout	[0,?]	0	Single cmd timeout override
			(0=no override)
-set_timeout	[0,?]	0	Persistent timeout override
			(0=no override)

Appendix D
CIL Commands

D.1 ata get

Command Name(s): `ata get`

Description: This command will read and display info on the desired part of an ata device. Append ata get with one of the following commands to get the info. Possible Commands: STATUS - Status Register, ERROR - Error Register, ERRLBA - Lower 32 bits of the 48 bit LBA, ERRLBAHI - Upper 16 bits of the 48 bit LBA, ERRLBA28 - All of the 28 bit LBA, SCR - Sector Count Register, SNR - Sector Number / LBA Low Register, CLR - Cylinder Low / LBA Mid Register, CHR - Cylinder High / LBA High Register, DHR - Device Head Register, PSCR - Sector Count Register, PSNR - Sector Number / LBA Low Register, PCLR - Cylinder Low / LBA Mid Register, PCHR - Cylinder High / LBA High Register, SHADOW - Default to show it all.

Default Parm Order: `commands`

Parameters:

Name	Description
`commands`	See possible commands in the description above.

Examples:

```
ata get status
ata get error
```

D.2 buff adlerchksum

Command Name(s): `buff adlerchksum`

Description: Compute checksum of buffer data using Adler32 algorithm with base of 1.

Default Parm Order: `index offset length`

Parameters:

Name	Description
`index`	Buffer Index (Number or send or recv)
`offset`	offset to begin checksum calculation at
`length`	length of checksum calculation

Examples:

```
buff adlerchksum send 0 512
```

D.3 buff checksum

Command Name(s): `buff checksum`

Description: Computes 32-bit checksum of buffer data.

Default Parm Order: `index offset length`

Parameters:

Name	Description
`index`	Buffer Index (Number or send or recv)
`offset`	offset to begin checksum calculation at
`length`	length of checksum calculation

Examples:

```
  #calculate checksum of receive buffer
rdmx 0x1000 0xFFFF
buff checksum recv 0 0xFFFF
```

D.4 buff compare

Command Name(s): `buff compare`

Description: This function compares the contents of two buffers over a specified range for a match. If the buffers match, 0 is returned, if buffer 1 < buffer 2, -1 is returned. If buffer 1 > buffer 2, 1 is returned. This command is useful for verification operations in CSO.

Default Parm Order: `index1 index2 length ?offset1?  ?offset2?`

Parameters:

Name	Description
`index1`	First buffer index to compare
`index2`	Second buffer index to compare
`length`	Amount of buffer to compare
`offset1`	Optional: Offset into buffer 1 to begin compare (default=0)
`offset2`	Optional: Offset into buffer 2 to begin compare (default=0)

Examples:

```
buff compare 0 1 256
# Random cso with (software) data compare
proc rw_compare_cso {iterations max_tl} {
   set block_size [device info blocksize]

   for {set i 0} {$i<$iterations} {incr i} {
      #generate random data / lba
      set tl [expr int(rand() * $max_tl) + 1]
      bfr send 0 [expr $tl*$block_size]
      set lba [randlba $tl]
      # Send the data
      w10 $lba $tl
      # Read the data
      r10 $lba $tl
      # Compare the data
      if {[buff compare send recv [expr $tl*$block_size]]} {
         error "Miscompare: lba=$lba tl=$tl"
      }
   }
   puts "No miscompares detected"
}
```

D.5 buff copy

Command Name(s): `buff copy`

Description: This function can be used to copy portions of an existing buffer to another buffer. One useful application of this command is the mode sense/select combo; mode sense data can be copied from the receive buffer to the send buffer, changes can be made, then the new buffer can be "mode selected".

Default Parm Order: `source_index dest_index ?source_offset? ?dest_offset? ?length?`

Parameters:

Name	Description
`source_index`	Index of the buffer to copy from
`dest_index`	Index of the buffer to copy to
`source_offset`	Optional: Offset into from buffer
`dest_offset`	Optional: Offset into to buffer
`length`	Optional: Amount of buffer to copy (D=All)

Examples:

```
buff copy
buff copy 3 0
buff copy recv send 0x24
```

D.6 buff crc

Command Name(s): `buff crc`

Description: Computes 32-bit CRC of buffer data.

Default Parm Order: `index offset length`

Parameters:

Name	Description
`index`	Buffer Index (Number or send or recv)
`offset`	offset to begin CRC calculation at
`length`	length of CRC calculation

Examples:

```
    #calculate CRC of receive buffer
rdmx 0x1000 0xFFFF
buff crc recv 0 0xFFFF
```

D.7 buff diff

Command Name(s): `buff diff`

Description: This function diffs the contents of two buffers over a specified range for a match. In the standard implementation the information reported is: the address of the first difference and the number of bytes that are different in the file. When -v is used the same information is shown in addition to a hex dump of any two 8 byte regions that contain differences between the files (differences are surrounded by () characters). For backward-compatibility reasons, offset1 is applied to both buffers if offset2 is not specified.

Default Parm Order: `index1 index2 amount ?offset1? ?offset2? ?-b? ?-a? ?-v? ?-d? ?-dw? ?-dd? ?-pre? ?-fp? ?-f? ?-fa? ?-do?`

Parameters:

Name	Description
index1	First Index to diff
index2	Second Index to diff
amount	Amount to diff
offset1	Optional: Offset into buffer 1 to begin diff (default=0)
offset2	Optional: Offset into buffer 2 to begin diff (default=0)
-b	Optional: The number of bytes to display per line (default=8)
-a	Optional: Show ASCII
-v	Optional: Show all differences between buffers
-d	Optional: Dump entire buffer range, highlighting differences
-dw	Optional: show 16-bit differences
-dd	Optional: show 32-bit differences
-pre	Optional: string to place in front of different lines
-fp	Optional: show diffs with fewer parans
-f	Optional: write data to file (instead of returning data)
-fa	Optional: append data to file
-do	Optional: display buffer data only

Examples:

```
buff diff send recv 512 -v
```

D.8 buff dump

Command Name(s): `buff dump, bd`

Description: This function dumps the data contained in a specified buffer in hex. For the "specified buffer", either an index, "send" or "recv" can be used. Output can be formated using the several optional format flags. The column, sub, and hex width used to be set like "-column_width 16", but that has had to be changed to just the value in the correct order.

Default Parm Order: `?index? ?offset? ?length? ?column_width? ?sub_width? ?hex_width? ?-big_endian? ?-hide_ascii? ?-hide_header? ?-hide_offset? ?-hide_divider? ?-show_x? ?-no_newline? ?-dw? ?-dd? ?-qw? ?-lp? ?-op? ?-column_width? ?-sub_width? ?-hex_width? ?-be? ?-ha? ?-hh? ?-ho? ?-hd? ?-nn? ?-sx? ?-cw? ?-sw? ?-hw?`

Parameters:

Name	Description
index	Optional: Buffer Index (Number or send or recv)
offset	Optional: Offset into buffer (Default = 0)
length	Optional: Number of bytes to dump (Default = 255)
column_width	Optional: Width of hex data in bytes (Default = 16)
sub_width	Optional: Width of sub columns in bytes (Default = 8)
hex_width	Optional: Word length in bytes (Default = 1)
-big_endian	Optional: Display words in big endian form
-hide_ascii	Optional: Do not display ASCII data
-hide_header	Optional: Do not display the table header
-hide_offset	Optional: Do not display the address column
-hide_divider	Optional: Do not display the dashed divider
-show_x	Optional: Display the "0x" before all hex values
-no_newline	Optional: Do not display the last blank line
-dw	Optional: Deprecated: Output data in little endian 16-bit words
-dw	Optional: Deprecated: Output data in little endian 16-bit words
-dd	Optional: Deprecated: Output data in little endian 32-bit words
-dd	Optional: Deprecated: Output data in little endian 32-bit words
-qw	Optional: Deprecated: Output data in little endian 64-bit words
-qw	Optional: Deprecated: Output data in little endian 64-bit words
-lp	Optional: Treat length as number of pages (512 bytes)
-op	Optional: Treat length as number of pages (512 bytes)
-column_width	Optional: column_width is also the 4th optional flag
-sub_width	Optional: sub_width is also the 5th optional flag
-hex_width	Optional: hex_width is also the 6th optional flag
-be	Optional: Display words in big endian form
-ha	Optional: Do not display ASCII data
-hh	Optional: Do not display the table header
-ho	Optional: Do not display the address column
-hd	Optional: Do not display the dashed divider
-nn	Optional: Do not display the last blank line
-sx	Optional: Display the "0x" before all hex values
-cw	Optional: column_width is also the 4th optional flag
-sw	Optional: sub_width is also the 5th optional flag
-hw	Optional: hex_width is also the 6th optional flag

Examples:

```
buff dump recv
```

D.9 buff e2e

Command Name(s): `buff e2e`

Description: This command can be used to calculate the guard of end-to-end buffer data. It can also be used to set the apptag value and the reftag value. All parameters are optional and at least one parameter should be specified for the command to do something. Note that the very first time this function is called, a lookup-table is generated. Future calls simply use the lookup table, resulting in improved performance.

Default Parm Order: `?-guard? ?-offset? ?-blksize? ?-buffindex? ?-mdatabuffindex? ?-index? ?-length? ?-apptag? ?-lba? ?-check? ?-info? ?-dix?`

Parameters:

Name	Description
`-guard`	Optional: Specify this flag to calculate the guard crc value
`-guard`	Optional: Specify this flag to calculate the guard crc value
`-offset`	Optional: Block Offset into the buffer (Default = 0)
`-blksize`	Optional: Current Block Size w/o e2e Data (Default = 512)
`-buffindex`	Optional: Buffer index to use (Default = current send idx)
`-mdatabuffindex`	Optional: Buffer index to use for metadata buffer (Default = current mdata idx)
`-mdatabuffindex`	Optional: Buffer index to use for metadata buffer (Default = current mdata idx)
`-index`	Optional: DEPRECATED: Buffer Index (number, send, or recv)
`-length`	Optional: Number of blocks to calculate (Default = 1)
`-apptag`	Optional: Set the app tag value (Default = no change)
`-lba`	Optional: Set the starting reftag value (Default = no change)
`-lba`	Optional: Set the starting reftag value (Default = no change)
`-check`	Optional: Check existing data and return error if it does not match expected
`-check`	Optional: Check existing data and return error if it does not match expected
`-info`	Optional: Do not set buffer data, but return calculated guard value
`-info`	Optional: Do not set buffer data, but return calculated guard value
`-dix`	Optional: Use Data Integrity Extension (DIX - separate metadata buffer) mode
`-dix`	Optional: Use Data Integrity Extension (DIX - separate metadata buffer) mode

Examples:

```
# Prepare an e2e block to write
device set blocksize 520
bfr send 0 512
buff e2e -guard -lba 0x1000 -apptag 0
w10 0x1000 1
```

D.10 buff exists

Command Name(s): `buff exists`

Description: Checks whether a buffer exists for the given index. The command will return 1 if the buffer exists, 0 or error otherwise.

Default Parm Order: `index`

Parameters:

Name	Description
`index`	Buffer Index (number, send, or recv, or other alias)

Examples:

```
buff exists 7
```

D.11 buff fill byte

Command Name(s): `buff fill byte, bfb`

Description: This function fills a buffer with a particular byte value.

Default Parm Order: `index offset length byte`

Parameters:

Name	Description
`index`	Buffer Index (number, send, or recv)
`offset`	Offset into the buffer, in bytes
`length`	Length of pattern to write
`byte`	The byte value to write

Examples:

```
buff fill byte send 0 256 0x45
#Fill the send buffer with 5's
bfb send 0 512 5
#Fill buffer 7 with 0xAB
bfb 7 0 1024 0xAB
```

D.12 buff fill float

Command Name(s): `buff fill float, bff`

Description: This function can be used to insert a 32-bit or 64-bit float value into a buffer. This can be useful for setting up CDB parameters and other tasks.

Default Parm Order: `index offset float size ?-le?`

Parameters:

Name	Description
`index`	Buffer Index (number, send, or recv)
`offset`	Offset into the buffer, in bytes
`float`	The float value to write
`size`	The size of float to use, 4 or 8 bytes
`-le`	Optional: Use little endian mode

Examples:

```
buff fill float send 0 3.14 8 -le
```

D.13 buff fill int

Command Name(s): `buff fill int, bfi`

Description: This function can be used to insert an integer value into a buffer. This can be useful for setting up CDB parameters and for tagging write data with information (such as LBA or other information).

Default Parm Order: `index offset int ?-le?`

Parameters:

Name	Description
`index`	Buffer Index (number, send, or recv)
`offset`	Offset into the buffer, in bytes
`int`	The integer value to write
`-le`	Optional: Use little endian mode

Examples:

```
buff fill int send 12 45678 -le
# This function performs random write cso with tagged LBA
proc lba_tagged_cso {iterations} {
   do $iterations {
      # Generate random data
      bfr send 0 512
      # Get a random lba
      set lba [randlba]
      # Tag write data
      buff fill int send 0 $lba
      # Write the data
      w10 $lba
   }
}
```

D.14 buff fill int64

Command Name(s): `buff fill int64`

Description: This function can be used to insert a 64-bit value into a buffer. This can be useful for setting up CDB parameters and other tasks.

Default Parm Order: `index offset int64 ?-le?`

Parameters:

Name	Description
index	Buffer Index (number, send, or recv)
offset	Offset into the buffer, in bytes
int64	The long value to write
-le	Optional: Use little endian mode

Examples:

```
buff fill int64 send 0 0x11223344556677 -le
```

D.15 buff fill one

Command Name(s): buff fill one, bfo

Description: This commands fills a specified buffer with 0xFF bytes.

Default Parm Order: index offset length

Parameters:

Name	Description
index	Buffer Index (number, send, or recv)
offset	Offset into the buffer, in bytes
length	Length of pattern to write

Examples:

```
buff fill one send 0 512
```

D.16 buff fill patt

Command Name(s): buff fill patt, bfp

Description: This function fills a specified buffer with a pattern of bytes. The bytes should each be 0-255 (0x00-0xff) and should have 0x prepended if the intended values are in hex.

Default Parm Order: index offset length byte_list

Parameters:

Name	Description
index	Buffer Index (number, send, or recv)
offset	Offset into the buffer, in bytes
length	Length of pattern to write
byte_list	List of byte patterns

Examples:

```
buff fill patt send 0 512 0xFF 0x00
# Fill buffer 7 with the pattern 1 2 3 4 5
buff fill patt 7 0 100 1 2 3 4 5
puts [bd 7 0 100]
# Now fill in the pattern aa, bb, cc, dd in the current send buffer
# and write the data to the drive
bfp send 0 512 0xAA 0xBB 0xCC 0xDD
w10
# Look at the data
puts [r10]
```

D.17 buff fill rand

Command Name(s): `buff fill rand, bfr`

Description: This function writes random bytes of data to a specified buffer. This function is useful for setting up a buffer for random CSO operations.

Default Parm Order: `index offset length ?channel?`

Parameters:

Name	Description
`index`	Buffer Index (number, send, or recv)
`offset`	Offset into the buffer, in bytes
`length`	Length of pattern to write
`channel`	Optional: The Channel to use to seed the value

Examples:

```
buff fill rand send 0 512 10
# This function performs random write cso
proc write_cso {iterations} {
   do $iterations {
      # Generate random data
      bfr send 0 512
      # Write the data
      w10 [randlba]
   }
}
```

D.18 buff fill seq

Command Name(s): `buff fill seq, bfs`

Description: This function writes a sequence of data to the specified buffer. The command is useful for generating sequential data patterns. The minimum and maximum values for the pattern can be optionally specified. If no minimum or maximum value is specified 0-255 (0x00-0xff) is used as a default range.

Default Parm Order: `index offset length ?min?  ?max?`

Parameters:

Name	Description
`index`	Buffer Index (number, send, or recv)
`offset`	Offset into the buffer, in bytes
`length`	Length of pattern to write
`min`	Optional: The value to start the seq data at
`max`	Optional: The value to end the seq data at

Examples:

```
buff fill seq
# Write a sequence of 0-32 to the drive
buff fill seq send 0 512 0 0x20
w10
puts [r10]
# Write a sequence of 0-255 to the drive
bfs send 0 512
w10
puts [r10]
```

D.19 buff fill short

Command Name(s): `buff fill short, bfsh`

Description: This function can be used to insert a 16-bit value into a buffer. This can be useful for setting up CDB parameters and other tasks.

Default Parm Order: `index offset short ?-le?`

Parameters:

Name	Description
`index`	Buffer Index (number, send, or recv)
`offset`	Offset into the buffer, in bytes
`short`	The short value to write
`-le`	Optional: Use little endian mode

Examples:

```
buff fill short send 0 0xFFCC -le
```

D.20 buff fill string

Command Name(s): `buff fill string, bfstr`

Description: This function is used to put a string into a buffer. The command is useful for tasks such as time-date and LBA stamping. See the provided example for a demonstration of this.

Default Parm Order: `index offset string ?length?`

Parameters:

Name	Description
`index`	Buffer Index (number, send, or recv)
`offset`	Offset into the buffer, in bytes
`string`	String to write
`length`	Optional: Number of bytes to write

Examples:

```
buff fill string send 25 "Hi"
# This procedure performs random write cso with time/date stamp
proc dated_cso {iterations} {
   do $iterations {
      # Fill random data
      bfr send 0 512
      # Fill in the date
      buff fill string send 0 [clock format [clock seconds]]
      # Write the data
      w10 [randlba]
   }
}
```

D.21 buff fill zero

Command Name(s): `buff fill zero, bfz`

Description: This function provides a convenient method for zeroing out a buffer.

Default Parm Order: `index offset length`

Parameters:

Name	Description
`index`	Buffer Index (number, send, or recv)
`offset`	Offset into the buffer, in bytes
`length`	Length of pattern to write

Examples:

```
buff fill zero send 0 512
```

D.22 buff find

Command Name(s): `buff find`

Description: This function searches the specified buffer for one or more occurances of the specified data. The function will return a list of offsets, each offset indicating the starting location of the pattern match. If no matches are found, then an empty list is returned. Note that this search algorithm will find a pattern, even if it overlaps another found pattern. For example, if you search for "aaa" in the string "bbbaaaaabbb", three offsets will be returned: {3 4 5}.

Default Parm Order: `search_buff_index search_buff_offset search_buff_length data_buff_index data_buff_offset data_buff_length`

Parameters:

Name	Description
`search_buff_index`	Buffer to search in
`search_buff_offset`	Offset to start the search at
`search_buff_length`	Length of buffer to search in
`data_buff_index`	Buffer containing search data
`data_buff_offset`	Offset of the start of the search data
`data_buff_length`	Length of the search data

Examples:

```
buff find send 0 16384 recv 0 16
```

D.23 buff findstr

Command Name(s): `buff findstr`

Description: This function searches the specifies buffer for one or more occurances of the specified string. The function will return a list of offsets, each offset indicating the starting location of the pattern match. If no matches are found, then an empty list is returned. Note that this search algorithm will find a pattern, even if it overlaps another found pattern. For example, if you search for "aaa" in the string "bbbaaaaabbb", three offsets will be returned: {3 4 5}.

Default Parm Order: `search_buff_index search_buff_offset search_buff_length string`

Parameters:

Name	Description
search_buff_index	Buffer to search in
search_buff_offset	Offset to start the search at
search_buff_length	Length of buffer to search in
string	String to search for

Examples:

```
buff findstr send 0 16384 "ABC"
```

D.24 buff format

Command Name(s): buff format, bf

Description: This command is useful for extracting buffer data in a formatted way. Many CDB commands return a binary "table" of information. Information from these tables can be easily extracted using this command. This command works a lot like a scanf() in C and supports many of the same operators. The "offset" and "length" parameters correspond to the keys found in the format string and determine the offset and length of buffer data to use as input into the keys (note that the number of keys in the format string must match the number of offset/length parameters). See the example below... Important: The "offset" and "length" fields given above are NOT parsed by the buff format command. This means that when using variables, you must surround the fields with quotes and not braces (or you will get a syntax error). %L is the same as %u only it puts the value in little endian form. %h and %H work the same as %x and %X respectively only the value is in little endian form.

Default Parm Order: index fmt_str parm_list ?-flip_endian? ?-word_length?

Parameters:

Name	Description
index	Buffer Index (number, send, or recv)
fmt_str	Format String, printf style
parm_list	List of offset and length pairs
-flip_endian	Optional: Flip endianness of buffer, word aligned
-word_length	Optional: Word length in Bytes for flip_endian, defaults to 4 if not specified
-word_length	Optional: Word length in Bytes for flip_endian, defaults to 4 if not specified

Examples:

```
buff format recv "LBA: 0x%08X Xfer Len: 0x%08X\n {5 4} {9 4}
# Log sense on page 0xD
lgsns 0xD 16
puts [buff format recv "The current drive temperature is %d C" {9 1}]

# Mode sense, page 4
mdsn10 4 0x28
```

```
puts [buff format recv "This drive has %d heads and \
spins at %d rpm" "0x15 1" "36 2"]

# rdcap
rdcap
puts [buff format recv "max_lba = %d (0x%X)" {0 4} {0 4}]
```

D.25 buff get address

Command Name(s): `buff get address`

Description: Returns the address of the buffer.

Default Parm Order: `index`

Parameters:

Name	Description
`index`	Buffer Index

Examples:

```
buff get address 1
```

D.26 buff get count

Command Name(s): `buff get count`

Description: This function returns the maximum number of buffers available (the default setting is 10). This maximum can be changed with "buff set count".

Default Parm Order: `<None>`

Examples:

```
buff get count
```

D.27 buff get dsize

Command Name(s): `buff get dsize`

Description: This functions returns the minimum buffer allocation size. This parameter controls the size a buffer created at when it is first filled with data. Note that, buffers auto expand when needed so this command is not needed for functional operations. The usefulness of this command is for certain "algorithm" patterns where successively larger chunks of data are read into a buffer, requiring a lot of "costly" resizing.For most applications, this command can be safely ignored.

Default Parm Order: <None>

Examples:

```
buff get dsize
```

D.28 buff get last_si

Command Name(s): buff get last_si

Description: This function returns the index of the send buffer which was used to send the last CDB command.

Default Parm Order: <None>

Examples:

```
buff get last_si
```

D.29 buff get ri

Command Name(s): buff get ri

Description: This function returns the current receive buffer index. CDB commands that return data (such as read10) will fill this buffer index with data. Note that, when specifying indexes in commands that require them, a "recv" can be used in place of the index (see the example below). Also note that "recv" only works with Niagara specific commands. When in doubt, use "buff get ri", which always works.

Default Parm Order: <None>

Examples:

```
buff get ri
```

D.30 buff get si

Command Name(s): buff get si

Description: This function returns the current send buffer index. CDB commands that send data (such as write10) will send data from this buffer. Note that, when specifying indexes in commands that require them, a "send" can be used in place of the index. Also note that "send" only works with Niagara specific commands. When in doubt, use "buff get si", which always works.

Default Parm Order: <None>

Examples:

```
buff get si
```

D.31 buff get size

Command Name(s): `buff get size`

Description: Returns the amount of space that is allocated to a specified buffer. This is the same value that is used by buff diff when a length of 0 is specified. buff get size index.

Default Parm Order: `index`

Parameters:

Name	Description
`index`	Buffer Index

Examples:

```
  buff fill rand 3 0 0x4000;   # Fill buffer three with wore data than the default siz
buff get size 3
```

D.32 buff gets

Command Name(s): `buff gets`

Description: This function is intended to parse through a text fle loaded into a buffer. the variable varname is loaded with the contents between the offset given and the first pattern character. The value returned is the first non pattern character past the string returned. The pattern sequence is \n, \r and NULL.

Default Parm Order: `buff offset varname`

Parameters:

Name	Description
`buff`	Buffer Index (number, send, or recv)
`offset`	Offset into buffer, in bytes
`varname`	The name of the variable to store the results

Examples:

```
# Load a buffer with a file
set len [lindex [buff load test.txt] 0]

# Parse the file
set ptr 0
while {$ptr < $len} {
   set ptr [buff gets 0 $ptr line]
   puts $line
}
```

D.33 buff load

Command Name(s): `buff load`

Description: This function is used to load the contents of a (binary) file into a buffer. This function is useful in binary parsing operations, as the "buff format" command can be used to easily parse the data contained in a structured binary file (such as a saved binary logdump or a serial dump). Important note: When using a Windows(tm) system, use forward slashes instead of backward slashes. Backward slashes in TCL are interpreted as special characters (another option is to use two back slashes). Use the ASCII command to import an ASCII file in binary mode. Note that The ASCII import expect data in a format similar to that produced by the bd command (an address followed by 16 hex bytes per row). In ASCII mode, the file_offset and file_length variables represent the offset and length of the interpreted binary data, not the file itself (in strait binary the file offset and binary offsets are equivalent).

Default Parm Order: `filename ?buffer? ?buff_offset? ?file_offset? ?length? ?-ascii? ?-ascii_b?`

Parameters:

Name	Description
`filename`	Filename to load
`buffer`	Optional: Buffer to load into (Default = send)
`buff_offset`	Optional: Offset into the buffer
`file_offset`	Optional: Offset into the file
`length`	Optional: Number of bytes to read
`-ascii`	Optional: Load the data in ascii format
`-ascii_b`	Optional: Load the data in big endian format

Examples:

```
buff load "C:/temp/data.bin"
buff load "~/temp/data.txt" -ascii 7 0 0 100
```

D.34 buff peek

Command Name(s): `buff peek`

Description: This function can be used to get a byte from a buffer.

Default Parm Order: `index offset`

Parameters:

Name	Description
`index`	Buffer Index (number, send, or recv)
`offset`	Offset into the buffer, in bytes

Examples:

```
buff peek recv 13
```

D.35 buff poke

Command Name(s): `buff poke`

Description: This function can be used to insert a byte into a buffer. This can be useful for setting up CDB parameters and other tasks.

Default Parm Order: `index offset byte`

Parameters:

Name	Description
`index`	Buffer Index (number, send, or recv)
`offset`	Offset into the buffer, in bytes
`byte`	Byte to write to the buffer

Examples:

```
buff poke send 24 0x03
```

D.36 buff print sgl

Command Name(s): `buff print sgl`

Description: This command prints a representation of an SGL as specified by the parameter, buffer index.

Default Parm Order: `index`

Parameters:

Name	Description
`index`	Buffer Index

Examples:

```
buff print sgl
```

D.37 buff reset

Command Name(s): `buff reset`

Description: This function clears the contents of existing buffers. This is useful for freeing up memory resources after a memory intensive operation.

Default Parm Order: `?index?`

Parameters:

Name	Description
`index`	Optional: Buffer Index (Number or send or recv)

Examples:

```
buff reset
```

D.38 buff rsa keygen

Command Name(s): `buff rsa keygen`

Description: Create an RSA public/private key pair. These keys are used during signature creation and verification. The command returns a list of key sizes {pubKeySize privKeySize} buff rsa keygen ?random_data_buffer? ?random_data_length? ?private_key_buffer? ?public_key_buffer?.

Default Parm Order: `random_data_buffer random_data_length private_key_buffer public_key_buffer`

Parameters:

Name	Description
`random_data_buffer`	Buffer index of random data (Number or send or recv)
`random_data_length`	Length of random data
`private_key_buffer`	Buffer index for private key (Number or send or recv)
`public_key_buffer`	Buffer index for public key (Number or send or recv)

Examples:

```
buff rsa keygen recv 512 2 3
```

D.39 buff rsa sign

Command Name(s): `buff rsa sign`

Description: Create an RSA signature using the provided message data and private key. The buffer index specified for the signature buffer will contain the signature after the command returns. The command returns the length of the signature in bytes. buff rsa keygen ?message_buffer_index? ?message_length? ?random_data_buffer? ?random_data_length? ?priavte_key_buffer_index? ?signature_buffer_index?.

Default Parm Order: `message_buffer_index message_length random_data_buffer random_data_length private_key_buffer signature_buffer_index`

Parameters:

Name	Description
`message_buffer_index`	Buffer index of message data (Number or send or recv)
`message_length`	Length of random data
`random_data_buffer`	Buffer index of random data (Number or send or recv)
`random_data_length`	Length of random data
`private_key_buffer`	Buffer index of private key (Number or send or recv)
`signature_buffer_index`	Buffer index for signature (Number or send or recv)

Examples:

```
    >set codeLength [lindex [buff load c:/code/VCGNA554.bin send] 0]   #load the code in
>buff load c:/code/privateKey.txt 2                                    #load the private key
>buff fill rand recv 0 512                                             #fill recv buffer wit
>buff rsa sign send $codeLength recv 512 2 3                           #create a signature f
```

D.40 buff rsa verify

Command Name(s): `buff rsa verify`

Description: Verify an RSA signature using the provided message data, public key and signature. The command will return 1 if the signature verifies correctly, 0 or error otherwise. buff rsa verify ?message_buffer_index? ?message_length? ?public_key_buffer? ?signature_buffer? ?signature_length?.

Default Parm Order: `message_buffer_index message_length public_key_buffer signature_buffer signature_length`

Parameters:

Name	Description
message_buffer_index	Buffer index of message data (Number or send or recv)
message_length	Length of random data
public_key_buffer	Buffer index of public key (Number or send or recv)
signature_buffer	Buffer index for signature (Number or send or recv)
signature_length	Length of signature data

Examples:

```
    >set codeLength [lindex [buff load c:/code/VCGNA554.bin send] 0]   #load the code ir
>set sigLength [lindex [buff load c:/code/VCGNA554.sig recv] 0]    #load the signature i
>buff load c:/code/publicKey.txt 2                                 #load the public key
>buff rsa verify send $codeLength 2 recv $sigLength                #verify the signature
```

D.41 buff save

Command Name(s): buff save

Description: This command saves the contents of a buffer to a file. Binary data is stored. An existing file can be appended to with the -append option. If you wish to store an ASCII hex dump, use the -ascii option...

Default Parm Order: filename buffer buff_offset length ?-ascii? ?-ascii_dw? ?-ascii_dd? ?-append?

Parameters:

Name	Description
filename	Filename to save
buffer	Buffer to save from (Default = send)
buff_offset	Offset into the buffer
length	Number of bytes to write
-ascii	Optional: Save the data in ascii format
-ascii_dw	Optional: Save the data in word format
-ascii_dd	Optional: Save the data in dword format
-append	Optional: Append the data to the end of the file

Examples:

```
buff save ~/data.txt -ascii
buff save "C:/temp/data.bin send 0 0 100"
```

D.42 buff set count

Command Name(s): buff set count

Description: This function is used to change the number of available buffer indices (the default is 10). This function is useful in applications where a maximum of 10 buffers is restrictive. Note that any pre-existing buffer data is lost after this command is executed.

Default Parm Order: `count`

Parameters:

Name	Description
`count`	Maximum Number of Buffers

Examples:

```
buff set count 15
```

D.43 buff set dsize

Command Name(s): `buff set dsize`

Description: This functions sets the minimum buffer allocation size. This controls the size a buffer created at when it is first filled with data. Note that, buffers auto expand when needed so this command is not needed for functional operations. The usefulness of this command is for certain "algorithm" patterns where successively larger chunks of data are read into a buffer, requiring a lot of "costly" resizing. For most applications, this command can be safely ignored.

Default Parm Order: `size`

Parameters:

Name	Description
`size`	The Default Size of a Buffers

Examples:

```
buff set dsize 0x100000
```

D.44 buff set ri

Command Name(s): `buff set ri, bri`

Description: This function sets the current receive buffer index. CDB commands that receive data (such as read10) will read data into this buffer index. Note that, when specifying indexes in commands that require them, a "recv" can be used in place of the index. Also note that "recv" only works with Niagara specific commands. When in doubt, use "buff set ri", which always works.

Default Parm Order: `index`

Parameters:

Name	Description
`index`	Buffer index (must be a number)

Examples:

```
buff set ri 5
```

D.45 buff set sgl

Command Name(s): `buff set sgl`

Description: This procedure sets an sgl as specified by its parameter.

Default Parm Order: `data`

Parameters:

Name	Description
`data`	data

Examples:

```
buff set sgl
```

D.46 buff set si

Command Name(s): `buff set si, bsi`

Description: This function sets the current send buffer index. CDB commands that send data (such as write10) will use send data from this index. Note that, when specifying indexes in commands that require them, a "send" can be used in place of the index. Also note that "send" only works with Niagara specific commands. When in doubt, use "buff set si", which always works.

Default Parm Order: `index`

Parameters:

Name	Description
`index`	Buffer index (must be a number)

Examples:

```
buff set si 3
```

D.47 buff set size

Command Name(s): `buff set size`

Description: Allows the user to set the size of a specified buffer in memory. This is the same value that will be used by buff diff when a length of 0 is specified. Note: The buffer will still resize if more information is written to the buffer than there is space for. buff set size index size.

Default Parm Order: `index size`

Parameters:

Name	Description
`index`	Buffer Index
`size`	Buffer Size (in bytes)

Examples:

```
  buff fill rand 3 0 0x4000;   # Fill buffer three with more data than the default siz
buff set size 3 0x200
```

D.48 console_sync

Command Name(s): `console_sync`

Description: Send the current value of ::guimaker::syncConsoleGui. This value is used to determine if more than one device could be selected.

Default Parm Order: `is_synced`

Parameters:

Name	Description
`is_synced`	Is the console synchronized?

Examples:

```
console_sync 1
```

D.49 device count

Command Name(s): `device count`

Description: This function returns the number of devices connected to the current UIL driver.

Default Parm Order: `?uil_index?`

Parameters:

Name	Description
`uil_index`	Optional: UIL Index

Examples:

```
device count
```

D.50 device create

Command Name(s): `device create`

Description: This command can be used to change the initiator id of aparticular card. The command can also be used to manually set up a target. A device with the specified parameters is added to the device list. No checks are made to see if the device actually exists. Because of this, you may need to do perform a "read_capacity" and "inquiry" to update drive fields. This command is not available for all drivers.

Default Parm Order: `channel host_id target_id lun`

Parameters:

Name	Description
`channel`	ID of card
`host_id`	Desired Initiator ID
`target_id`	Target ID
`lun`	LUN

Examples:

```
# Create a device on the first card with host=7, target=0, lun=0
device create 0 7 0 0
# Select our new target
dsi [expr [device count]-1]
# Update inquiry and capacity information
inq
rdcap
```

D.51 device get allow_set_when_locked

Command Name(s): `device get allow_set_when_locked`

Description: This function returns the current targe allow_set_when_locked flag.

Default Parm Order: `<None>`

Examples:

```
device get allow_set_when_locked
```

D.52 device get callback create

Command Name(s): `device get callback create`

Description: This function returns the callback mapped to a specific device command. Callbacks are code segments that are automatically called whenever a command is executed. Currently supported callbacks are lock, unlock, rescan, create, remove and "set index".

Default Parm Order: `<None>`

Examples:

```
device get callback lock
```

D.53 device get callback lock

Command Name(s): `device get callback lock`

Description: This function returns the callback mapped to a specific device command. Callbacks are code segments that are automatically called whenever a command is executed. Currently supported callbacks are lock, unlock, rescan, create, remove and "set index".

Default Parm Order: `<None>`

Examples:

```
device get callback lock
```

D.54 device get callback remove

Command Name(s): `device get callback remove`

Description: This function returns the callback mapped to a specific device command. Callbacks are code segments that are automatically called whenever a command is executed. Currently supported callbacks are lock, unlock, rescan, create, remove and "set index".

Default Parm Order: `<None>`

Examples:

```
device get callback lock
```

D.55 device get callback rescan

Command Name(s): `device get callback rescan`

Description: This function returns the callback mapped to a specific device command. Callbacks are code segments that are automatically called whenever a command is executed. Currently supported callbacks are lock, unlock, rescan, create, remove and "set index".

Default Parm Order: `<None>`

Examples:

```
device get callback lock
```

D.56 device get callback "set index"

Command Name(s): `device get callback "set index"`

Description: This function returns the callback mapped to a specific device command. Callbacks are code segments that are automatically called whenever a command is executed. Currently supported callbacks are lock, unlock, rescan, create, remove and "set index".

Default Parm Order: `<None>`

Examples:

```
device get callback lock
```

D.57 device get callback unlock

Command Name(s): `device get callback unlock`

Description: This function returns the callback mapped to a specific device command. Callbacks are code segments that are automatically called whenever a command is executed. Currently supported callbacks are lock, unlock, rescan, create, remove and "set index".

Default Parm Order: `<None>`

Examples:

```
device get callback lock
```

D.58 device get index

Command Name(s): `device get index`

Description: This function returns the current target index. Because FCAL/SCSI are not the only potential interfaces available to Niagara, devices on the loop/bus are enumerated. This function returns the index of the currently selected device.

Default Parm Order: `?uil_index?`

Parameters:

Name	Description
`uil_index`	Optional: UIL Index

Examples:

```
device get index
```

D.59 device get interface

Command Name(s): `device get interface`

Description: This function returns the interface type (SCSI/FCAL/SAS/etc) of the current device, if the driver has knowledge of this information. In other cases Unsupported Feature is returned.

Default Parm Order: `<None>`

Examples:

```
device get interface
```

D.60 device get last_cmd

Command Name(s): `device get last_cmd`

Description: This function returns the last CDB/ATA command to be executed. If the last CDB/ATA command execution attempt failed due to a syntax error, a partially formed CDB/ATA command may be returned.

Default Parm Order: `<None>`

Examples:

```
device get last_cmd
```

D.61 device get last_cmd_time

Command Name(s): `device get last_cmd_time`

Description: Returns the command execution time for the last cmd in microseconds.

Default Parm Order: <None>

Examples:

```
device get last_cmd_time
```

D.62 device get read_xfer

Command Name(s): device get read_xfer

Description: Returns the current state of read xfers. One means that buffer transfers on read are enabled. Zero means transfers are disabled. If read_xfers are disabled, you cannot count on data being in the buffer after a transfer from target to host. This command is useful when performance is more important than data verification.

Default Parm Order: <None>

Examples:

```
# Check to see if read xfer is enabled
if {[device get read_xfer]} {
   puts "read xfers are enabled"
}
```

D.63 device get receive_count

Command Name(s): device get receive_count

Description: Certain commands, such as inquiry or mode_sense, may return fewer bytes back than were asked for. This function can be used to determine the number of data bytes that we actually sent from the target to the host. This feature is not available in all drivers.

Default Parm Order: <None>

Examples:

```
# Perform a command
inq -pagecode 0 -alloc 255
# Get the receive count
puts "We asked for 255 bytes, inquiry actually returned [device get receive_count]"
```

D.64 device get reserved

Command Name(s): device get reserved

Description: This command will get the reserved status of a device on a given interface at a given index. If the device is reserved nothing other than Niagara can send I/O requests to the device. If no index is given then the current device will be used.

Default Parm Order: `?index?`

Parameters:

Name	Description
`index`	Optional: The index to reserve

Examples:

```
device get reserved 1
```

D.65 device get send_count

Command Name(s): `device get send_count`

Description: This function returns the number of bytes actually requested by the device for the last command. This feature is not available in all drivers.

Default Parm Order: `<None>`

Examples:

```
device get send_count
```

D.66 device get timeout

Command Name(s): `device get timeout`

Description: This function returns the timeout value for the current device in milliseconds.

Default Parm Order: `?-override_persistent?`

Parameters:

Name	Description
`-override_persistent`	Optional: The timeout currently overriding persistent timeouts. O is disabled
`-override_persistent`	Optional: The timeout currently overriding persistent timeouts. O is disabled

Examples:

```
puts "The current device has a timout of [device get timeout] ms"
```

D.67 device get xfer_mode

Command Name(s): `device get xfer_mode`

Description: Returns the current transfer mode for the current driver. Possible modes are: normal, hc, copy, random, random_hc, random_seed, random_seed_keyed, keyed, keyed_hc, inc, inc_hc, repeat, repeat_hc.

Default Parm Order: `<None>`

Examples:

```
# Get the current xfer mode
device get xfer_mode
```

D.68 device hbareset

Command Name(s): `device hbareset`

Description: This function resets the device driver and rescans the interface of the current UIL driver for new/removed devices. Note that not all UIL drivers support hbaresets.

Default Parm Order: `<None>`

Examples:

```
device hbareset
```

D.69 device info

Command Name(s): `device info`

Description: This function returns the following information about a device: VendorID, Serial#, CodeLevel, HostID, SCSIChannel, DeviceID, LUN, SCSIID, Protection, ProtectionType, Phy_BlockSize, Blocksize, MaxLBA, and ProductID. This information can be requested individually as shown in the example section. Performing an inquiry updates the serial number, code level, and vendor id (This is useful after a code download). Performing a rdcap updates the blocksize and maxlba fields (This is useful after changing the blocksize via a format).

Default Parm Order: `?device_index?  ?uil_index?  ?-prev?`

Parameters:

Name	Description
`device_index`	Optional: Device Index
`uil_index`	Optional: UIL Index
`-prev`	Optional: Info from the dev selected before the last rescan

Examples:

```
device info
device info vendor //VendorID
device info serial //Serial#
device info codelevel //CodeLevel
device info host //HostID
device info channel //SCSIChannel
device info target //DeviceID
device info lun //LUN
device info scsiid //SCSIID
device info protection //Protection
device info protection_type //ProtectionType
device info phy_blocksize //Phy_Blocksize
device info blocksize //BlockSize
device info maxlba //MaxLBA
device info prodctid //ProductID
device info peripheral //PeripheralType
```

D.70 device info blocksize

Command Name(s): `device info blocksize`

Description: This function returns the current blocksize for a device. This function does not issue a read_capacity but instead relies on a previous call to read_capacity for the information. It is a good idea to execute a read_capacity after formatting a device to a different blocksize to update this field.

Default Parm Order: `?device_index?  ?uil_index?  ?-prev?`

Parameters:

Name	Description
`device_index`	Optional: Device Index
`uil_index`	Optional: UIL Index
`-prev`	Optional: Blocksize from the dev selected before the last rescan
`-prev`	Optional: Blocksize from the dev selected before the last rescan

Examples:

```
device info blocksize
```

D.71 device info channel

Command Name(s): `device info channel`

Description: This function returns the channel id for a device.

Default Parm Order: `?device_index?  ?uil_index?  ?-prev?`

Parameters:

Name	Description
device_index	Optional: Device Index
uil_index	Optional: UIL Index
-prev	Optional: Info from the dev selected before the last rescan

Examples:

```
device info channel
```

D.72 device info codelevel

Command Name(s): device info codelevel

Description: This function returns the code level for a device. This function does not perform an inquiry but instead relies on a past inquiry for the information. Performing an appropriate inquiry will automatically update the information. Is is generally a good idea to execute an inquiry command after performing a download and save operation to update this field.

Default Parm Order: ?device_index? ?uil_index? ?-full? ?-prev?

Parameters:

Name	Description
device_index	Optional: Device Index
uil_index	Optional: UIL Index
-full	Optional: Full Codelevel
-prev	Optional: Info from the dev selected before the last rescan

Examples:

```
device info codelevel
```

D.73 device info host

Command Name(s): device info host

Description: This function returns the host id of the current device.

Default Parm Order: ?device_index? ?uil_index? ?-prev?

Parameters:

Name	Description
device_index	Optional: Device Index
uil_index	Optional: UIL Index
-prev	Optional: Info from the dev selected before the last rescan

Examples:

```
device info host
```

D.74 device info inq_pages

Command Name(s): `device info inq_pages`

Description: This function returns a dictionary of inquiry pages and corresponding values of 1/0 to show if they are valid for a given device.

Default Parm Order: `?device_index?  ?uil_index?  ?-prev?  ?-valid?`

Parameters:

Name	Description
`device_index`	Optional: Device Index
`uil_index`	Optional: UIL Index
`-prev`	Optional: Info from the dev selected before the last rescan
`-valid`	Optional: Return whether the inq_pages have been set instead of the inq_pages dictionary
`-valid`	Optional: Return whether the inq_pages have been set instead of the inq_pages dictionary

Examples:

```
device info inq_pages
```

D.75 device info lun

Command Name(s): `device info lun`

Description: This function returns the LUN of the currently selected device.

Default Parm Order: `?device_index?  ?uil_index?  ?-prev?`

Parameters:

Name	Description
`device_index`	Optional: Device Index
`uil_index`	Optional: UIL Index
`-prev`	Optional: Info from the dev selected before the last rescan

Examples:

```
device info lun
```

D.76 device info markersize

Command Name(s): `device info markersize`

Description: This function returns the stored HA Marker Size (MRKSZ) for the selected device.

Default Parm Order: `?device_index?  ?uil_index?  ?-prev?`

Parameters:

Name	Description
`device_index`	Optional: Device Index
`uil_index`	Optional: UIL Index
`-prev`	Optional: Info from the dev selected before the last rescan

Examples:

```
device info markersize
```

D.77 device info maxlba

Command Name(s): `device info maxlba`

Description: This function returns the stored maxlba for the current device. Note that this function does not perform a read_capacity, therefore it is possible for the results of this command to be reformatted. Every time an appropriate read_capacity is executed, however this field is automatically updated. If Niagara was started with a drive spun down, this command may return a zero. Executing a read_capacity will correct the value automatically.

Default Parm Order: `?device_index?  ?uil_index?  ?-prev?`

Parameters:

Name	Description
`device_index`	Optional: Device Index
`uil_index`	Optional: UIL Index
`-prev`	Optional: Info from the dev selected before the last rescan

Examples:

```
device info maxlba
```

D.78 device info mdata_inline

Command Name(s): `device info mdata_inline`

Description: This function returns whether the device has inline metadata.

Default Parm Order: `?device_index?  ?uil_index?  ?-prev?`

Parameters:

Name	Description
`device_index`	Optional: Device Index
`uil_index`	Optional: UIL Index
`-prev`	Optional: Info from the dev selected before the last rescan

Examples:

```
device info mdata_inline
```

D.79 device info mdata_size

Command Name(s): `device info mdata_size`

Description: This function returns the metadata size of a device.

Default Parm Order: `?device_index?  ?uil_index?  ?-prev?`

Parameters:

Name	Description
`device_index`	Optional: Device Index
`uil_index`	Optional: UIL Index
`-prev`	Optional: Info from the dev selected before the last rescan

Examples:

```
device info mdata_size
```

D.80 device info name

Command Name(s): `device info name`

Description: This function returns the name of the currently selected device.

Default Parm Order: `?device_index?  ?uil_index?  ?-prev?`

Parameters:

Name	Description
device_index	Optional: Device Index
uil_index	Optional: UIL Index
-prev	Optional: Name from the dev selected before the last rescan

Examples:

```
device info name
```

D.81 device info peripheral

Command Name(s): device info peripheral

Description: This function returns the peripheral type of a device. This function does not issue an inquiry for the information but instead relies on information stored from a previous inquiry for the information.

Default Parm Order: ?device_index? ?uil_index? ?-prev?

Parameters:

Name	Description
device_index	Optional: Device Index
uil_index	Optional: UIL Index
-prev	Optional: Info from the dev selected before the last rescan

Examples:

```
device info peripheral
```

D.82 device info phy_blocksize

Command Name(s): device info phy_blocksize

Description: This function returns the current physical blocksize for a device. This function does not issue a read_capacity16 but instead relies on a previous call to read_capacity16 for the information. It is a good idea to execute a read_capacity16 after formatting a device to a different blocksize to update this field.

Default Parm Order: ?device_index? ?uil_index? ?-prev?

Parameters:

Name	Description
device_index	Optional: Device Index
uil_index	Optional: UIL Index
-prev	Optional: PhyBlocksize from the dev selected before the last rescan
-prev	Optional: PhyBlocksize from the dev selected before the last rescan

Examples:

```
device info physical_blocksize
```

D.83 device info productid

Command Name(s): `device info productid`

Description: This function returns the stored product id for a device. Note that this command does not perform an inquiry for the information but instead depends on a inquiry that was called when the CIL was started. Performing an inquiry at any time will update the information returned by this command. Setting the device_index to -1 will return the product id for the currently selected device on the UIL specified.

Default Parm Order: `?device_index? ?uil_index? ?-dev? ?-uil? ?-prev?`

Parameters:

Name	Description
`device_index`	Optional: Device Index
`uil_index`	Optional: UIL Index
`-dev`	Optional: Device Index - This is also the 1st optional flag
`-uil`	Optional: UIL Index - This is also the 2nd optional flag
`-prev`	Optional: Info from the dev selected before the last rescan

Examples:

```
device info productid
```

D.84 device info project

Command Name(s): `device info project`

Description: This function returns the project of the currently selected device.

Default Parm Order: `?device_index? ?uil_index? ?-prev?`

Parameters:

Name	Description
`device_index`	Optional: Device Index
`uil_index`	Optional: UIL Index
`-prev`	Optional: Info from the dev selected before the last rescan

Examples:

```
device info project
```

D.85 device info protection

Command Name(s): `device info protection`

Description: This function returns true if end to end protection is enabled for the currently selected device.

Default Parm Order: `?device_index?  ?uil_index?  ?-prev?`

Parameters:

Name	Description
`device_index`	Optional: Device Index
`uil_index`	Optional: UIL Index
`-prev`	Optional: Info from the dev selected before the last rescan

Examples:

```
device info protection
```

D.86 device info protection_location

Command Name(s): `device info protection_location`

Description: This function returns the protection location of a device.

Default Parm Order: `?device_index?  ?uil_index?  ?-prev?`

Parameters:

Name	Description
`device_index`	Optional: Device Index
`uil_index`	Optional: UIL Index
`-prev`	Optional: Info from the dev selected before the last rescan

Examples:

```
device info protect_location
```

D.87 device info protection_type

Command Name(s): `device info protection_type`

Description: Returns the protection type for a device. This is the protection type value defined in the SCSI spec. It is equal to the value returned by rdcap16 + 1.

Default Parm Order: `?device_index?  ?uil_index?  ?-prev?`

Parameters:

Name	Description
`device_index`	Optional: Device Index
`uil_index`	Optional: UIL Index
`-prev`	Optional: Info from the dev selected before the last rescan

Examples:

```
device info protection_type
```

D.88 device info protocol

Command Name(s): `device info protocol`

Description: This function returns the protocol of a device (scsi or ata). This function does not issue an inquiry for the information but instead relies on information stored from a previous inquiry for the information.

Default Parm Order: `?device_index?  ?uil_index?  ?-prev?`

Parameters:

Name	Description
`device_index`	Optional: Device Index
`uil_index`	Optional: UIL Index
`-prev`	Optional: Info from the dev selected before the last rescan

Examples:

```
device info protocol
```

D.89 device info rto

Command Name(s): `device info rto`

Description: This function returns true if reference tag own is enabled for the currently selected device.

Default Parm Order: `?device_index?  ?uil_index?  ?-prev?`

Parameters:

Name	Description
`device_index`	Optional: Device Index
`uil_index`	Optional: UIL Index
`-prev`	Optional: Info from the dev selected before the last rescan

Examples:

```
device info rto
```

D.90 device info scsiid

Command Name(s): `device info scsiid`

Description: This function returns the SCSIID of the currently selected device.

Default Parm Order: `?device_index?  ?uil_index?  ?-prev?`

Parameters:

Name	Description
`device_index`	Optional: Device Index
`uil_index`	Optional: UIL Index
`-prev`	Optional: Info from the dev selected before the last rescan

Examples:

```
device info scsiid
```

D.91 device info serial

Command Name(s): `device info serial`

Description: This function returns the stored serial number for a device. Note that this command does not perform an inquiry for the information but instead depends on a inquiry that was called when the CIL was started. Performing an inquiry at any time will update the information returned by this command. Setting the device_index to -1 will return the serial for the currently selected device on the UIL specified.

Default Parm Order: `?device_index?  ?uil_index?  ?-dev?  ?-uil?  ?-prev?`

Parameters:

Name	Description
`device_index`	Optional: Device Index
`uil_index`	Optional: UIL Index
`-dev`	Optional: Device Index - This is also the 1st optional flag
`-uil`	Optional: UIL Index - This is also the 2nd optional flag
`-prev`	Optional: Info from the dev selected before the last rescan

Examples:

```
device info serial
```

D.92 device info serial_asic_version

Command Name(s): `device info serial_asic_version`

Description: This function returns the ASIC version of a device connected over serial.

Default Parm Order: `?device_index?  ?uil_index?  ?-prev?`

Parameters:

Name	Description
`device_index`	Optional: Device Index
`uil_index`	Optional: UIL Index
`-prev`	Optional: Info from the dev selected before the last rescan

Examples:

```
device info serial_asic_version
```

D.93 device info target

Command Name(s): `device info target`

Description: This function returns the target ID of the current device.

Default Parm Order: `?device_index?  ?uil_index?  ?-prev?`

Parameters:

Name	Description
`device_index`	Optional: Device Index
`uil_index`	Optional: UIL Index
`-prev`	Optional: Info from the dev selected before the last rescan

Examples:

```
device info target
```

D.94 device info vendor

Command Name(s): `device info vendor`

Description: This function returns the vendor id for a device. This function does not issue an inquiry for the information but instead relies on information stored from a previous inquiry for the information.

Default Parm Order: `?device_index?  ?uil_index?  ?-prev?`

Parameters:

Name	Description
`device_index`	Optional: Device Index
`uil_index`	Optional: UIL Index
`-prev`	Optional: Info from the dev selected before the last rescan

Examples:

```
device info vendor
```

D.95 device info wwid

Command Name(s): `device info wwid`

Description: This function returns the stored wwid for a device. Note that this command does not perform an inquiry for the information but instead depends on a inquiry that was called when the CIL was started. Performing an inquiry at any time will update the information returned by this command. Setting the device_index to -1 will return the WWID for the currently selected device on the UIL specified.

Default Parm Order: `?device_index?  ?uil_index?  ?-dev?  ?-uil?  ?-prev?`

Parameters:

Name	Description
`device_index`	Optional: Device Index
`uil_index`	Optional: UIL Index
`-dev`	Optional: Device Index - This is also the 1st optional flag
`-uil`	Optional: UIL Index - This is also the 2nd optional flag
`-prev`	Optional: Info from the dev selected before the last rescan

Examples:

```
device info wwid
```

D.96 device islocked

Command Name(s): `device islocked`

Description: This function returns 1 if a device is locked, 0 otherwise. A locked device returns a "Device Locked" error to any commands directed at it.

Default Parm Order: `index ?uil_index?`

Parameters:

Name	Description
index	The index of the drive to query
uil_index	Optional: The index of the uil to use (if not the current)

Examples:

```
# See what devices are locked
for {set i 0} {$i < [device count]} {incr i} {
   if {[device islocked $i]} {
      puts "device $i, is locked"
   } else {
      puts "device $i, is unlocked"
   }
}
```

D.97 device list

Command Name(s): `device list`

Description: This function returns a table of devices connected to the current uil. This table contains the following information: index of device, vendor id,host id, card id, target id, LUN, maxlba of device, and the blocksize of the device.

Default Parm Order: `?uil_index?`

Parameters:

Name	Description
uil_index	Optional: The index of the uil to use (if not the current)

Examples:

```
device list
```

D.98 device lock

Command Name(s): `device lock`

Description: This function locks a device. A locked device return an error if any commands are sent to it. This function is useful for protecting an internal drive or other non-testing device from accidental damage.

Default Parm Order: `index ?uil_index?`

Parameters:

Name	Description
index	The device index to lock
uil_index	Optional: The uil index to lock the device on

Examples:

```
# Lock device 0
device lock 0
```

D.99 device lock serial

Command Name(s): `device lock serial`

Description: Locks a device based on the device serial number.

Default Parm Order: `serial_num ?uil_index?`

Parameters:

Name	Description
serial_num	The serial number of the drive to lock
uil_index	Optional: The uil that the device is on (if not the current one)

Examples:

```
device lock serial "HUA1234414"
```

D.100 device remove

Command Name(s): `device remove`

Description: This command removes a device from the device list. If the current device index is greater than or equal to the removed device, it is decremented automatically. This command is not available for all drivers.

Default Parm Order: `index`

Parameters:

Name	Description
index	Index of device to remove

Examples:

```
# Remove device 0 from the list
device remove 0
```

D.101 device rescan

Command Name(s): `device rescan`

Description: This function rescans the interface of the current UIL driver for new/removed devices. Note that not all UIL drivers support device rescans.

Default Parm Order: `?uil_index?`

Parameters:

Name	Description
`uil_index`	Optional: The uil index to lock the device on

Examples:

```
device rescan
```

D.102 device set allow_set_when_locked

Command Name(s): `device set allow_set_when_locked`

Description: Default is device is allowed to be set when locked.

Default Parm Order: `on/off`

Parameters:

Name	Description
`on/off`	1 to allow device to be set when locked, 0 to disallow

Examples:

```
device set allow_set_when_locked 0
```

D.103 device set blocksize

Command Name(s): `device set blocksize`

Description: This function sets the blocksize for the device. Normally this should be done using a read_capacity cdb (which sets the parameter automatically). This command is provided for cases where the blocksize needs to be explicitly set (such as during drive bringup).

Default Parm Order: `blocksize`

Parameters:

Name	Description
`blocksize`	The blocksize to set the device to

Examples:

```
device set blocksize 512
```

D.104 device set callback create

Command Name(s): `device set callback create`

Description: This function is used to set a callback for a particular command. Each time the command is called, this callback is executed as code. Currently supported callbacks are "lock", "unlock", "rescan", "create", "remove", and "set index". Each of these callbacks correspond to the associated device command. Note that in the cases of lock, unlock, create, remove, and set index, the variable device_index and uil_index are set to provide further information within the callback (rescan only sets uil_index). Also note that it is generally a good idea to make sure a callback is unset before setting it yourself. To remove a callback, set it to "".

Default Parm Order: `callback`

Parameters:

Name	Description
`callback`	Code to execute

Examples:

```
device set callback lock {puts "device $device_index" is locked }
device lock 0
# Remove the callback
device set callback lock ""
```

D.105 device set callback lock

Command Name(s): `device set callback lock`

Description: This function is used to set a callback for a particular command. Each time the command is called, this callback is executed as code. Currently supported callbacks are "lock", "unlock", "rescan", "create", "remove", and "set index". Each of these callbacks correspond to the associated device command. Note that in the cases of lock, unlock, create, remove, and set index, the variable device_index and uil_index are set to provide further information within the callback (rescan only sets uil_index). Also note that it is generally a good idea to make sure a callback is unset before setting it yourself. To remove a callback, set it to "".

Default Parm Order: `callback`

Parameters:

Name	Description
`callback`	Code to execute

Examples:

```
device set callback lock {puts "device $device_index" is locked }
device lock 0
# Remove the callback
device set callback lock ""
```

D.106 device set callback remove

Command Name(s): `device set callback remove`

Description: This function is used to set a callback for a particular command. Each time the command is called, this callback is executed as code. Currently supported callbacks are "lock", "unlock", "rescan", "create", "remove", and "set index". Each of these callbacks correspond to the associated device command. Note that in the cases of lock, unlock, create, remove, and set index, the variable device_index and uil_index are set to provide further information within the callback (rescan only sets uil_index). Also note that it is generally a good idea to make sure a callback is unset before setting it yourself. To remove a callback, set it to "".

Default Parm Order: `callback`

Parameters:

Name	Description
`callback`	Code to execute

Examples:

```
device set callback lock {puts "device $device_index" is locked }
device lock 0
# Remove the callback
device set callback lock ""
```

D.107 device set callback rescan

Command Name(s): `device set callback rescan`

Description: This function is used to set a callback for a particular command. Each time the command is called, this callback is executed as code. Currently supported callbacks are "lock", "unlock", "rescan", "create", "remove", and "set index". Each of these callbacks correspond to the associated device command. Note that in the cases of lock, unlock, create, remove, and set index, the variable device_index and uil_index are set to provide further information within the callback (rescan only sets uil_index). Also note that it is generally a good idea to make sure a callback is unset before setting it yourself. To remove a callback, set it to "".

Default Parm Order: `callback`

Parameters:

Name	Description
`callback`	Code to execute

Examples:

```
device set callback lock {puts "device $device_index" is locked }
device lock 0
# Remove the callback
device set callback lock ""
```

D.108 device set callback "set index"

Command Name(s): `device set callback "set index"`

Description: This function is used to set a callback for a particular command. Each time the command is called, this callback is executed as code. Currently supported callbacks are "lock", "unlock", "rescan", "create", "remove", and "set index". Each of these callbacks correspond to the associated device command. Note that in the cases of lock, unlock, create, remove, and set index, the variable device_index and uil_index are set to provide further information within the callback (rescan only sets uil_index). Also note that it is generally a good idea to make sure a callback is unset before setting it yourself. To remove a callback, set it to "".

Default Parm Order: `callback`

Parameters:

Name	Description
`callback`	Code to execute

Examples:

```
device set callback lock {puts "device $device_index" is locked }
device lock 0
# Remove the callback
device set callback lock ""
```

D.109 device set callback unlock

Command Name(s): `device set callback unlock`

Description: This function is used to set a callback for a particular command. Each time the command is called, this callback is executed as code. Currently supported callbacks are "lock", "unlock", "rescan", "create",

"remove", and "set index". Each of these callbacks correspond to the associated device command. Note that in the cases of lock, unlock, create, remove, and set index, the variable device_index and uil_index are set to provide further information within the callback (rescan only sets uil_index). Also note that it is generally a good idea to make sure a callback is unset before setting it yourself. To remove a callback, set it to "".

Default Parm Order: `callback`

Parameters:

Name	Description
`callback`	Code to execute

Examples:

```
device set callback lock {puts "device $device_index" is locked }
device lock 0
# Remove the callback
device set callback lock ""
```

D.110 device set index

Command Name(s): `device set index, dsi`

Description: Each UIL object can generally access one or more devices. This function selects a device for a particular uil. Note that a given UIL instance "remembers" its current device index so this command only applies to the current uil. This command is generally used to communicate with multiple drives on a single bus/loop.

Default Parm Order: `index ?uil?`

Parameters:

Name	Description
`index`	Index of Device
`uil`	Optional: UIL to use

Examples:

```
# Random read cso on random drives on bus/loop
proc rand_cso_loop {iterations} {
   set dc [device count]
   for {set i 0} {$i < $iterations} {incr i} {
      dsi [expr int(rand() * $dc)]
      r10 [randlba]
   }
}
```

D.111 device set markersize

Command Name(s): `device set markersize`

Description: This function sets the HA Marker Size (MRKSZ) for the selected device. NOTE:This command will overwrite the Niagara internal markersize value, but will not change the value on the actual device (as it is read only).

Default Parm Order: `markersize`

Parameters:

Name	Description
`markersize`	Marker size in bytes

Examples:

```
device set markersize 3000000
```

D.112 device set maxlba

Command Name(s): `device set maxlba`

Description: This function sets the maximum LBA for the device. Normally this should be done using a read_capacity cdb (which sets the parameter automatically). This command is provided for cases where the max lba needs to be explicitly set (such as during drive bringup).

Default Parm Order: `maxlba`

Parameters:

Name	Description
`maxlba`	Maximum LBA in blocks

Examples:

```
device set maxlba 3000000
```

D.113 device set phy_blocksize

Command Name(s): `device set phy_blocksize`

Description: This function sets the physical blocksize for the device. Normally this should be done using a read_capacity cdb (which sets the parameter automatically). This command is provided for cases where the physical blocksize needs to be explicitly set (such as during drive bringup).

Default Parm Order: `phy_blocksize`

Parameters:

Name	Description
`phy_blocksize`	The physical blocksize to set the device to

Examples:

```
device set phy_blocksize 0x1000
```

D.114 device set project

Command Name(s): `device set project`

Description: This function will set a new project id for the drive.

Default Parm Order: `project`

Parameters:

Name	Description
`project`	The name of the project id

Examples:

```
device set project ccb7
```

D.115 device set protocol

Command Name(s): `device set protocol`

Description: This function sets the current device's protocol. This is something that is set during a device rescan. This is useful if you want certain tools to act a specific way depending on the device's protocol. SuperCSO is an example of that.

Default Parm Order: `device_protocol`

Parameters:

Name	Description
`device_protocol`	The protocol to set the device to: scsi, ata, ahci, or unknown
`device_protocol`	The protocol to set the device to: scsi, ata, ahci, or unknown

Examples:

```
device set protocol ata
```

D.116 device set read_xfer

Command Name(s): `device set read_xfer`

Description: This command (might) disable transfers to the current receive buffer. The intent of this function is to allow application that are only concerned with maximum speed to disable the transfers in exchange for faster command execution. Note that not all drivers support this function.

Default Parm Order: `read_xfer_state`

Parameters:

Name	Description
`read_xfer_state`	Set to one if on, 0 if off

Examples:

```
proc test_xfer {iterations tl} {
# A quick test to see if read_xfer is supported
# for the current driver
if {[catch {device set read_xfer 0}]} {
   error "read_xfer not supported by this driver"
}

set max [expr $iterations*$tl]

# Now for a speed test
device set read_xfer 1
set t1 [clock seconds]
for {set i 0} {$i < $max} {incr i $tl} { r10 $i $tl }
set t2 [clock seconds]
puts "$iterations reads with read xfer on: [expr $t2-$t1] sec"

device set read_xfer 0
set t1 [clock seconds]
for {set i 0} {$i < $max} {incr i $tl} { r10 $i $tl }
set t2 [clock seconds]
puts "$iterations reads with read xfer off: [expr $t2-$t1] sec"
}
```

D.117 device set reserved

Command Name(s): `device set reserved`

Description: This command will get the reserved status of a device on a given interface at a given index. If the device is reserved nothing other than Niagara can send I/O requests to the device. If no index is given then the current device will be used.

Default Parm Order: `reserve ?index?`

Parameters:

Name	Description
`reserve`	Bool used to reserve or release the device
`index`	Optional: The index to reserve

Examples:

```
device set reserved 1 1
```

D.118 device set serial

Command Name(s): `device set serial`

Description: This function will set a new serial number for the drive.

Default Parm Order: `serial`

Parameters:

Name	Description
`serial`	The Serial Number in which you would like to set to the current drive

Examples:

```
device set serial_num SN123456
```

D.119 device set timeout

Command Name(s): `device set timeout`

Description: This function sets a command timeout value for devices. Note that different UIL drivers support different units for timeout values, not all allow timing to millisecond accuracy. Therefore after setting a timeout, the actual timeout set is returned.

Default Parm Order: `time ?-override_persistent?`

Parameters:

Name	Description
`time`	Timeout in milliseconds
`-override_persistent`	Optional: Temporarily override the persistent timeouts set for each command. Use 0 to disable.
`-override_persistent`	Optional: Temporarily override the persistent timeouts set for each command. Use 0 to disable.

Examples:

```
# Set device timeout for 2 seconds
device set timeout 2000
```

D.120 device set xfer_mode

Command Name(s): `device set xfer_mode`

Description: Sets the transfer mode for the current driver, return error if not supported, 0 if hardware implemented, 1 if software emulated. Available modes are: normal, hc, copy, random, random_hc, random_seed, random_seed_keyed, keyed, keyed_hc, inc, inc_hc, repeat, repeat_hc, or repeat_read_hc. Note that for most drivers (i-Tech), only read/write commands generate data from hardware.

Default Parm Order: `mode ?seed?`

Parameters:

Name	Description
`mode`	The string representation of the mode
`seed`	Optional: Random Seek

Examples:

```
# Set the current xfer mode to random
device set xfer_mode random
w10
```

D.121 device unlock

Command Name(s): `device unlock`

Description: This command unlocks a device. A locked device returns a "Device Locked" error if any CDBs or other commands are sent to it.

Default Parm Order: `index ?uil_index?`

Parameters:

Name	Description
`index`	The index of the device to unlock
`uil_index`	Optional: The uil index to lock the device on

Examples:

```
# Unlock device 0
device unlock 0
```

D.122 device unlock serial

Command Name(s): `device unlock serial`

Description: Unlocks a device based on the drive serial number.

Default Parm Order: `serial_num ?uil_index?`

Parameters:

Name	Description
`serial_num`	The serial number of the drive to unlock
`uil_index`	Optional: The uil that the device is on (if not the current one)

Examples:

```
device unlock serial "HUA123545"
```

D.123 disable_niagara_log

Command Name(s): `disable_niagara_log`

Description: This function will disable outputing to Niagara_Log.txt.

Default Parm Order: `<None>`

Examples:

```
disable_niagara_log
```

D.124 enable_niagara_log

Command Name(s): `enable_niagara_log`

Description: This function will enable outputing to Niagara_Log.txt.

Default Parm Order: `<None>`

Examples:

```
enable_niagara_log
```

D.125 encode

Command Name(s): `encode`

Description: This function encrypts a TCL file, producing a <filename>.stc file. Once encrypted, the file cannot be viewed or edited. The file can be executed with the esource command, however.

Default Parm Order: `tcl_filename`

Parameters:

Name	Description
`tcl_filename`	tcl filename

Examples:

```
encode
```

D.126 eparse

Command Name(s): `eparse`

Description: This function opens a special file that contains definition for CDBs/ATA commands and sense data.

Default Parm Order: `filename ?-dont_overwrite?`

Parameters:

Name	Description
`filename`	Filename to parse
`-dont_overwrite`	Optional: Do not overwrite any previous commands of the same name
`-dont_overwrite`	Optional: Do not overwrite any previous commands of the same name

Examples:

```
eparse $::env(NiagaraPath)/commands/cdb_commands.txt
```

D.127 err_str

Command Name(s): `err_str`

Description: Converts the error code returned by the ec variable into a human readable string.

Default Parm Order: `error_code`

Parameters:

Name	Description
`error_code`	error code

Examples:

```
err_str
```

D.128 esource

Command Name(s): `esource`

Description: This version of source acts identical to TCL version except that it can also execute .stc files. A .stc file is an encrypted TCL file created with the encode command.

Default Parm Order: `filename`

Parameters:

Name	Description
`filename`	Tcl filename

Examples:

```
  encode test.tcl
esource test.stc
```

D.129 fcal abort_task_set

Command Name(s): `fcal abort_task_set`

Description: This function sends a low-level abort_task_set FCAL frame to the device. Clearly, this command is intended for FCAL devices only.

Default Parm Order: `?ox_id?`

Parameters:

Name	Description
`ox_id`	Optional: ox_id to us with command

Examples:

```
fcal abort_task_set ?aux_id?
```

D.130 fcal abts

Command Name(s): `fcal abts`

Description: This function sends a low-level abort sequence FCAL frame to the device. Clearly, this command is intended for FCAL devices only.

Default Parm Order: `<None>`

Examples:

```
fcal abts
```

D.131 fcal clear_aca

Command Name(s): `fcal clear_aca`

Description: This function sends a low-level clear_aca FCAL frame to the device. Clearly, this command is intended for FCAL devices only.

Default Parm Order: `<None>`

Examples:

```
fcal clear_aca
```

D.132 fcal clear_task_set

Command Name(s): `fcal clear_task_set`

Description: This function sends a low-level clear_task_set FCAL frame to the device. Clearly, this command is intended for FCAL devices only.

Default Parm Order: `<None>`

Examples:

```
fcal clear_task_set
```

D.133 fcal lip_reset

Command Name(s): `fcal lip_reset`

Description: This function sends a LIP followed by a port and process login to all devices.Normally you would want to use a device rescan instead of this command.

Default Parm Order: `<None>`

Examples:

```
fcal lip_reset
```

D.134 fcal port_login

Command Name(s): `fcal port_login`

Description: This function sends a low-level port_login FCAL frame to the device. Clearly, this command is intended for FCAL devices only.

Default Parm Order: `payload_size`

Parameters:

Name	Description
`payload_size`	New payload size

Examples:

```
fcal port_login
```

D.135 fcal process_login

Command Name(s): `fcal process_login`

Description: This function sends a low-level process_login FCAL frame to the device. Clearly, this command is intended for FCAL devices only.

Default Parm Order: `<None>`

Examples:

```
fcal process_login
```

D.136 fcal reset

Command Name(s): `fcal reset`

Description: This function sends a reset followed by a port and process login to all devices. Normally you would want to use a device rescan instead of this command.

Default Parm Order: `<None>`

Examples:

```
fcal reset
```

D.137 fcal target_reset

Command Name(s): `fcal target_reset`

Description: This function sends a low-level target_reset FCAL frame to the device. Clearly, this command is intended for FCAL devices only.

Default Parm Order: `<None>`

Examples:

```
fcal target_reset
```

D.138 fcal term_task

Command Name(s): `fcal term_task`

Description: This function sends a low-level term_task FCAL frame to the device.Clearly, this command is intended for FCAL devices only.

Default Parm Order: `<None>`

Examples:

```
fcal term_task
```

D.139 feedback asynccqe

Command Name(s): `feedback asynccqe`

Description: This function turns the output of asynchronus NVMe CQEs on/off.A zero value turns the CQE display off, a non-zero value turns CQE output on.This function is used to make a trade-off between functionality and output speed.

Default Parm Order: `?flag?`

Parameters:

Name	Description
`flag`	Optional: 0 to not show NVME CQEs, non-zero to show NVME CQEs

Examples:

```
feedback asynccqe
```

D.140 feedback color

Command Name(s): `feedback color`

Description: This function turns embedded color on/off. A zero value turns embedded color on, non-zero value turns color off. Note that this setting is only effective when using Niagara through an ANSI compatible terminal. On other terminals, this option will add "garbage" characters to the display. The default setting for this function is off. This setting is generally called in a startup script.

Default Parm Order: `?flag?`

Parameters:

Name	Description
`flag`	Optional: 0 turns off color, non zero turns on color

Examples:

```
feedback color
```

D.141 feedback default

Command Name(s): `feedback default`

Description: This function sets the current feedback level to a known state. This setting sets maxlen=255, showcmd=true, and showatafis=true.

Default Parm Order: `<None>`

Examples:

```
feedback default
```

D.142 feedback maxlen

Command Name(s): `feedback maxlen`

Description: This function changes the maximum number of buffer bytes returned by a command. Whenever a command is called, the number of buffer bytes will be either the number actually returned, or the number specified by this function, whichever is smaller. If a returned buffer prints less bytes than are returned, a ... is printed at the end of the hex dump. This command is primarily used to make a trade-off between return verbosity and speed. When trying to achieve maximum performance on high speed commands, set this value to zero for a notable speed improvement.

Default Parm Order: `?length?`

Parameters:

Name	Description
`length`	Optional: Number of buffer bytes

Examples:

```
feedback maxlen
```

D.143 feedback min

Command Name(s): `feedback min`

Description: This function sets the maximum number of returned buffer bytes to zero and turns off CDB output. The result is improved execution performance. Use this command to set Niagara for maximum speed.

Default Parm Order: `<None>`

Examples:

```
feedback min
```

D.144 feedback pop

Command Name(s): `feedback pop`

Description: This function pops the current state off the "feedback stack". This allows a user to store a feedback state and recall it. This command is useful for storing a feedback state before changing it for performance, or other reasons.

Default Parm Order: `<None>`

Examples:

```
feedback pop
```

D.145 feedback push

Command Name(s): `feedback push`

Description: This function pushes the current state off the "feedback stack". This allows a user to recall a stored feedback state.

Default Parm Order: `<None>`

Examples:

```
feedback push
```

D.146 feedback showatafis

Command Name(s): `feedback showatafis`

Description: This function turns the output of the ATA return FIS on/off. A zero value turns the FIS display off, a non-zero value turns FIS output on. This function is used to make a trade-off between functionality and output speed.

Default Parm Order: `?flag?`

Parameters:

Name	Description
`flag`	Optional: 0 to not show ATA return FIS, non-zero to show ATA return FIS
`flag`	Optional: 0 to not show ATA return FIS, non-zero to show ATA return FIS

Examples:

```
feedback showatafis
```

D.147 feedback showcdb

Command Name(s): `feedback showcdb`

Description: This function turns the output of the CDB/ATA command on/off. A zero value turn the CDB display off, a non-zero value turns CDB output on. This function is used to make a (slight) trade-off between functionality and output speed.

Default Parm Order: `?flag?`

Parameters:

Name	Description
`flag`	Optional: 0 to not show cdb, non-zero to show CDB

Examples:

```
feedback showcdb
```

D.148 feedback showcqe

Command Name(s): `feedback showcqe`

Description: This function turns the output of the NVMe Completion Queue entry on/off.A zero value turns the CQE display off, a non-zero value turns CQE output on.This function is used to make a trade-off between functionality and output speed.

Default Parm Order: `?flag?`

Parameters:

Name	Description
`flag`	Optional: 0 to not show NVMe CQEs, non-zero to show NVMe CQE

Examples:

```
feedback showcqe
```

D.149 get_cil_list

Command Name(s): `get_cil_list`

Description: Return a list of all CIL commands, listed in the order that the command was created.

Default Parm Order: `<None>`

Examples:

```
get_cil_list
```

D.150 get_cq_str

Command Name(s): `get_cq_str`

Description: This command returns the full descriptive error string associated with the specified cq value.

Default Parm Order: `cq`

Parameters:

Name	Description
`cq`	CQ value to get matching error string

Examples:

```
dword3 80170010
```

D.151 get_kcq_str

Command Name(s): `get_kcq_str`

Description: This command returns the full descriptive error string associated with the specified kcq value.

Default Parm Order: `kcq`

Parameters:

Name	Description
`kcq`	KCQ value to get matching error string

Examples:

```
kcq 052000
```

D.152 init

Command Name(s): `init`

Description: No Description Given.

Default Parm Order: `<None>`

Examples:

```
No Examples given
```

D.153 niagara_log_puts

Command Name(s): `niagara_log_puts`

Description: This function will log a message to the Niagara log file which can be useful in debugging problemsshould they arise.

Default Parm Order: `message`

Parameters:

Name	Description
`message`	Message to log to the Niagara log

Examples:

```
niagara_log_puts "Hello World
" ;# This will print the line "Hello World" into the Niagara log
```

D.154 nvme dump_cid

Command Name(s): `nvme dump_cid`

Description: This function dumps a specific command from the submission and completion queue based on command identifier.

Default Parm Order: `qid cid ?index?`

Parameters:

Name	Description
`qid`	Submission Queue ID
`cid`	Command identifier
`index`	Optional: Device to run the command on

Examples:

```
nvme dump_cid 1 0x0014
```

D.155 nvme dump_cq

Command Name(s): `nvme dump_cq`

Description: This function dumps the current head, tail, size, and contents of a completion queue.If 'all' is specified, it dumps every completion queue owned by this controller, orthe controller associated with the selected namespace.

Default Parm Order: `qid ?first_entry? ?last_entry? ?index?`

Parameters:

Name	Description
`qid`	Queue ID to dump
`first_entry`	Optional: First command dumped
`last_entry`	Optional: Last command dumped
`index`	Optional: Device to run the command on

Examples:

```
nvme dump_cq 0

nvme dump_cq 0 6 25
```

D.156 nvme dump_sq

Command Name(s): `nvme dump_sq`

Description: This function dumps the current head, tail, size, and contents of a submission queue.If 'all' is specified, it dumps every submission queue owned by this controller, orthe controller associated with the selected namespace.

Default Parm Order: `qid ?first_entry? ?last_entry? ?index?`

Parameters:

Name	Description
`qid`	Queue ID to dump
`first_entry`	Optional: First command dumped
`last_entry`	Optional: Last command dumped
`index`	Optional: Device to run the command on

Examples:

```
nvme dump_sq 1

nvme dump_sq 0 6 25
```

D.157 nvme get callback reset

Command Name(s): `nvme get callback reset`

Description: This function returns the callback mapped to a specific device command. Callbacks are code segments that are automatically called whenever a command is executed. Currently supported callbacks are reset.

Default Parm Order: `<None>`

Examples:

```
nvme get callback reset
```

D.158 nvme get controller

Command Name(s): `nvme get controller`

Description: This function returns the controller id corresponding to the given device index.

Default Parm Order: `dev_idx`

Parameters:

Name	Description
dev_idx	The device index of the controller

Examples:

```
nvme get controller 0
```

D.159 nvme get cq_ids

Command Name(s): nvme get cq_ids

Description: This function returns a list of completion queue ids that currently exist for the selected controller, or the controller associated with the selected namespace.

Default Parm Order: ?index?

Parameters:

Name	Description
index	Optional: Device to run the command on

Examples:

```
nvme get cq_ids
```

D.160 nvme get cq_size

Command Name(s): nvme get cq_size

Description: This function returns the size of a completion queue.

Default Parm Order: qid ?index?

Parameters:

Name	Description
qid	Queue ID to return size of
index	Optional: Device to run the command on

Examples:

```
nvme get cq_size 0
```

D.161 nvme get device_index

Command Name(s): `nvme get device_index`

Description: This function returns the device index corresponding to the given controller id.

Default Parm Order: `ctrlid serial_num`

Parameters:

Name	Description
`ctrlid`	ID of the controller
`serial_num`	Serial number of the controller

Examples:

```
nvme get device index 0 CJH001000303
```

D.162 nvme get drain_cq

Command Name(s): `nvme get drain_cq`

Description: This function returns the value describing whether the completion queue will automatically receive commands, or if it will wait until nvme drain_cq is set to 1.

Default Parm Order: `?index?`

Parameters:

Name	Description
`index`	Optional: Device to run the command on

Examples:

```
nvme get drain_cq
```

D.163 nvme get last_cid

Command Name(s): `nvme get last_cid`

Description: This function returns the command ID (CID) of the last completion entry received by the driver. It may be helpful to turn off asynchronus events, otherwise the "last" completion entry may not be for the command that was just issued.

Default Parm Order: `?index?`

Parameters:

Name	Description
`index`	Optional: Device to run the command on

Examples:

```
nvme get last_cid
```

D.164 nvme get last_dword

Command Name(s): `nvme get last_dword`

Description: This function returns the specified dword of the last completion entry received by the driver. It may be helpful to turn off asynchronus events, otherwise the "last" completion entry may not be for the command that was just issued.

Default Parm Order: `dword ?index?`

Parameters:

Name	Description
`dword`	Dword to gather
`index`	Optional: Device to run the command on

Examples:

```
nvme get last_dword 2
```

D.165 nvme get last_dword0

Command Name(s): `nvme get last_dword0`

Description: This function returns the first dword of the last completion entry received by the driver. It may be helpful to turn off asynchronus events, otherwise the "last" completion entry may not be for the command that was just issued.

Default Parm Order: `?index?`

Parameters:

Name	Description
`index`	Optional: Device to run the command on

Examples:

```
nvme get last_dword0
```

D.166 nvme get last_dword1

Command Name(s): `nvme get last_dword1`

Description: This function returns the second dword of the last completion entry received by the driver. It may be helpful to turn off asynchronus events, otherwise the "last" completion entry may not be for the command that was just issued.

Default Parm Order: `?index?`

Parameters:

Name	Description
`index`	Optional: Device to run the command on

Examples:

```
nvme get last_dword1
```

D.167 nvme get last_err_logpage

Command Name(s): `nvme get last_err_logpage`

Description: This function returns the last error log page. This is set when the more bit is set in a completion queue entry.

Default Parm Order: `?index?`

Parameters:

Name	Description
`index`	Optional: Device to run the command on

Examples:

```
nvme get last_err_logpage
```

D.168 nvme get last_status

Command Name(s): `nvme get last_status`

Description: This function returns the status code and status code type of the last completion queue entry received by the driver. It may be helpful to turn off asynchronus events, otherwise the "last" completion entry may not be for the command that was just issued.

Default Parm Order: `?index?`

Parameters:

Name	Description
`index`	Optional: Device to run the command on

Examples:

```
nvme get last_status
```

D.169 nvme get page_size

Command Name(s): `nvme get page_size`

Description: This function returns the memory page size (in bytes) for the selected controller, or the controller associated with the selected namespace.

Default Parm Order: `?index?`

Parameters:

Name	Description
`index`	Optional: Device to run the command on

Examples:

```
nvme get page_size
```

D.170 nvme get register

Command Name(s): `nvme get register`

Description: This function reads a specified register. The registers can be referenced byname (CAP, VS, CC, CSTS, AQA, ASQ, ACQ), or by address.

Default Parm Order: `reg ?index?`

Parameters:

Name	Description
`reg`	Register to get
`index`	Optional: Device to run the command on.

Examples:

```
nvme get register CC
```

D.171 nvme get sq_ids

Command Name(s): `nvme get sq_ids`

Description: This function returns a list of submission queue ids that currently exist for the selected controller, or the controller associated with the selected namespace.

Default Parm Order: `?index?`

Parameters:

Name	Description
`index`	Optional: Device to run the command on

Examples:

```
nvme get sq_ids
```

D.172 nvme get sq_size

Command Name(s): `nvme get sq_size`

Description: This function returns the size of a submission queue.

Default Parm Order: `qid ?index?`

Parameters:

Name	Description
`qid`	Queue ID to return size of
`index`	Optional: Device to run the command on

Examples:

```
nvme get sq_size 1
```

D.173 nvme reset

Command Name(s): `nvme reset`

Description: This function performs either a NVM subsystem reset, a controller reset, or a shutdown on the device. A controller reset with the shutdown_type specified will shutdown the controller before resetting it. Valid reset types are "controller" and "nvm". "c" and "n" are also accepted.

Default Parm Order: `type ?shutdown_type?  ?index?  ?-no_reenable?  ?-timeout?`

Parameters:

Name	Description
`type`	The type of reset to perform
`shutdown_type`	Optional: Used to perform a normal/abrupt shutdown as part of the reset
`shutdown_type`	Optional: Used to perform a normal/abrupt shutdown as part of the reset
`index`	Optional: Device to run the command on.
`-no_reenable`	Optional: Do not attempt to re-enable the device
`-timeout`	Optional: Set the timeout (ms) value for the command

Examples:

```
nvme reset controller

nvme reset controller 1

nvme reset c 1 -no_reenable
```

D.174 nvme reset_queue

Command Name(s): `nvme reset_queue`

Description: This function performs a queue level reset, destroying a queue after allowing commands to complete and recreating it. A new queue depth can be specified for when it is recreated.

Default Parm Order: `type qid ?depth?  ?device?`

Parameters:

Name	Description
`type`	Type of queue, submission or completion
`qid`	ID of the queue to reset
`depth`	Optional: Depth of the queue to recreate with. Defaults to same as the destroyed queue
`depth`	Optional: Depth of the queue to recreate with. Defaults to same as the destroyed queue
`device`	Optional: Device to perform the queue reset on

Examples:

```
nvme reset_queue completion 1

nvme reset_queue c 1 8

nvme reset_queue submission 1 8

nvme reset_queue s 1 8 2
```

D.175 nvme set callback reset

Command Name(s): `nvme set callback reset`

Description: This function is used to set a callback for a particular command. Each time the command is called, this callback is executed as code. Currently supported callbacks are "reset". Each of these callbacks correspond to the associated nvme command. Note that it is generally a good idea to make sure a callback is unset before setting it yourself. To remove a callback, set it to "".

Default Parm Order: `callback`

Parameters:

Name	Description
`callback`	Code to execute

Examples:

```
nvme set callback reset
```

D.176 nvme set drain_cq

Command Name(s): `nvme set drain_cq`

Description: This function attempts to set a value that determines whether the completion queue will automatically receive commands. Setting this value to 1 will drain the completion queue of completed commands while 0 will cause Niagara to wait until drain_cq is set back to 1 before draining commands from CQ.

Default Parm Order: `auto_drain ?index?`

Parameters:

Name	Description
`auto_drain`	Determines whether CQ automatically receives commands
`index`	Optional: Device to run the command on

Examples:

```
nvme set drain_cq 0
```

D.177 nvme set page_size

Command Name(s): `nvme set page_size`

Description: This function sets the memory page size (in bytes) for the selected controller, or the controller associated with the selected namespace. This value is used for PRP entry size and will result in a controller reset.

Default Parm Order: `size ?index?`

Parameters:

Name	Description
`size`	page size (in bytes) to set for controller. Valid Options: 4k/8k
`index`	Optional: Device to run the command on

Examples:

```
nvme set page_size 4096

nvme set page_size 4k
```

D.178 nvme set register

Command Name(s): `nvme set register`

Description: This function attempts to set a given NVMe register to a given value. The register can be specified by name, or as an offset and a mask. !!!WARNING: SETTING REGISTER BY OFFSET CAN CORRUPT THE DEVICE AND/OR NIAGARA!!! USE AT YOUR OWN RISK.

Default Parm Order: `reg data ?mask? ?index? ?-o?`

Parameters:

Name	Description
`reg`	The register to set
`data`	The value to
`mask`	Optional: Mask to use when setting the register. Only used when a bar offset is given.
`mask`	Optional: Mask to use when setting the register. Only used when a bar offset is given.
`index`	Optional: Device to run the command on.
`-o`	Optional: Overrides register restriction, USE WITH CARE!!!

Examples:

```
nvme set register CC.AMS 0x03
nvme set register 0x14 0x1000 0x1800
```

D.179 parse

Command Name(s): `parse`

Description: This function opens a special file that contains definition for CDBs/ATA commands and sense data.

Default Parm Order: `filename ?-dont_overwrite?`

Parameters:

Name	Description
`filename`	Filename to parse
`-dont_overwrite`	Optional: Do not overwrite any previous commands of the same name
`-dont_overwrite`	Optional: Do not overwrite any previous commands of the same name

Examples:

```
parse $::env(NiagaraPath)/commands/cdb_commands.txt
```

D.180 pcie get config

Command Name(s): `pcie get config`

Description: This function reads a specified register. The registers can be referenced byname (ID, CMD, STS, RID, CC, CLS, MLT, HTYPE, BIST, BAR0, BAR1, BAR2, CCPTR, SS, EROM, CAP, INTR), or by starting offset.Sub-addresses can be specified with a period.

Default Parm Order: `reg ?length?  ?index?`

Parameters:

Name	Description
`reg`	Registery name or starting address
`length`	Optional: Length of the read. Valid lengths are 1, 2, and 4 bytess. Defaults to 4.
`length`	Optional: Length of the read. Valid lengths are 1, 2, and 4 bytess. Defaults to 4.
`index`	Optional: Device to run the command on.

Examples:

```
pcie get config 0x00 1 0x0F
pcie get config ID
pcie get config ID.DID
```

D.181 pcie get driver

Command Name(s): `pcie get driver`

Description: This function reports active drivers per drive.

Default Parm Order: `?index?  ?-d?  ?-v?`

Parameters:

Name	Description
`index`	Optional: Device to run the command on. Only supported on pciport driver.
`index`	Optional: Device to run the command on. Only supported on pciport driver.
`-d`	Optional: Allow specifying an lspci device for index instead
`-v`	Optional: Outputs more verbose info in human readable format, else outputs 1 for up and 0 for down
`-v`	Optional: Outputs more verbose info in human readable format, else outputs 1 for up and 0 for down

Examples:

```
pcie get driver
```

D.182 pcie get speed

Command Name(s): `pcie get speed`

Description: This function returns a drive's reported speed or -1 for no device found.

Default Parm Order: `?index?  ?-d?  ?-v?`

Parameters:

Name	Description
`index`	Optional: Device index to run the command on. Only supported on pciport driver..
`index`	Optional: Device index to run the command on. Only supported on pciport driver..
`-d`	Optional: Allow specifying an lspci device for index instead
`-v`	Optional: Outputs more verbose info in human readable format, else outputs speed
`-v`	Optional: Outputs more verbose info in human readable format, else outputs speed

Examples:

```
pcie get speed
```

D.183 pcie get status

Command Name(s): `pcie get status`

Description: This function returns 1 for up, 0 for down, -1 for no device found.

Default Parm Order: `?index?  ?-d?  ?-v?`

Parameters:

Name	Description
`index`	Optional: Niagara device to run the command on. Only supported on pciport driver.
`index`	Optional: Niagara device to run the command on. Only supported on pciport driver.
`-d`	Optional: Allow specifying an lspci device for index instead
`-v`	Optional: Outputs more verbose info in a human-readable format

Examples:

```
pcie get status

pcie get status 0
```

D.184 pcie get width

Command Name(s): `pcie get width`

Description: This function determines a drive's reported width.

Default Parm Order: `?index?  ?-d?  ?-v?`

Parameters:

Name	Description
`index`	Optional: Device to run the command on. Only supported on pciport driver.
`index`	Optional: Device to run the command on. Only supported on pciport driver.
`-d`	Optional: Allow specifying an lspci device for index instead
`-v`	Optional: Outputs more verbose info in human readable format, else outputs width
`-v`	Optional: Outputs more verbose info in human readable format, else outputs width

Examples:

```
pcie get width
```

D.185 pcie set config

Command Name(s): `pcie set config`

Description: This function attempts to set a given pcie register to a given value. The register can be specified by name, or as an offset and a mask.

Default Parm Order: `reg data ?length?  ?mask?  ?index?`

Parameters:

Name	Description
`reg`	The register to set
`data`	The value to write
`length`	Optional: Length of the read. Valid lengths are 1, 2, and 4 bytes. Defaults to 4
`length`	Optional: Length of the read. Valid lengths are 1, 2, and 4 bytes. Defaults to 4
`mask`	Optional: Mask to use when setting the register. Only used when a bar offset is given.
`mask`	Optional: Mask to use when setting the register. Only used when a bar offset is given.
`index`	Optional: Device to run the command on.

Examples:

```
pcie set config CC.AMS 2
pcie set config 0x14 0x01 4 0x1800
```

D.186 perfcnt clicks

Command Name(s): `perfcnt clicks`

Description: This function functions like [clock clicks] but returns value in microseconds instead of milliseconds. See also: perfcnt count, perfcnt freq.

Default Parm Order: `<None>`

Examples:

```
perfcnt clicks
```

D.187 perfcnt count

Command Name(s): `perfcnt count`

Description: Result is given in a list of [HighPart, LowPart], with the HighPart given in seconds. See also: perfcnt freq, perfcnt clicks.

Default Parm Order: `<None>`

Examples:

```
perfcnt count
```

D.188 perfcnt delay

Command Name(s): `perfcnt delay`

Description: Delay is performed using a high resolution timer. See also: perfcnt count, perfcnt clicks.

Default Parm Order: `usecs_to_delay`

Parameters:

Name	Description
`usecs_to_delay`	Microseconds to delay

Examples:

```
perfcnt delay 1000000
```

D.189 perfcnt freq

Command Name(s): `perfcnt freq`

Description: Result is given in list of [HighPart, LowPart]. Most likely ticks per seconds will not exceed the LowPart. See also: perfcnt count, perfcnt clicks.

Default Parm Order: `<None>`

Examples:

```
perfcnt freq
```

D.190 pqi dump_iq

Command Name(s): `pqi dump_iq`

Description: This function dumps the current PI, CI, size, and contents of an inbound queue. If 'all' is specified, it dumps every inbound queue related to this device.

Default Parm Order: `qid`

Parameters:

Name	Description
`qid`	Queue ID to dump

Examples:

```
pqi dump_iq 0
```

D.191 pqi dump_oq

Command Name(s): `pqi dump_oq`

Description: This function dumps the current PI, CI, size, and contents of an outbound queue. If 'all' is specified, it dumps every completion queue owned by this controller, orthe controller associated with the selected namespace.

Default Parm Order: `qid`

Parameters:

Name	Description
`qid`	Queue ID to dump

Examples:

```
pqi dump_oq 1
```

D.192 pqi get register

Command Name(s): `pqi get register`

Description: This function reads a given PQI register. The register offset must be dword aligned.

Default Parm Order: `reg`

Parameters:

Name	Description
`reg`	Register offset to use

Examples:

```
pqi get register 0x20
```

D.193 pqi set register

Command Name(s): `pqi set register`

Description: This function attempts to set a given PQI register to a given value. The register offset must be dword aligned.

Default Parm Order: `reg data mask`

Parameters:

Name	Description
reg	The register to set
data	The value to write
mask	Mask to use when setting the register.

Examples:

```
pqi set register 0x14 0x1000 0x1800
```

D.194 qctl get auto_incr

Command Name(s): `qctl get auto_incr`

Description: This function returns a 1 if auto tag increment is enabled, zero otherwise.If auto tag increment is on, the tag_id sent with command is incremented by one each time a command is sent. See also: qctl set auto_incr, qctl set tag_id.

Default Parm Order: `<None>`

Examples:

```
qctl get auto_incr
```

D.195 qctl get ignore_queue_full

Command Name(s): `qctl get ignore_queue_full`

Description: The ignore_queue_full bit decides whether to report an error message if the current queue depth exceeds the maximum queue depth. A value of '1' disables the Queue Full error message report.

Default Parm Order: `<None>`

Examples:

```
qctl get ignore_queue_full
```

D.196 qctl get max_depth

Command Name(s): `qctl get max_depth`

Description: This function returns the current setting for the maximum queue depth. See also: qctl set max_depth.

Default Parm Order: `?sq?  ?-driver_max?  ?sq?`

Parameters:

Name	Description
sq	Optional: Return the depth of a specific submission queue
-driver_max	Optional: Return the driver's max queue depth instead of the current max queue depth
-driver_max	Optional: Return the driver's max queue depth instead of the current max queue depth
sq	Optional: Only send a specific submission queue's commands

Examples:

```
qctl get max_depth
```

D.197 qctl get num_queued

Command Name(s): `qctl get num_queued`

Description: This function returns the number of commands that have been issued, but haven't had status returned.

Default Parm Order: `?sq?  ?-all?`

Parameters:

Name	Description
sq	Optional: Return the number of commands of a specific submission queue
sq	Optional: Return the number of commands of a specific submission queue
-all	Optional: Returns number of commands of all submission queues

Examples:

```
qctl get num_queued 1
```

D.198 qctl get num_waiting

Command Name(s): `qctl get num_waiting`

Description: This function returns the number of queued commands that have returned status.This function is useful for determining an estimation of realized queue depth. See also: qmode concurrent, qmode stacked.

Default Parm Order: `?device?`

Parameters:

Name	Description
device	Optional: Device to recieve from

Examples:

```
qctl get num_waiting
```

D.199 qctl get tag_type

Command Name(s): `qctl get tag_type`

Description: This function returns the current queue tag type. Types include simple, ordered and head. See also: qctl set tag_type.

Default Parm Order: `<None>`

Examples:

```
qctl get tag_type
```

D.200 qctl idx_info

Command Name(s): `qctl idx_info`

Description: This function returns status information for a specified queue index. When "qctl recv all" is called, queue entries are placed in an internal table.This function is one of the way to examine the contents of the table. See also: qctl recv all, qctl table_info, qctl tag_info.

Default Parm Order: `index ?-list?`

Parameters:

Name	Description
`index`	Index of command (ordered by reply from device)
`-list`	Optional: Output results in a list structure

Examples:

```
qmode concurrent
do 8 {r10 [randlba]}
qctl recv all
qctl idx_info 0
```

D.201 qctl recv

Command Name(s): `qctl recv`

Description: This function waits for and retrieves the next available command from the device. Status on the commands success is returned See also: qctl recv all, qctl recv tag.

Default Parm Order: `?device?`

Parameters:

Name	Description
`device`	Optional: Device to recieve from

Examples:

```
# Queue 1000 commands at qd=8
qmode concurrent
for {set i 0} {$i < 8} {incr i} { r10 [randlba] }
for {set i 0} {$i < 1000} {incr i} {
   r10 [randlba]
   qctl recv
}
```

D.202 qctl recv all

Command Name(s): `qctl recv all`

Description: Waits for and retrieves all outstanding commands from the device. Returns successif all commands were completed successfully or an error if any commands did not.Commands results are placed in a table that can be parsed with the "qctl idx_info","qctl tag_info", and "qctl table_info" commands. See also: qctl recv, qctl recv tag.

Default Parm Order: `?device?`

Parameters:

Name	Description
`device`	Optional: Device to recieve from

Examples:

```
    qmode concurrent
do 8 {r10 [randlba]}
qctl recv all
qctl table_info
```

D.203 qctl recv tag

Command Name(s): `qctl recv tag`

Description: This function waits for and retrieves commands from the current device until the tag id is received. Returns success if all commands up to and including <tag id> were successful andan error if any were not. See also: qctl set tag_id,qctl recv, qctl recv all.

Default Parm Order: `tag_id ?device?`

Parameters:

Name	Description
`tag_id`	Tag ID to wait for
`device`	Optional: Device to recieve from

Examples:

```
    qmode concurrent
do 8 {r10 [randlba]}
qctl recv all
qctl table_info
```

D.204 qctl send

Command Name(s): `qctl send`

Description: This function sends an internal table of commands that were built up in stacked/pcie queuing mode to a device. This command cannot be used in concurrent mode. See also: qmode stacked, qmode concurrent, qmode info, qmode pcie, init, qctl recv.

Default Parm Order: `?device?`

Parameters:

Name	Description
`device`	Optional: Send commands queued in specified device

Examples:

```
# Send 32 stacked commands to the drive
qmode stacked
do 32 { r10 [randlba] }
qctl send
qctl recv all
qmode disable
```

D.205 qctl set auto_incr

Command Name(s): `qctl set auto_incr`

Description: This function turns tag_id auto increment on/off. When auto increment is on, the tag_id will automatically increment by 1 every time a command is send. Normally you want this setting on for convenience. See also: qctl set tag_id, qctl get auto_incr.

Default Parm Order: `auto_incr`

Parameters:

Name	Description
`auto_incr`	Zero turn off auto increment, non zero turns it on

Examples:

```
qctl set auto_incr 1
```

D.206 qctl set ignore_queue_full

Command Name(s): `qctl set ignore_queue_full`

Description: This function enables or disables QUEUE FULL error. The default setting is '0', if the queue buffer exceeds the max queue depth, a QUEUE_FULL error message will be returned. If sets to '1', the driver ignores any potential risk of sending too many commands to the HBA and will alow you to send as many commands as you like without error. Please be warned that this is an ADVANCED setting and you must make sure your queue depth does not exceed the maximum queue depth supported or you may experience an HBA hang.

Default Parm Order: `on/off`

Parameters:

Name	Description
`on/off`	1 to turn ignore_queue_full. 0 to turn ignore_queue_full off

Examples:

```
qctl set ignore_queue_full 0
```

D.207 qctl set max_depth

Command Name(s): `qctl set max_depth`

Description: This function tests a queuing depth to see if it is supported and makes any setting changes required to support the depth. Note that the default queuing depth may be deeperthan the depth specified. This command only guarantees that thecurrent queuing depth is at least that specified, the actual depth may be deeper. See also: qctl get max_depth.

Default Parm Order: `max_depth ?sq?`

Parameters:

Name	Description
max_depth	Maximum Queue Depth
sq	Optional: Sets the depth of a given submission queue

Examples:

```
qctl set max_depth 64
```

D.208 qctl set next_tag

Command Name(s): `qctl set next_tag`

Description: This function is used to explicitly set the tag_id of the next command. In most cases, "qctl set auto_incr 1" is a more convenient approach to handling tag ids. See also: qctl set auto_incr.

Default Parm Order: `tag_id`

Parameters:

Name	Description
tag_id	Tag ID of next tag

Examples:

```
qctl set tag_id 1234
r10
```

D.209 qctl set tag_type

Command Name(s): `qctl set tag_type`

Description: This function sets the tag type of commands that follow. Options for the type include simple, ordered and head. See also: qmode concurrent, qmode stacked, qctl get tag_type.

Default Parm Order: `tag_type`

Parameters:

Name	Description
tag_type	simple, ordered or head

Examples:

```
qmode stacked
# Send some simple commands
qctl set tag_type simple
r10
r10
r10
# Send some ordered commands
qctl set tag_type ordered
r10
r10
r10
qctl send
qctl recv all
qmode disable
```

D.210 qctl table_info

Command Name(s): `qctl table_info`

Description: This function returns a table describing the status of commands returnedby "qctl recv all". Using the -list option, the format is one that is easy to parse. Otherwise the format is one that is easy to read. See also: qctl recv all, qctl idx_info, qctl tag_info.

Default Parm Order: `?-list?`

Parameters:

Name	Description
`-list`	Optional: Output results in a list structure

Examples:

```
qmode concurrent
do 8 {r10 [randlba]}
qctl recv all
qctl table_info
```

D.211 qctl tag_info

Command Name(s): `qctl tag_info`

Description: This function returns information for a specified tag. When the "qctl recv all" commandis used, results of the commands are placed in an internal table. This funcition allow information for a specific tag to be extracted from the table. See also: qctl recv all, qctl idx_info, qctl table_info.

Default Parm Order: `tag_id ?-list?`

Parameters:

Name	Description
tag_id	Tag ID of command
-list	Optional: Output results in a list structure

Examples:

```
qmode concurrent
qctl set auto_incr 1
qctl set tag_id 21
do 8 {r10 [randlba]}
qctl recv tag 21
qctl tag_info 21
```

D.212 qmode concurrent

Command Name(s): qmode concurrent

Description: This function places Niagara in concurrent queuing mode. In this mode, Niagara does not wait for commands to return status before returning control to the user. When status is needed, it is retrieved using a "qctl recv" command. Return to normal mode with a "qmode disable" command. See also: qmode stacked, qmode info, qmode disable, qctl recv, init.

Default Parm Order: ?depth?

Parameters:

Name	Description
depth	Optional: Maximum queue depth (default is 16)

Examples:

```
# Test the performance of queued random reads at qd=1 vs qd=8
proc perf_test {iterations} {
   # Try to speed things up
   catch { device set read_xfer 0 }

   # qd = 1
   puts -nonewline "Timing $iterations random reads at qd=1... "
   set t1 [clock seconds]
   for {set i 0} {$i < $iterations} {incr i} { r10 [randlba] }
   set t2 [clock seconds]
   puts "[expr $t2 - $t1] seconds"

   set qit [expr $iterations - 8]

   #qd = 8
```

```
      qmode concurrent
      puts -nonewline "Timing $iterations random reads at qd=8... "
      set t1 [clock seconds]
      for {set i 0} {$i < 8} {incr i} { r10 [randlba] }
      for {set i 0} {$i < $qit} {incr i} {
         r10 [randlba]
         qctl recv
}
         set t2 [clock seconds]
         puts "[expr $t2 - $t1] seconds"

         # Clear queue
         qctl recv all

         # Everything back to normal
         init
         return "done"
      }
```

D.213 qmode disable

Command Name(s): `qmode disable`

Description: This function puts the current UIL in normal (non-queued) mode. In this mode commands to the drive wait for status (or time out) before returning control to the user. See also: qmode concurrent, qmode info, init.

Default Parm Order: `<None>`

Examples:

```
qmode disable
```

D.214 qmode info

Command Name(s): `qmode info`

Description: This function returns Niagara's current queuing mode. Possible modes are disabled, concurrent, stacked, and pcie. See also: qmode concurrent, qmode disable, qmode stacked, qmode pcie, init.

Default Parm Order: `<None>`

Examples:

```
qmode info
```

D.215 qmode pcie

Command Name(s): `qmode pcie`

Description: This function puts the current UIL into pcie queuing mode. In this mode, commands are placed into the IO submission queue specified by the -sq flag or into the last IO submission queue that received a command. They can be issued as a group with a "qctl send" command. Completion queues are not checked or emptied until a "qctl recv" command is issued.

Default Parm Order: `?depth?`

Parameters:

Name	Description
`depth`	Optional: Maximum allowed waiting commands per queue (default is SQ size)
`depth`	Optional: Maximum allowed waiting commands per queue (default is SQ size)

Examples:

```
# Send 32 pcie commands to the drive with one doorbell write
qmode pcie
do 32 { rn [randlba] }
qctl send
qctl recv all
qmode disable
```

D.216 qmode stacked

Command Name(s): `qmode stacked`

Description: This function puts the current UIL into stacked queuing mode. In this mode, commands are stacked in memory and then sent to the drive as fast as possible with a "qctl send" command. This mode is generally used to try and force a deep queue depthSee also: qmode concurrent, qmode info, qmode disable, qmode pcie, init.

Default Parm Order: `?depth?`

Parameters:

Name	Description
`depth`	Optional: Maximum queue depth (default is 16)

Examples:

```
# Send 32 stacked commands to the drive
qmode stacked
do 32 { r10 [randlba] }
qctl send
qctl recv all
qmode disable
```

D.217 rand

Command Name(s): `rand`

Description: This function generates a random, unsigned float in the range of [0.0:1.0]. An optional random channel can be specified (which can contain optional histogram information). If histogram information is provided, the range returned will be influenced by histogram information. If no channel is specified, channel 0 is used. - See Also: rand open, rand int, rand addhist, rand range, rand frange.

Default Parm Order: `?channel?`

Parameters:

Name	Description
`channel`	Optional: ID of the random channel to use (default=0)

Examples:

```
rand float
```

See Also: : `rand open` (page **??**), `rand int` (page 297), `rand addhist` (page 295), (page **??**)

D.218 rand addhist

Command Name(s): `rand addhist`

Description: This function defines a histogram entry for an open random channel. When a random number is generated, this entry will be used to predict the numbers value the specified percentage of the time. You can stack up to 32 histogram entrys per random generator. Note that once your added histogram entrys total 100 percent or above, adding further entrys will cause the least likely entrys in the histogram to never take effect (the most likely entrys will always consume the 100% range, leaving extra entrys with a zero percent chance of being used). If your histogram entries total to less than 100 percent, the default range (which varies per command) is used for the remaining percentage. Another important thing to note is that, if you define a histogram range that is outside of a particular commands (rand int, rand double, rand range, etc) range limit, an error will result. It is your responsibility to ensure that your histogram ranges are appropriate for the commands you are using. - See Also: rand showhist, buff fill rand, randlba, rand int, rand addhist, rand float, rand range, rand frange, rand open.

Default Parm Order: `channel min max percentage`

Parameters:

Name	Description
`channel`	ID of open random channel (from rand open)
`min`	The minimum range of the histogram entry
`max`	The maximum range of the histogram entry
`percentage`	The percentage that the histogram entry is used

Examples:

```
# Fill a buffer with 5 percent 0xff and 95 percent zero:
  set r [rand open]
  rand addhist $r 0xff 0xff 5
  rand addhist $r 0 0 95
  bfr send 0 512 $r
  rand close $r
```

See Also: : (page **??**)

D.219 rand close

Command Name(s): `rand close`

Description: This function closes a random channel opened with the "rand open" function and frees resources associated with the random channel. You can not close channel 0. Because the number of random channels is limited (1024), you should always close a random channel when finished with it. - See Also: rand open, rand int, rand addhist, rand float, rand range, rand frange, rand seed.

Default Parm Order: `channel`

Parameters:

Name	Description
`channel`	ID of the random channel to output histogram for

Examples:

```
set r [rand open]
rand close $r
```

See Also: : `rand` (page **??**)

D.220 rand float

Command Name(s): `rand float`

Description: This function generates a random, unsigned float in the range of [0.0:1.0]. An optional random channel can be specified (which can contain optional histogram information). If histogram information is provided, the range returned will be influenced by histogram information. If no channel is specified, channel 0 is used. - See Also: rand open, rand int, rand addhist, rand range, rand frange.

Default Parm Order: `?channel?`

Parameters:

Name	Description
channel	Optional: ID of the random channel to use (default=0)

Examples:

```
rand float
```

See Also: : rand open (page **??**), rand int (page 297), rand addhist (page 295), (page **??**)

D.221 rand frange

Command Name(s): rand frange

Description: This function generates a random, floating point number between [min:max]. An optional random channel can be specified. If no channel is specified, channel 0 is used. The formula used to determine the range is equivalent to: expr ([rand float]*($max-$min)) + $min. When using histograms with this command, it is generally clearer to define your histogram ranges between 0.0 and 1.0. All histogram ranges are applied to the formula given above. An alternative to this command is to simply bound the random range of a channel with the "rand addhist" command. - See Also: rand open, rand int, rand addhist, rand range.

Default Parm Order: min max ?channel?

Parameters:

Name	Description
min	Minimum float value
max	Maximum float value
channel	Optional: ID of the random channel to use (default=0)

Examples:

```
# One way to do it:
  rand frange 0.0 5.0

# Another way to do it:
  set r [rand open]
  rand addhist $r 0.0 5.0 100
  rand float $r
```

See Also: : rand open (page **??**), rand int (page 297), rand (page 295)

D.222 rand int

Command Name(s): rand int

Description: This function generates a random, unsigned integer between 0 and 0xFFFFFFFF. An optional random channel can be specified (which can contain optional histogram information). If no channel is specified, channel 0 is used. - See also: rand open, rand float, rand addhist, rand range, rand frange.

Default Parm Order: `?channel?`

Parameters:

Name	Description
`channel`	Optional: ID of the random channel to use (default=0)

Examples:

```
rand int
```

D.223 rand open

Command Name(s): `rand open`

Description: This Function creates a new random number generator channel. This channel can be used in commands like "randlba" and "buff fill rand" as well as any of the rand commands. This command returns a "handle" to the generator which can be used by the associated commands - See Also: buff fill rand, randlba, rand close, rand int, rand addhist, rand float, rand range, rand frange, rand seed.

Default Parm Order: `?seed?`

Parameters:

Name	Description
`seed`	Optional: The starting random seed (integer)

Examples:

```
# Create two independant random sources:
  set rlba [rand open 1234]
  set rblk [rand open 5678]

# Do some CSO with predicitable randomness
  for {set i 0} {$i < 1000} {incr i} {
     bfr 0 0 512 $rblk
     r10 [randlba 1 $rlba]
  }
```

See Also: : `buff fill rand` (page **??**), (page **??**)

D.224 rand range

Command Name(s): `rand range`

Description: This function generates a random, unsigned integer between [min:max]. An optional random channel can be specified. If no channel is specified, channel 0 is used. The formula used to determine the range is equivalent to: expr ([rand int]%($max-$min + 1)) + $min. It is generally not recommended that histogram random channels be used in combination with this command (this is not generally a problem since you can define ranges within the histogram itself). Histogram ranges outside of those asked for with this command are applied to the formula given above. This can lead to confusing results. An alternative to this command is to simply bound the random range of a channel with the "rand addhist" command. - See Also: rand open, rand float, rand addhist, rand frange.

Default Parm Order: `min max ?channel?`

Parameters:

Name	Description
`min`	Minimum integer value
`max`	Maximum integer value
`channel`	Optional: ID of the random channel to use (default=0)

Examples:

```
# One way to do it:
  rand range 0 5

# Another way to do it:
  set r [rand open]
  rand addhist $r 0 5 100
  rand int $r
```

See Also: : `rand open` (page **??**), `rand float` (page 296), `rand` (page 295)

D.225 rand seed

Command Name(s): `rand seed`

Description: This funciton seeds a random channel. The seed value is an integer ranging from 0-0xFFFFFFFF. Is no channel is specified, channel zero is seeded - See Also: rand open, rand int, rand float, rand range, rand frange.

Default Parm Order: `chanSeed ?channel?`

Parameters:

Name	Description
chanSeed	Seeds a random channel
channel	Optional: ID of the random channel to seed (default is 0)

Examples:

```
# Seed channel zero
  rand seed [clock seconds]

# Seed channel 1
  rand seed [clock seconds] 1
```

See Also: : rand open (page **??**), rand int (page 297), rand float (page 296), rand range (page 299), rand frange (page 297)

D.226 rand showhist

Command Name(s): rand showhist

Description: This function outputs the histogram settings for an existing random channel. Note that the histogram entries may be in a different order than they were added. This happens because the rand addhist command orders the histogram entries for optimal performance - See Also: rand open, rand addhist.

Default Parm Order: channel

Parameters:

Name	Description
channel	ID of the random channel to output histogram for

Examples:

```
set r [rand open]
rand addhist $r 0 10 20
rand addhist $r 10 20 60
rand addhist $r 30 30 20
rand showhist
```

See Also: : rand open (page **??**), rand addhist (page 295)

D.227 randlba

Command Name(s): randlba

Description: This command provides a convenient way to produce a random lba. The lba produced will be somewhere between zero and "maxlba - maxblk". An optional random channel can be specified.

Default Parm Order: `?maxblk?  ?channel?`

Parameters:

Name	Description
`maxblk`	Optional: Maximum number of blocks that will be read
`channel`	Optional: Random channel to use (default=0)

Examples:

```
do 1000 { r10 [randlba] }
```

D.228 sas abort_task_set

Command Name(s): `sas abort_task_set`

Description: This function sends a low-level abort_task_set SAS frame to the device. Clearly, this command is intended for SAS devices only.

Default Parm Order: `?ox_id?`

Parameters:

Name	Description
`ox_id`	Optional: ox_id to use with this command

Examples:

```
    sas abort_task_set
sas abort_task_set 0
```

D.229 sas clear_aca

Command Name(s): `sas clear_aca`

Description: This function sends a low-level clear_aca SAS frame to the device. Clearly, this command is intended for SAS devices only.

Default Parm Order: `<None>`

Examples:

```
sas clear_aca
```

D.230 sas clear_task_set

Command Name(s): `sas clear_task_set`

Description: This function sends a low-level clear_task_set SAS frame to the device. Clearly, this command is intended for SAS devices only.

Default Parm Order: `<None>`

Examples:

```
sas clear_task_set
```

D.231 sas get_pod_address

Command Name(s): `sas get_pod_address`

Description: Gets SAS pod Address for current device.

Default Parm Order: `<None>`

Examples:

```
sas get_pod_address
```

D.232 sas get_sas_address

Command Name(s): `sas get_sas_address`

Description: Gets SAS Address for current device.

Default Parm Order: `<None>`

Examples:

```
sas get_sas_address
```

D.233 sas get_speed

Command Name(s): `sas get_speed`

Description: Returns the current speed of the SAS bus. This could be 1.5, 3.0, or 6.0 Gbs.

Default Parm Order: `<None>`

Examples:

```
sas get_speed
```

D.234 sas link_reset

Command Name(s): `sas link_reset`

Description: Mode page 0, byte 4 indicates whether this is enabled or not.

Default Parm Order: `<None>`

Examples:

```
sas link_reset
```

D.235 sas lun_reset

Command Name(s): `sas lun_reset`

Description: This function sends a logical unit reset SAS task to the device. It waits up to 2 seconds for a response, but it does not verify that good status was received back.

Default Parm Order: `<None>`

Examples:

```
sas lun_reset
```

D.236 sas nexus_reset

Command Name(s): `sas nexus_reset`

Description: Performs a SAS I-T Nexus reset sequence to a specific LUN. The optional LUN value has a default for LUN 0.

Default Parm Order: `?lun?`

Parameters:

Name	Description
`lun`	Optional: LUN

Examples:

```
sas nexus_reset
sas nexus_reset 1
```

D.237 sas notify

Command Name(s): `sas notify`

Description: Sends NOTIFY (ENABLE SPINUP) SAS primitive to allow automatic unit start. Mode page 0, byte 4 indicates whether this is enabled or not.

Default Parm Order: `<None>`

Examples:

```
sas notify
```

D.238 sas notify_epow

Command Name(s): `sas notify_epow`

Description: Sends a SAS Notify (Power Loss Expected), or EPOW Notify, triple primitive.

Default Parm Order: `<None>`

Examples:

```
sas notify_epow
```

D.239 sas phy_disable_offline

Command Name(s): `sas phy_disable_offline`

Description: Disables SAS Phy ports that are offline (used to isolate offline devices).

Default Parm Order: `<None>`

Examples:

```
sas phy_disable_offline
```

D.240 sas phy_enable_disabled

Command Name(s): `sas phy_enable_disabled`

Description: Enable SAS Phy ports that are disabled (used to reconnect isolated devices).

Default Parm Order: `<None>`

Examples:

```
sas phy_enable_disabled
```

D.241 sas phy_pulse_disable

Command Name(s): `sas phy_pulse_disable`

Description: Pulses the Phy disable bit for duration in milliseconds.

Default Parm Order: `duration`

Parameters:

Name	Description
`duration`	pulse duration in milliseconds

Examples:

```
sas phy_pulse_disable 1000
```

D.242 sas phy_reset

Command Name(s): `sas phy_reset`

Description: Performs a SAS phy reset sequence (link reset minus sending identify).

Default Parm Order: `<None>`

Examples:

```
sas phy_reset
```

D.243 sas power_manage

Command Name(s): `sas power_manage`

Description: Sets the SAS Power Management for the current device to either full, partial, or slumber. No parameter willquery the current power management state.

Default Parm Order: `?mode?`

Parameters:

Name	Description
`mode`	Optional: either "full", "partial", or "slumber"

Examples:

```
    sas power_manage
sas power_manage slumber
```

D.244 sas query_async_event

Command Name(s): `sas query_async_event`

Description: Sends a Query Asynchronous Event TMF (formerly known as Query Unit Attention).

Default Parm Order: `<None>`

Examples:

```
sas query_async_event
```

D.245 sas query_task_set

Command Name(s): `sas query_task_set`

Description: Sends a Query Task Set TMF.

Default Parm Order: `<None>`

Examples:

```
sas query_task_set
```

D.246 sas reset

Command Name(s): `sas reset`

Description: This performs a SAS hard reset sequence. Normally you would want to use a device rescan instead of this command.

Default Parm Order: `<None>`

Examples:

```
sas reset
```

D.247 sas set_sas_address

Command Name(s): `sas set_sas_address`

Description: Sets the SAS Address for the current device. Current device from device get index. The SAS Address top byte is always 0x50. User specifies next seven bytes.

Default Parm Order: `address_byte1 address_byte2 address_byte3 address_byte4 address_byte5 address_byte6 address_byte7`

Parameters:

Name	Description
address_byte1	1 of 7 bytes that will make up the address
address_byte2	2 of 7 bytes that will make up the address
address_byte3	3 of 7 bytes that will make up the address
address_byte4	4 of 7 bytes that will make up the address
address_byte5	5 of 7 bytes that will make up the address
address_byte6	6 of 7 bytes that will make up the address
address_byte7	7 of 7 bytes that will make up the address

Examples:

```
sas set_sas_address 0x65 23 0x33 0x24 15 0x6 97
```

D.248 sas set_speed

Command Name(s): `sas set_speed`

Description: Sets the speed of the SAS interface (0 = autodetect, 1 = 1.5 Gbps, 3 = 3.0 Gbps, 6 = 6.0 Gbps).

Default Parm Order: `speed`

Parameters:

Name	Description
speed	Interface speed

Examples:

```
sas set_speed
```

D.249 sata comreset

Command Name(s): `sata comreset`

Description: This command will send a COM reset to the current port and hold it for the specified time. If the -ns switch is present, time will interpreted in nanoseconds, and if the switch is not present, time will be interpreted in seconds. If time is not specified, a default of 10 milliseconds is assumed.

Default Parm Order: `?-ns?  ?time?`

Parameters:

Name	Description
-ns	Optional: Treats time as nanosecond delay
time	Optional: Time to hold comreset

Examples:

```
    sata comreset
sata comreset -ns 1000
```

D.250 sata fis

Command Name(s): `sata fis`

Description: This command will access to the value of D2H register on the selected AHCI port. Append sata read with a command below to get the info. You can choose target port to obtain value by option of "-dev_idx".

Default Parm Order: `get`

Parameters:

Name	Description
`get`	Get value

Examples:

```
sata fis get count
sata fis get error -dev_idx 0
```

D.251 sata fis get

Command Name(s): `sata fis get`

Description: This command will access to the value of D2H register on the selected AHCI port. Append sata read with a command below to get the info. You can choose target port to obtain value by option of "-dev_idx".

Default Parm Order: `count lba lba_ext status error device ?-dev_idx?`

Parameters:

Name	Description
`count`	Count Register
`lba`	LBA Register
`lba_ext`	LBA_EXT Register
`status`	Status Register
`error`	Error Register
`device`	Device Register
`-dev_idx`	Optional: Device Index

Examples:

```
sata fis get count
sata fis get error -dev_idx 0
```

D.252 sata fis get count

Command Name(s): `sata fis get count`

Description: This command will access to the value of D2H register on the selected AHCI port. Append sata read with a command below to get the info. You can choose target port to obtain value by option of "-dev_idx".

Default Parm Order: `count lba lba_ext status error device ?-dev_idx?`

Parameters:

Name	Description
`count`	Count Register
`lba`	LBA Register
`lba_ext`	LBA_EXT Register
`status`	Status Register
`error`	Error Register
`device`	Device Register
`-dev_idx`	Optional: Device Index

Examples:

```
sata fis get count
sata fis get error -dev_idx 0
```

D.253 sata fis get device

Command Name(s): `sata fis get device`

Description: This command will access to the value of D2H register on the selected AHCI port. Append sata read with a command below to get the info. You can choose target port to obtain value by option of "-dev_idx".

Default Parm Order: `count lba lba_ext status error device ?-dev_idx?`

Parameters:

Name	Description
`count`	Count Register
`lba`	LBA Register
`lba_ext`	LBA_EXT Register
`status`	Status Register
`error`	Error Register
`device`	Device Register
`-dev_idx`	Optional: Device Index

Examples:

```
sata fis get count
sata fis get error -dev_idx 0
```

D.254 sata fis get error

Command Name(s): `sata fis get error`

Description: This command will access to the value of D2H register on the selected AHCI port. Append sata read with a command below to get the info. You can choose target port to obtain value by option of "-dev_idx".

Default Parm Order: `count lba lba_ext status error device ?-dev_idx?`

Parameters:

Name	Description
`count`	Count Register
`lba`	LBA Register
`lba_ext`	LBA_EXT Register
`status`	Status Register
`error`	Error Register
`device`	Device Register
`-dev_idx`	Optional: Device Index

Examples:

```
sata fis get count
sata fis get error -dev_idx 0
```

D.255 sata fis get lba

Command Name(s): `sata fis get lba`

Description: This command will access to the value of D2H register on the selected AHCI port. Append sata read with a command below to get the info. You can choose target port to obtain value by option of "-dev_idx".

Default Parm Order: `count lba lba_ext status error device ?-dev_idx?`

Parameters:

Name	Description
count	Count Register
lba	LBA Register
lba_ext	LBA_EXT Register
status	Status Register
error	Error Register
device	Device Register
-dev_idx	Optional: Device Index

Examples:

```
sata fis get count
sata fis get error -dev_idx 0
```

D.256 sata fis get lba_ext

Command Name(s): `sata fis get lba_ext`

Description: This command will access to the value of D2H register on the selected AHCI port. Append sata read with a command below to get the info. You can choose target port to obtain value by option of "-dev_idx".

Default Parm Order: `count lba lba_ext status error device ?-dev_idx?`

Parameters:

Name	Description
count	Count Register
lba	LBA Register
lba_ext	LBA_EXT Register
status	Status Register
error	Error Register
device	Device Register
-dev_idx	Optional: Device Index

Examples:

```
sata fis get count
sata fis get error -dev_idx 0
```

D.257 sata fis get status

Command Name(s): `sata fis get status`

Description: This command will access to the value of D2H register on the selected AHCI port. Append sata read with a command below to get the info. You can choose target port to obtain value by option of "-dev_idx".

Default Parm Order: `count lba lba_ext status error device ?-dev_idx?`

Parameters:

Name	Description
`count`	Count Register
`lba`	LBA Register
`lba_ext`	LBA_EXT Register
`status`	Status Register
`error`	Error Register
`device`	Device Register
`-dev_idx`	Optional: Device Index

Examples:

```
sata fis get count
sata fis get error -dev_idx 0
```

D.258 sata get

Command Name(s): `sata get`

Description: This command will read and display info on the desired part of selected(or targetted) ata device. Append sata get with a command below to get the info.

Default Parm Order: `status error control active ?-dev_idx?`

Parameters:

Name	Description
`status`	Status Register
`error`	Error Register
`control`	Control Register
`active`	Active Register
`-dev_idx`	Optional: Device Index

Examples:

```
sata get status -dev_idx 0
sata get error
```

D.259 sata get active

Command Name(s): `sata get active`

Description: This command will read and display info on the desired part of selected(or targetted) ata device. Append sata get with a command below to get the info.

Default Parm Order: `status error control active ?-dev_idx?`

Parameters:

Name	Description
`status`	Status Register
`error`	Error Register
`control`	Control Register
`active`	Active Register
`-dev_idx`	Optional: Device Index

Examples:

```
sata get status -dev_idx 0
sata get error
```

D.260 sata get control

Command Name(s): `sata get control`

Description: This command will read and display info on the desired part of selected(or targetted) ata device. Append sata get with a command below to get the info.

Default Parm Order: `status error control active ?-dev_idx?`

Parameters:

Name	Description
`status`	Status Register
`error`	Error Register
`control`	Control Register
`active`	Active Register
`-dev_idx`	Optional: Device Index

Examples:

```
sata get status -dev_idx 0
sata get error
```

D.261 sata get error

Command Name(s): `sata get error`

Description: This command will read and display info on the desired part of selected(or targetted) ata device. Append sata get with a command below to get the info.

Default Parm Order: `status error control active ?-dev_idx?`

Parameters:

Name	Description
`status`	Status Register
`error`	Error Register
`control`	Control Register
`active`	Active Register
`-dev_idx`	Optional: Device Index

Examples:

```
sata get status -dev_idx 0
sata get error
```

D.262 sata get status

Command Name(s): `sata get status`

Description: This command will read and display info on the desired part of selected(or targetted) ata device. Append sata get with a command below to get the info.

Default Parm Order: `status error control active ?-dev_idx?`

Parameters:

Name	Description
`status`	Status Register
`error`	Error Register
`control`	Control Register
`active`	Active Register
`-dev_idx`	Optional: Device Index

Examples:

```
sata get status -dev_idx 0
sata get error
```

D.263 sata get_auto_tags

Command Name(s): `sata get_auto_tags`

Description: This command will return whether the driver should automatically choose a free tag for NCQ commands.

Default Parm Order: `<None>`

Examples:

```
sata get_auto_tags
```

D.264 sata get_clear_ncq_err

Command Name(s): `sata get_clear_ncq_err`

Description: This command will return whether the driver should automatically clear a NCQ command error.

Default Parm Order: `<None>`

Examples:

```
sata get_clear_ncq_tags
```

D.265 sata get_cmd_fencing

Command Name(s): `sata get_cmd_fencing`

Description: This command will return whether the driver will wait to issue non-NCQ commands until all issued NCQ commands finish.

Default Parm Order: `<None>`

Examples:

```
sata get_cmd_fencing
```

D.266 sata get_connected_drive_mask

Command Name(s): `sata get_connected_drive_mask`

Description: This command will return the information that which port is connected the test drives. Each bit corresponds to a port number, where bit 0 corresponds to command port 0. For example, if a drive is connected to port 0, the value will be 1(2 $\hat{0}$), if a drive is connected to port 1, the value will be 2(2$\hat{1}$). If both of port 1 and 2 are connected, the value will be 3(= 2$\hat{0}$ | 2$\hat{1}$).

Default Parm Order: `?ahci_id?`

Parameters:

Name	Description
ahci_id	Optional: Target ahci id. It should be 0 3, The default value is 0

Examples:

```
sata get_connected_drive_mask 0
```

D.267 sata get_ncq_sequencing

Command Name(s): sata get_ncq_sequencing

Description: This command will return whether the driver will wait to issue new NCQ commands until all issued NCQ commands are started.

Default Parm Order: <None>

Examples:

```
sata get_ncq_sequencing
```

D.268 sata get_preserve_tags

Command Name(s): sata get_preserve_tags

Description: This command will return whether the driver will assign a new command tag (0) or use the UIL command tag (1).

Default Parm Order: <None>

Examples:

```
sata get_preserve_tags
```

D.269 sata get_selected_port

Command Name(s): sata get_selected_port

Description: Gets selected drive's port number. When the command is not selected or drives are not found it will return -1.

Default Parm Order: <None>

Examples:

```
sata get_selected_port_num
```

D.270 sata get_signature

Command Name(s): `sata get_signature`

Description: This command will return PxSIG(Port context signature) value. PxSIG contains value of first count register and LBA register value infirst D2H immediately after start drive. The 0 7 bit offset is count register, and 8 31 bit offset is the LBA value.

Default Parm Order: `<None>`

Examples:

```
sata get_signature
```

D.271 sata get_speed

Command Name(s): `sata get_speed`

Description: Gets the speed of SATA Drive.

Default Parm Order: `<None>`

Examples:

```
sata get_speed
```

D.272 sata get_starting_tfd

Command Name(s): `sata get_starting_tfd`

Description: This command will return TFD(Task File Data) value when spin up. TFD contains value of error and status registers.The 0x0 - 0x7 bits are status value, 0x8-0xF bits are error value.if the drive has not spun up, it will return 0x7F(Initial value of TFD).

Default Parm Order: `<None>`

Examples:

```
sata get_staritng_tfd
```

D.273 sata phy_disable_offline

Command Name(s): `sata phy_disable_offline`

Description: Disables SATA Phy ports they are offline (used to isolate offline devices).

Default Parm Order: `<None>`

Examples:

```
sata phy_disable_offline
```

D.274 sata phy_enable_disabled

Command Name(s): `sata phy_enable_disabled`

Description: Enable SATA Phy ports that are disabled (used to reconnect isolated devices).

Default Parm Order: `<None>`

Examples:

```
sata phy_enable_disabled
```

D.275 sata pm

Command Name(s): `sata pm`

Description: This command is used to changed the interface sleep state sata pm active . Return interface to active state sata pm partial . Set interface to partial state sata pm slumber . Set interface to slumber state.

Default Parm Order: `mode`

Parameters:

Name	Description
`mode`	The Mode can be "active", "partial", or "slumber"

Examples:

```
sata pm active
sata pm slumber
```

D.276 sata pm aggressive

Command Name(s): `sata pm aggressive`

Description: This Command is used to change the hba's aggressive sleep state sata pm aggressive off . Turn off HBA aggressive PM sata pm aggressive partial . Turn on HBA aggressive PM to the partial state sata pm aggressive slumber . Turn on HBA aggressive PM to the slumber state.

Default Parm Order: `mode`

Parameters:

Name	Description
mode	The Mode can be "off", "partial", or "slumber"

Examples:

```
    sata pm aggressive slumber
sata pm aggressive off
```

D.277 sata read_port_regs

Command Name(s): `sata read_port_regs`

Description: This command will read and display the port registers (Status, Error, and Sata Control).

Default Parm Order: `?-dev_idx?`

Parameters:

Name	Description
-dev_idx	Optional: Device Index

Examples:

```
sata read_port_regs -dev_idx 0
```

D.278 sata set_auto_tags

Command Name(s): `sata set_auto_tags`

Description: This command will set whether the driver should automatically choose a free tag for NCQ commands.

Default Parm Order: `state`

Parameters:

Name	Description
state	Can be "on", "off", 0, 1

Examples:

```
sata set_auto_tags on
```

D.279 sata set_clear_ncq_err

Command Name(s): `sata set_clear_ncq_err`

Description: This command will set whether the driver should automatically drop the queue and issue a read_log_ext to log 0x10 when an error occurs in NCQ.

Default Parm Order: `state`

Parameters:

Name	Description
`state`	1 or on to enable, 0 or off to disable

Examples:

```
sata set_clear_ncq_tags on
```

D.280 sata set_cmd_fencing

Command Name(s): `sata set_cmd_fencing`

Description: This command will set whether the driver will wait to issue non-NCQ commands until all issued NCQ commands finish.

Default Parm Order: `state`

Parameters:

Name	Description
`state`	1 or on to enable, 0 or off to disable

Examples:

```
sata set_cmd_fencing 1
```

D.281 sata set_ncq_sequencing

Command Name(s): `sata set_ncq_sequencing`

Description: This command will set whether the driver will wait to issue new NCQ commands until all issued NCQ commands are started.

Default Parm Order: `state`

Parameters:

Name	Description
state	1 or on to enable, 0 or off to disable

Examples:

```
sata set_ncq_sequencing 1
```

D.282 sata set_preserve_tags

Command Name(s): sata set_preserve_tags

Description: This command will set whether the driver will assign a new command tag (0) or use the UIL command tag (1).

Default Parm Order: state

Parameters:

Name	Description
state	1 or on to enable, 0 or off to disable

Examples:

```
sata set_preserve_tags 1
```

D.283 sata set_speed

Command Name(s): sata set_speed

Description: input speed rate by speed code.speed codes are as follows 0 : No speed negotiation restriction 1 : 1.5Gbps 2 : 3Gbps 3 : 6Gbps after do this command, you should do "COMRESET".

Default Parm Order: speed speed

Parameters:

Name	Description
speed	Interface speed
speed	Interface speed

Examples:

```
sata set_speed 2
```

D.284 sata soft_reset

Command Name(s): `sata soft_reset`

Description: This command will issue a soft reset by toggling the SRST bit in the control register.

Default Parm Order: `<None>`

Examples:

```
    sata srst
sata soft_reset
```

D.285 sata srst

Command Name(s): `sata srst`

Description: This command will issue a soft reset by toggling the SRST bit in the control register.

Default Parm Order: `<None>`

Examples:

```
    sata srst
sata soft_reset
```

D.286 scsi abort

Command Name(s): `scsi abort`

Description: This function sends a low-level abort SCSI message to the device. Clearly, this command is intended for SCSI devices only. See also "scsi abort_tag".

Default Parm Order: `<None>`

Examples:

```
    scsi abort
```

D.287 scsi abort_tag

Command Name(s): `scsi abort_tag`

Description: This function sends a low-level abort tag SCSI message to the device. Clearly, this command is intended for SCSI devices only. See also "scsi abort".

Default Parm Order: `tagID`

Parameters:

Name	Description
tagID	ID tag to abort (0x00 - 0xFF)

Examples:

```
scsi abort_tag 0x02
```

D.288 scsi clear_queue

Command Name(s): scsi clear_queue

Description: This command clears all queued tasks from the drive.

Default Parm Order: <None>

Examples:

```
scsi clear_queue
```

D.289 scsi device_reset

Command Name(s): scsi device_reset

Description: Sends a SCSI device reset message to the current target. See also "device rescan", "scsi reset".

Default Parm Order: <None>

Examples:

```
scsi device_reset
```

D.290 scsi id_mode

Command Name(s): scsi id_mode

Description: This function changes the IDENTIFY message the initiator uses after a SCSI SELECT. Options include 0x00 (no identify sent), 0x80 (no disconnects allowed) and 0xC0 (disconnects are allowed). Attempting to use any other mode will return an error.

Default Parm Order: mode

Parameters:

Name	Description
mode	Options are: 0x00, 0x80, 0xC0

Examples:

```
scsi device_reset
```

D.291 scsi ppr_mode

Command Name(s): scsi ppr_mode

Description: This function changes when PPR (Parallel Protocol Request) negotiations will occur. Options include all (every command), check (after next check status), none (disallow), once (after next command only), required (if context requires), and target (if initiated by target). Attempting to use any other mode will return an error See also "scsi ppr_mode target".

Default Parm Order: mode

Parameters:

Name	Description
mode	Options are: all/check/none/once/required/target

Examples:

```
scsi ppr_mode
```

D.292 scsi ppr_mode_parms

Command Name(s): scsi ppr_mode_parms

Description: This command sets the desired Parallel Protocol Request (PPR) parameters to use in a ll subsequent PPR negotiations. It will also force a PPR negotiation to occur on the following selection, either I/O or microprogramming functions.

Default Parm Order: offset period width ppr_parms

Parameters:

Name	Description
offset	The offset for synchronous negotiations
period	The period for synchronous negotiations
width	The width, 1 or 2
ppr_parms	Parallel protocol request parameters

Examples:

```
scsi ppr_mode_parms 62 9 2 6
```

D.293 scsi reset

Command Name(s): `scsi reset`

Description: This function preforms a SCSI bus reset. See also "device rescan", "scsi device_reset".

Default Parm Order: `<None>`

Examples:

```
scsi reset
```

D.294 scsi sync_mode

Command Name(s): `scsi sync_mode`

Description: This function changes when synchronous negotiations will occur. Options include all (every command), check (after next check status), none (disallow), once (after next command only), required (if context requires), and target (if initiated by target). Attempting to use any other mode will return an error.

Default Parm Order: `mode`

Parameters:

Name	Description
`mode`	Options are: all/check/none/once/required/target

Examples:

```
scsi sync_mode target
```

D.295 scsi sync_mode_parms

Command Name(s): `scsi sync_mode_parms`

Description: This command will set the desired synchronous period and offset for use in all subsequent synchronous negotiations. It will also force a synchronous data transfer request negotiation to occur on the following selection, either I/O or microprogramming functions.

Default Parm Order: `offset period`

Parameters:

Name	Description
offset	The offset for synchronous negotiations
period	The period for synchronous negotiations

Examples:

```
scsi sync_mode_parms 16 12
```

D.296 scsi wide_mode

Command Name(s): scsi wide_mode

Description: This function changes when width negotiations will occur. Options include all (every command), check (after next check status), none (disallow), once (after next command only), required (if context requires), and target (if initiated by target). Attempting to use any other mode will return an error.

Default Parm Order: mode

Parameters:

Name	Description
mode	Options are: all/check/none/once/required/target

Examples:

```
scsi wide_mode target
```

D.297 scsi wide_mode_parms

Command Name(s): scsi wide_mode_parms

Description: This command will set the desired data width for use in all subsequent wide negotiations. It will also force a wide data transfer request negotiation to occur on the following selection, either I/O or microprogramming functions. A value of 2 for the width specifies that there should be no forced negotiation.

Default Parm Order: width

Parameters:

Name	Description
width	Options are 1 or 2

Examples:

```
scsi wide_mode_parms 2
```

D.298 sop get cdb_iu_type

Command Name(s): `sop get cdb_iu_type`

Description: This function returns what IU type is currently being used as the default when sending CDBs to a SOP device.

Default Parm Order: `<None>`

Examples:

```
sop get iu_type
```

D.299 sop set cdb_iu_type

Command Name(s): `sop set cdb_iu_type`

Description: This function sets the default wrapper for CDBs. Any CDBs issued to a SOP device will be sent as this IU type. Valid types are "limited_command", "command", and "extended_command". "limited_command" is the default type. The numerical value of the IU type is also accepted.

Default Parm Order: `type`

Parameters:

Name	Description
`type`	The IU type to use as the new default

Examples:

```
sop set iu_type command
```

D.300 transport_cdb get always_on

Command Name(s): `transport_cdb get always_on`

Description: Returns the current state of the transport_cdb always_on flag.

Default Parm Order: `<None>`

Examples:

```
transport_cdb get always_on
```

D.301 transport_cdb get desc_format

Command Name(s): `transport_cdb get desc_format`

Description: Returns the current state of the transport_cdb descriptor format flag.

Default Parm Order: `<None>`

Examples:

```
transport_cdb get desc_format
```

D.302 transport_cdb get pad_boundary

Command Name(s): `transport_cdb get pad_boundary`

Description: Returns the current state of the transport_cdb padding boundary.

Default Parm Order: `<None>`

Examples:

```
transport_cdb get pad_boundary
```

D.303 transport_cdb get padding

Command Name(s): `transport_cdb get padding`

Description: Returns the current state of the transport_cdb padding flag.

Default Parm Order: `<None>`

Examples:

```
transport_cdb get padding
```

D.304 transport_cdb get protocol

Command Name(s): `transport_cdb get protocol`

Description: Use this command to return the APT protocol used during a transport command.Either "DMA" or "PIO" will be returned.

Default Parm Order: `<None>`

Examples:

```
transport_cdb get protocol
```

D.305 transport_cdb set always_on

Command Name(s): `transport_cdb set always_on`

Description: Use this command to turn on the transport_cdb flag for every CDB issued.

Default Parm Order: `always_on`

Parameters:

Name	Description
`always_on`	Enable transport cdb with every command

Examples:

```
transport_cdb set always_on 1
```

D.306 transport_cdb set desc_format

Command Name(s): `transport_cdb set desc_format`

Description: The sns size of the 0xC3 option 3 command defaults to 32 bytes and fixed format. Enable descriptor sns format with this command.

Default Parm Order: `desc_format`

Parameters:

Name	Description
`desc_format`	Enable desc_format

Examples:

```
transport_cdb set desc_format 1
```

D.307 transport_cdb set pad_boundary

Command Name(s): `transport_cdb set pad_boundary`

Description: When padding flag is set, this command defines the padding boundary. Default is 512. The send/recv buffer will be padded, as will the amount of bytes sent/received.

Default Parm Order: `pad_boundary`

Parameters:

Name	Description
pad_boundary	Set padding boundary

Examples:

```
transport_cdb set pad_boundary 4
```

D.308 transport_cdb set padding

Command Name(s): `transport_cdb set padding`

Description: Enabling this flag will pad every transport_cdb command to the next 512 bytes. The send/recv buffer will by padded, as will the amount of bytes sent/received. This functionality is necessary for the Audacious chip set.

Default Parm Order: `padding`

Parameters:

Name	Description
padding	Enable padding of DMA cmds

Examples:

```
transport_cdb set padding 1
```

D.309 transport_cdb set protocol

Command Name(s): `transport_cdb set protocol`

Description: Use this command to set the APT protocol used during a transport command. Users can give a value of "DMA" or "PIO".

Default Parm Order: `protocol`

Parameters:

Name	Description
protocol	DMA or PIO

Examples:

```
transport_cdb set protocol DMA
```

D.310 uil count

Command Name(s): `uil count`

Description: This function returns the number of loaded UIL drivers. For more details about the drivers, uil list can be used. Also see "uil list".

Default Parm Order: `<None>`

Examples:

```
uil count
```

D.311 uil create

Command Name(s): `uil create`

Description: This function is used to create a new driver instance. If the creation is successful, the driver is added to the uil list and the uil information is returned. Any parameters given to this function after the driver name are passed into the driver initialization routine. The initialization parameters vary, some drivers may have none. Parameters supported by some drivers include "-loglevel=l", "-noscan", and "-scsiid=h:c:t:l" Creating multiple instances of a driver works for most drivers. Also see "uil info", "uil list".

Default Parm Order: `driver_name ?driver_parms?`

Parameters:

Name	Description
`driver_name`	Name of driver to create
`driver_parms`	Optional: Driver parameters

Examples:

```
uil create spti
```

D.312 uil get autosense

Command Name(s): `uil get autosense`

Description: This function returns a 1 if autosense is currently active. Zero is returned otherwise. If the current driver does not support the disabling of autosense, 1 is always returned. Also see "uil set autosense".

Default Parm Order: `<None>`

Examples:

```
uil get autosense
```

D.313 uil get buffsize

Command Name(s): `uil get buffsize`

Description: This function returns the memory allocation for the current driver's internal data buffer.

Default Parm Order: `<None>`

Examples:

```
uil get buffsize
```

D.314 uil get callback create

Command Name(s): `uil get callback create`

Description: This function returns the callback mapped to a specific uil command. Callbacks are code segments that are automatically called whenever a command is executed. Currently supported callbacks are create, remove and "set index". Also see "uil set callback".

Default Parm Order: `<None>`

Examples:

```
uil get callback "set index"
```

D.315 uil get callback remove

Command Name(s): `uil get callback remove`

Description: This function returns the callback mapped to a specific uil command. Callbacks are code segments that are automatically called whenever a command is executed. Currently supported callbacks are create, remove and "set index". Also see "uil set callback".

Default Parm Order: `<None>`

Examples:

```
uil get callback "set index"
```

D.316 uil get callback "set index"

Command Name(s): `uil get callback "set index"`

Description: This function returns the callback mapped to a specific uil command. Callbacks are code segments that are automatically called whenever a command is executed. Currently supported callbacks are create, remove and "set index". Also see "uil set callback".

Default Parm Order: `<None>`

Examples:

```
uil get callback "set index"
```

D.317 uil get err_info

Command Name(s): `uil get err_info`

Description: This function dumps to stdout or to a file the error information returned by ERR_STR_MESSAGE errInfo call.

Default Parm Order: `?outputfile?`

Parameters:

Name	Description
`outputfile`	Optional: File to dump error info to

Examples:

```
No Examples given
```

D.318 uil get filter

Command Name(s): `uil get filter`

Description: This function returns the filter flag of the current driver, currently only in the ASPI driver.

Default Parm Order: `<None>`

Examples:

```
uil get filter
```

D.319 uil get index

Command Name(s): `uil get index`

Description: This function returns the index of the currently active UIL driver. The uil info command can be used in place of this one if more detailed information is required. If the optional 'uil_driver' is provided, this function returns the index of 'uil_driver' or -1 if not loaded. Also see "uil count", "uil info", "uil list", "uil name", "uil set index".

Default Parm Order: `?uil_driver?`

Parameters:

Name	Description
`uil_driver`	Optional: The UIL driver to get the index of

Examples:

```
uil get index
uil get index wincil
```

D.320 uil get max_xfer_len

Command Name(s): `uil get max_xfer_len`

Description: This function gets the driver's max transfer bytes.

Default Parm Order: `<None>`

Examples:

```
uil get max_xfer_len
```

D.321 uil get mpio enabled

Command Name(s): `uil get mpio enabled`

Description: This function gets the mpio enabled setting for this driver.

Default Parm Order: `<None>`

Examples:

```
uil get mpio enabled
```

D.322 uil get mpio path_select

Command Name(s): `uil get mpio path_select`

Description: This function gets the mpio data path select setting for this driver.

Default Parm Order: `<None>`

Examples:

```
uil get mpio path_select
```

D.323 uil get mpio tm_path_select

Command Name(s): `uil get mpio tm_path_select`

Description: This function gets the mpio task management path select setting for this driver.

Default Parm Order: `<None>`

Examples:

```
uil get mpio tm_path_select
```

D.324 uil get speed

Command Name(s): `uil get speed`

Description: This function gets the interface speed for this driver. It will return an interface type as well as the speed that it is running at.

Default Parm Order: `<None>`

Examples:

```
No Examples given
```

D.325 uil get timeout

Command Name(s): `uil get timeout`

Description: This function gets the default timeout value for all devices. This default timeout value is the timeout assigned to a device upon a device rescan.

Default Parm Order: `<None>`

Examples:

```
# Get the default device timout
uil get timeout
```

D.326 uil get version

Command Name(s): `uil get version`

Description: This function gets the interface version for this driver.

Default Parm Order: `?-uil?`

Parameters:

Name	Description
-uil	Optional: Get the UIL version instead of the core tcluil library.

Examples:

```
No Examples given
```

D.327 uil info

Command Name(s): `uil info`

Description: This function returns a brief description about the current uil driver. Information returned includes the index of the current driver, the name of the current driver, the version of the driver and the number of devices that the driver currently has access to. Also see "uil list".

Default Parm Order: `?uilIndex?`

Parameters:

Name	Description
uilIndex	Optional: name of UIL index

Examples:

```
uil info
```

D.328 uil list

Command Name(s): `uil list`

Description: This function returns a list of the currently installed UIL drivers. Each entry shows the index, name, and version of the driver. The number of devices connected to each driver is also reported. Also see "uil info".

Default Parm Order: `<None>`

Examples:

```
uil list
```

D.329 uil load

Command Name(s): `uil load`

Description: This function is used to load a TCL "C" extension that needs access to the UIL. These types of extensions generally extend the TCL command set with high performance macro functions (such as CSO operations). One example of this command is used to add the serial commands to TCL. Also see "uil create".

Default Parm Order: `driver_name`

Parameters:

Name	Description
`driver_name`	Name of driver to load

Examples:

```
uil load SerialCmds.dll
```

D.330 uil message

Command Name(s): `uil message`

Description: This function can be used to send special, non standard messages to a driver. Potential applications for this including the setting and or checking of internal driver settings. This command is provided as a path to non standard driver features.

Default Parm Order: `message ?uil_index?`

Parameters:

Name	Description
`message`	String message to send to uil
`uil_index`	Optional: UIL index to send message to

Examples:

```
uil message get_delay
```

D.331 uil name

Command Name(s): `uil name`

Description: This function returns the name of the current driver or specified uil driver. This function is useful for determining which driver is currently running, for conditional code. It is also useful for determining what drivers are available. For maximum versatility, use this feature only when necessary. Also see "uil info".

Default Parm Order: `?index?`

Parameters:

Name	Description
index	Optional: Name of UIL index

Examples:

```
  # The bad way
   if {[uil name] == "wincil"} {
      #turn on hardware data         generation
      device set xfer_mode random
   }
   else {
      puts "using software data generation"
      buff fill rand 0 0 512
   }
   w10 [randlba]

# The better way
   if {[catch {device set xfer_mode random}]} {
      puts "using software data generation"
      buff fill rand 0 0 512
   }
   w10 [randlba]
```

D.332 uil remove

Command Name(s): `uil remove`

Description: This function uninitializes a driver and removes it from the uil list. Also see "uil list".

Default Parm Order: `index`

Parameters:

Name	Description
index	Index of UIL to remove

Examples:

```
  uil remove test
```

D.333 uil set autosense

Command Name(s): `uil set autosense`

Description: This function is used to activate/deactivate "autosense" for the current uil driver. When autosense is active (the default setting) the driver will automatically aquire / return sense information when a check condition

occurs. Note that some uil drivers operate in autosense mode only and do not allow autosense to be deactivated. Also see "uil get autosense".

Default Parm Order: `?enable?`

Parameters:

Name	Description
`enable`	Optional: 0 for off, 1 for on

Examples:

```
uil set autosense 1
```

D.334 uil set callback create

Command Name(s): `uil set callback create`

Description: This function is used to set a callback for a particular uil command. Each time the command is called, this callback is executed as code. Currently supported callbacks are "create", "remove", and "set index". Each of these callbacks correspond to the associated uil command. The variable uil_index is set to provide further information within the callback. Also note that it is generally a good idea to make sure a callback is unset before setting it yourself. To remove a callback, set it to "". Also see "uil get callback".

Default Parm Order: `callback`

Parameters:

Name	Description
`callback`	Code to execute

Examples:

```
uil set callback "set index" { puts "device $uil_index" is selected }
usi 0
#remove the callback
device set callback "set index" ""
```

D.335 uil set callback remove

Command Name(s): `uil set callback remove`

Description: This function is used to set a callback for a particular uil command. Each time the command is called, this callback is executed as code. Currently supported callbacks are "create", "remove", and "set index". Each of these callbacks correspond to the associated uil command. The variable uil_index is set to provide further

information within the callback. Also note that it is generally a good idea to make sure a callback is unset before setting it yourself. To remove a callback, set it to "". Also see "uil get callback".

Default Parm Order: `callback`

Parameters:

Name	Description
`callback`	Code to execute

Examples:

```
    uil set callback "set index" { puts "device $uil_index" is selected }
   usi 0
#remove the callback
device set callback "set index" ""
```

D.336 uil set callback "set index"

Command Name(s): `uil set callback "set index"`

Description: This function is used to set a callback for a particular uil command. Each time the command is called, this callback is executed as code. Currently supported callbacks are "create", "remove", and "set index". Each of these callbacks correspond to the associated uil command. The variable uil_index is set to provide further information within the callback. Also note that it is generally a good idea to make sure a callback is unset before setting it yourself. To remove a callback, set it to "". Also see "uil get callback".

Default Parm Order: `callback`

Parameters:

Name	Description
`callback`	Code to execute

Examples:

```
    uil set callback "set index" { puts "device $uil_index" is selected }
   usi 0
#remove the callback
device set callback "set index" ""
```

D.337 uil set index

Command Name(s): `uil set index, usi`

Description: This function sets the current uil driver index. Using this command will retarget subsequent CDB/ATA commands to the specified driver. Also see "uil count", "uil get index", "uil info", "uil list".

Default Parm Order: `index`

Parameters:

Name	Description
`index`	UIL index

Examples:

```
    #send some CDB/ATA commands to uil 0
uil set index 0
r10
w10
#now send to uil 1
uil set index 1
r10
inq
#now back to uil 0 (the shorthand way)
usi 0
r10
inq
```

D.338 uil set loglevel

Command Name(s): `uil set loglevel`

Description: This function sets the current loglevel for the current UIL. Higher loglevels will return more verbosity than lower loglevels. This verbosity can be used for anything from getting more information about executing commands to debugging the drivers themselves. Level 10 is considered to request a maximum level of verbosity. Note that in some environments, namely Windows, Niagara needs to be run from within a command prompt to see the outputted messages.

Default Parm Order: `level`

Parameters:

Name	Description
`level`	Logging level

Examples:

```
  uil set loglevel 4
```

D.339 uil set mpio enabled

Command Name(s): `uil set mpio enabled`

Description: This function sets the mpio enabled setting for this driver.

Default Parm Order: `enabled`

Parameters:

Name	Description
`enabled`	0 for disabled, 1 for enabled

Examples:

```
uil set mpio enabled 1
```

D.340 uil set mpio path_select

Command Name(s): `uil set mpio path_select`

Description: This function sets the mpio data path select setting for this driver.

Default Parm Order: `path_select`

Parameters:

Name	Description
`path_select`	round_robin, min_queue_depth, none, path_0, path_1, path_2, path_3
`path_select`	round_robin, min_queue_depth, none, path_0, path_1, path_2, path_3

Examples:

```
uil set mpio path_select min_queue_depth
```

D.341 uil set mpio tm_path_select

Command Name(s): `uil set mpio tm_path_select`

Description: This function sets the mpio task management path select setting for this driver.

Default Parm Order: `path_select`

Parameters:

Name	Description
path_select	round_robin, min_queue_depth, none, path_0, path_1, path_2, path_3
path_select	round_robin, min_queue_depth, none, path_0, path_1, path_2, path_3

Examples:

```
uil set mpio tm_path_select round_robin
```

D.342 uil set speed

Command Name(s): `uil set speed`

Description: This function sets the interface speed for this driver. Options for ?interface? are scsi, fcal, or serial.

Default Parm Order: `interface speed`

Parameters:

Name	Description
interface	Interface type
speed	Data xfer speed

Examples:

```
No Examples given
```

D.343 uil set timeout

Command Name(s): `uil set timeout`

Description: This function sets the command timeout value for all devices. Note that different UIL drivers support different units for timeout values, not all allow timing to millisecond accuracy. Therefore after setting a timeout, the actual timeout set is returned.

Default Parm Order: `time ?-override_persistent?`

Parameters:

Name	Description
time	Timeout in milliseconds
-override_persistent	Optional: Temporarily override the persistent timeouts set for each command. Use 0 to disable.
-override_persistent	Optional: Temporarily override the persistent timeouts set for each command. Use 0 to disable.

Examples:

```
# Set device timeout for 5 seconds
uil set timeout 5000
```

D.344 validate_commands

Command Name(s): `validate_commands`

Description: If you give a file name it will check validation of target command definition file, otherwise there are no command definition files, It will check commands which are already loaded in TCL interpreter.

Default Parm Order: `?filename?`

Parameters:

Name	Description
`filename`	Optional: Filename to check validation

Examples:

```
validate_commands $::env(NiagaraPath)/commands/cdb_commands.txt
```

Appendix E
Serial Commands

E.1 get_serial_list

Command Name(s): `get_serial_list`

Description: This function returns a list of available serial commands. The function is intended to be used by TCL programs (such as documentation generators) that need to know the current set of available commands.

Default Parm Order: `<No Parameters>`

See Also: `get_cdb_list` (page **??**), `get_ata_list` (page **??**), `get_cil_list` (page 265)

E.2 sabortCDB

Command Name(s): `sabortCDB`

Description: This function will send CDB abort command, aborting the CDB command that is currently in progress. It is used as a last resort when communication sync is lost and cannot be regained.

Default Parm Order: `<No Parameters>`

E.3 scdb

Command Name(s): `scdb`

Description: This function is used to send a UART2 cdb frame to the drive. The first ?length? bytes in the send buffer are used as the cdb to send.

Default Parm Order: `?length?`

Parameters:

Name	Description
`length`	Optional: Length of cdb to send

E.4 sclose

Command Name(s): `sindex`

Description: Close the serial port. Useful to allow external programs access to the serial port. Usage: sclose.

Default Parm Order: `<No Parameters>`

E.5 sdelay

Command Name(s): `sdelay`

Description: This function is used to set the serial comminication timing delay. Slower machines may need to increase the delay to keep comminication in sync. The delay is set in micro seconds.

Default Parm Order: `delay`

Parameters:

Name	Description
`delay`	Length of delay in micro seconds

E.6 secho

Command Name(s): `secho`

Description: This function requests an echo response from the serial device. The response contains the serial number of the drive. This function is useful to asserting a good serial connection with the drive.

Default Parm Order: `<No Parameters>`

E.7 sget_speed

Command Name(s): `sget_speed`

Description: This function requests a Get UART Parameters response from the serial device. This function will parse out the line speed from the return information.

Default Parm Order: `<No Parameters>`

E.8 sindex

Command Name(s): `sindex`

Description: Get/Set the suil device index. Usage sindex ?index?.

Default Parm Order: `?index?`

Parameters:

Name	Description
`index`	Optional: Optional: The device index you wish to select.

E.9 sio

Command Name(s): `sio`

Description: This command allows low level UART communication on a byte-by-byte basis. The send length and recieve length are specified; then, ?send_length? number of bytes are sent from the send buffer and ?recv_length? number of bytes are read into the recieve buffer.

Default Parm Order: `?send_length?  ?recv_length?`

Parameters:

Name	Description
`send_length`	Optional: number of bytes to send through UART
`recv_length`	Optional: number of bytes to recieve through UART

E.10 slip

Command Name(s): `slip`

Description: This function causes the drive to request LIP if it is a Fibre Channel drive. The drive will send a response block to the host indicating the command is acknowledged and then call the appropriate function to request LIP.

Default Parm Order: `?LIP type?`

Parameters:

Name	Description
`LIP type`	Optional: type of LIP requested (F7 or F8)

E.11 squery

Command Name(s): `squery`

Description: This function will send a UART query command to the drive. This command will be used by the host when it is polling the drive to see if it is ready for the next command.

Default Parm Order: `<No Parameters>`

E.12 sread

Command Name(s): `sread`

Description: This function returns a portion of memory from the current serial device. Information is also read into the current receive buffer where it can be manipulated further by "buff" commands. Note that, unlike Serial Debugger, you need to append 0x to all parameters you wish to be in hex.

Default Parm Order: `address length ?-dw? ?-dd? ?-qw? ?-transport_cdb? ?-show_status?`

Parameters:

Name	Description
`address`	Address in memory to read
`length`	Number of bytes to read
`-dw`	Optional: Display data in 16-bit word format
`-dd`	Optional: Display data in 32-bit double word format
`-qw`	Optional: Display data in 64-bit quad word format
`-transport_cdb`	Optional: If "-transport_cdb 1", send command as a 0xC3 transport command
`-transport_cdb`	Optional: If "-transport_cdb 1", send command as a 0xC3 transport command
`-show_status`	Optional: To skip status output append "-show_status 0" to the command
`-show_status`	Optional: To skip status output append "-show_status 0" to the command

See Also: `buff dump` (page 203), `buff save` (page 221), `buff format` (page 213), `swrite` (page 350)

E.13 sreadsp

Command Name(s): `sreadsp`

Description: This function returns the stack pointer location for the current device.

Default Parm Order: `<No Parameters>`

E.14 srescan

Command Name(s): `srescan`

Description: Requests that the current suil performs a device rescan. Note: This skips any device rescan binds.

Default Parm Order: `<No Parameters>`

E.15 sreset

Command Name(s): `sreset`

Description: The command initiates a reset on the drive. The drive will send a response block to the host indicating that the command is acknowledged and then call the appropriate reset function.

Default Parm Order: `?command flags? ?drive type?`

Parameters:

Name	Description
command flags	Optional: command flags that specify the type of reset
drive type	Optional: drive type byte

E.16 sset_speed

Command Name(s): sset_speed

Description: This function requests a Get UART Parameters response from the serial device, it then modifies the line speed section, sends a Set UART Parameters request, and changes the host speed to match the new speed.

Default Parm Order: coded_speed ?-quick_scan?

Parameters:

Name	Description
coded_speed	The coded speed value. See UART3 Specification.
-quick_scan	Optional: quick scan the bus.

E.17 sspeed

Command Name(s): sspeed

Description: Get/Set the serial speed for suil in bps. It is valid for both serail/serial3 drivers. Usage sspeed ?speed?.

Default Parm Order: ?index?

Parameters:

Name	Description
index	Optional: The speed in bits per second (bps)

E.18 sstatus

Command Name(s): sstatus

Description: This function is only valid for the serial3 driver.

Default Parm Order: <No Parameters>

E.19 suart2

Command Name(s): suart2

Description: This function is used to set the serial communication link to UART2.

Default Parm Order: <No Parameters>

E.20 suart3

Command Name(s): `suart3`

Description: This function is used to set the serial communication link to UART3. The speed value determines the line speed that will be used in the new mode. See the UART protocol definition for values.

Default Parm Order: `speed`

Parameters:

Name	Description
`speed`	UART speed setting

E.21 suil

Command Name(s): `suil`

Description: This command sets the UIL target index that serial commands (sread, etc...) will be directed at. Serial commands have an independent uil setting for convenience purposes. Note that if you are sending CDBs through the serial port, you should use "uil set index" to set your driver.

Default Parm Order: `?index?`

Parameters:

Name	Description
`index`	Optional: UIL Index (without index, suil returns the current index)

See Also: `uil set index` (page 340)

E.22 sversion

Command Name(s): `sversion`

Description: This function requests that the drive report back to the host the version of the UART interface it supports. A drive that supports the UART 2 interface will return the Version response block with the version field set to at least 2, while a drive that does not will either return an error on the transmission (legacy versions) or return the response block with a different value (future versions). UART 3 devices will return the response block with a value of at least 5.

Default Parm Order: `<No Parameters>`

E.23 swrite

Command Name(s): `swrite`

Description: This function is used to write data to memory in the drive. Data can be entered directly as part of the swrite command or the current send buffer can be used (if no data is specified, the send buffer is used). All entered parameters (including data) must be preceeded by a 0x if the data is in hex format. The "buff" commands can be used to assist in pre-buffer setup (such as loading a buffer with contents from a file).

Default Parm Order: `address length ?-dw?  ?-dd?  ?hex data?`

Parameters:

Name	Description
`address`	Address in memory to read
`length`	Number of bytes to read
`-dw`	Optional: Command line data is in 16-bit word format
`-dd`	Optional: Command line data is in 32-bit double word format
`hex data`	Optional: One or more space separated bytes of write data

See Also: `buff load` (page 217), `buff fill` (page **??**), `sread` (page 347)

E.24 sxfer

Command Name(s): `sxfer`

Description: This command sends a UART2 data xfer command block to the drive. The type of data xfer is defined by the parameters used.

Default Parm Order: `?Transfer Bit?  ?Status Bit?  ?Send Length?  ?Recv Length?`

Parameters:

Name	Description
`Transfer Bit`	Optional: direction of data transfer (0 inbound, 1 outbound)
`Status Bit`	Optional: 1 indicates transfer is for reporting status
`Send Length`	Optional: number of bytes to send
`Recv Length`	Optional: number of bytes to recv

Index