The NSCL Scripted DAQ Reference guide

Ron Fox (fox@nscl.msu.edu)

Table of contents:

CHAPTER 1 7

GENERAL APPROACH 7

Configuration Management issues and goals 8
The Hardware Description File 9
Application Configuration files 10

CHAPTER 2 12

THE HARDWARE DESCRIPTION FILE 12

Basic Concepts 12
Syntax 13
Hierarchy of the CAEN 16/32 channel digitizer modules 17
The CAENVxxx Base module 17
The CAENV785 Module extensions to CAENVxxx 19
The CAENV792 module extensions to CAENVxxx 19
The CAENV775 module extensions to CAENVxxx 19
The CAENV830 scaler module 20
The packet module 22

CHAPTER 3 24

THE READOUT CONFIGURATION FILE 24

The readout command 24
The scalerbank command 25

CHAPTER 4 27

THE SCALER DISPLAY CONFIGURATION FILE 27

The Channel Command 28
The Page Command 29
The Display Commands 30

CHAPTER 5 32

THE SPECTCL CONFIGURATION FILE 32

The unpack command 33
The parameter command 34
The spectrum command 35
The sbind command 37

Table of Figures:

Figure 1 Relationship between the acquisition, analysis and monitoring components. 7
Figure 2 Common hardware configuration files with application specific files................. 9
Figure 3 Structure of a packet.. 22
Figure 4 SpecTcl's event processing model.. 32

Table of Examples:

Example 1 Creating a module. 12
Example 2 Storing the configuration of a module for further processing. 13
Example 3 Configuring a module 13
Example 4 Sample syntax definition 13
Example 5 Syntax of the module and scaler command. 14
Example 6 Creating a module 14
Example 7 Listing defined modules. 14
Example 8 Sample module -list command and output 15
Example 9 Processing the output of a module list. 15
Example 10 Using the module's command. 15
Example 11 Configuring a module 15
Example 12 Sample configuration list 16
Example 13 Specifiying a packet. 23
Example 14 Syntax of the readout command. 24
Example 15 One way to add three modules to the readout list. 25
Example 16 A second way to add three modules to the readout list. 25
Example 17 Using the **readout list** command 25
Example 18 The **channel** command syntax 28
Example 19 Creating scaler channels from the hardware.tcl file. 28
Example 20 Syntax of the **page** command. 29
Example 21 Creating 10 pages with silly titles 30
Example 22 Syntax of the **display_single** command. 30
Example 23 Syntax of the **display_ratio** command. 30
Example 24 Syntax of the unpack command. 33
Example 25 Partial parameter command syntax. 34
Example 26 Sample parameter definition 34
Example 27 Format of spectrum -list output 34
Example 28 Sample output from parameter -list, and using it programmatically 35
Example 29 Deleting a parameter definition. 35
Example 30 The syntax of the **spectrum** command 35
Example 31 Creating spectra.. 36
Example 32 Output from Spectrum -list and using it in a script. 37
Example 33 Deleting the spectrum e1 37
Example 34 Syntax of the **sbind** commnd. 38
Example 35 sbind -list and using the output of sbind –list 38

Table of Tables

Table 1 Supported module types.. 14
Table 2 Configuration value types.. 17
Table 3 Common configuration parameters for the CAEN 32 channel digitizers............ 19
Table 4 Additional configuration parameters implemented by the CAEN V775 TDC.... 20
Table 5 Configuration parameters supported by the CAEN V830 scaler......................... 22
Table 6 packet module configuration options.. 23

Chapter 1

General Approach

This chapter describes the general approach to the configuration of the acquisition, monitoring and analysis components of the system. These components are named, respectively:

- Readout, responds to event triggers and reads out events.
- Scaler, accepts incremental scaler data from the online system and produces customizable displays of rates an totals.
- SpecTcl, accepts event data, histograms it, and allows you to interact with the analysis of the data.

The relationship between these programs is shown in Figure 1 below:

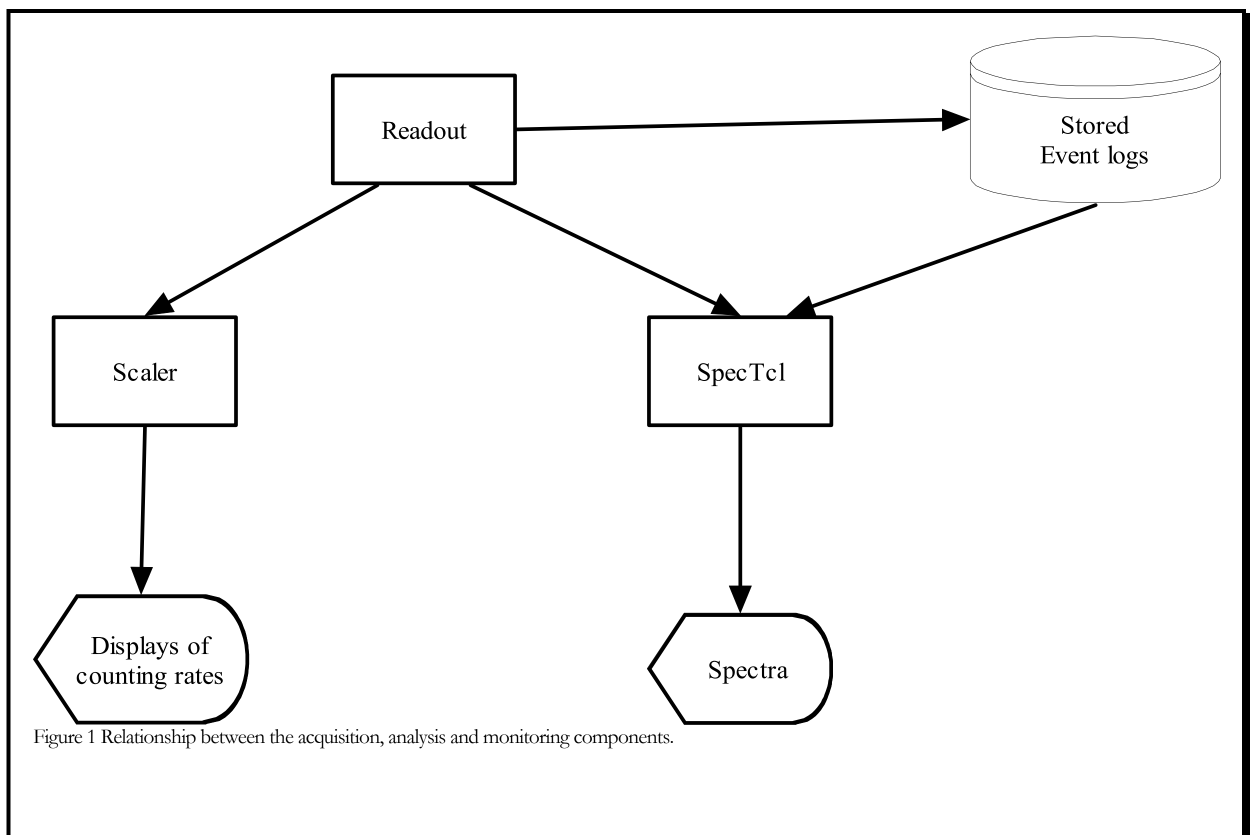


Figure 1 Relationship between the acquisition, analysis and monitoring components.

Readout produces event data. This event data is shipped in three directions:

- Stored event logs are produced that may be replayed by SpecTcl for offline analysis.
- A sample of the online event data are sent to SpecTcl which analyzes only as much data as it can without slowing down the data acquisition.
- Scaler receives a specialized subset of the data that comprises periodic scaler values. These are displayed in highly customizable screens containing rates totals, and optionally ratios of scalers and scaler pairs.

The remainder of this chapter describes:

- The design tension between the common data that all of these programs operate on against their needs to treat their handling of this data in different ways according to differing configuration needs.
- How this is managed via a common hardware configuration file that all applications use as the basis for their configuration.
- How each application has a secondary application specific configuration file that describes how the data described in the common hardware configuration file is handled.

Configuration Management issues and goals

Clearly all three components, Acquisition, Monitoring, Analysis require common configuration information. In the final analysis, there is experimental hardware. The experimental hardware has input channels. These input channels define what is read, the structure of events and the structure of scaler data. Each of the DAQ components must treat this information differently:

Readout needs to know how to configure the hardware, prepare it for readout, and what to read for each trigger. In summary, Readout must take the hardware configuration information and turn it into an event structure. Readout must also take the hardware configuration information and turn it into a set of actions that condition the hardware to take data appropriately.

Scaler requires knowledge of the correspondence between scaler channels in the hardware and the data it will see in scaler buffers. This information is present in the hardware configuration, but in addition, the scaler display program will need to know how to present scaler information to the user.

SpecTcl requires intimate knowledge of the hardware channels, and how they have been configured so that it can deduce the structure of the event in order to decode events into raw experimental parameters. In addition, SpecTcl will need information about the size (in bits) of these parameters as well as how to produce meaningful spectra from them. In more advanced

applications, SpecTcl will also need to know about sets of conditions and how they have been applied to spectra to produce conditionally incremented, or *gated* spectra.

The design of the configuration subsystem must resolve the design tension between an underlying common, hardware description, and the functional differentiation of the job that each program performs. The NSCLDAQ acquisition system resolves this design tension by splitting the configuration of each application into two separate files. A hardware description file describes the hardware, and its experimental configuration. Application specific configuration files describe how this information is used by each application.

This is shown schematically in Figure 2.

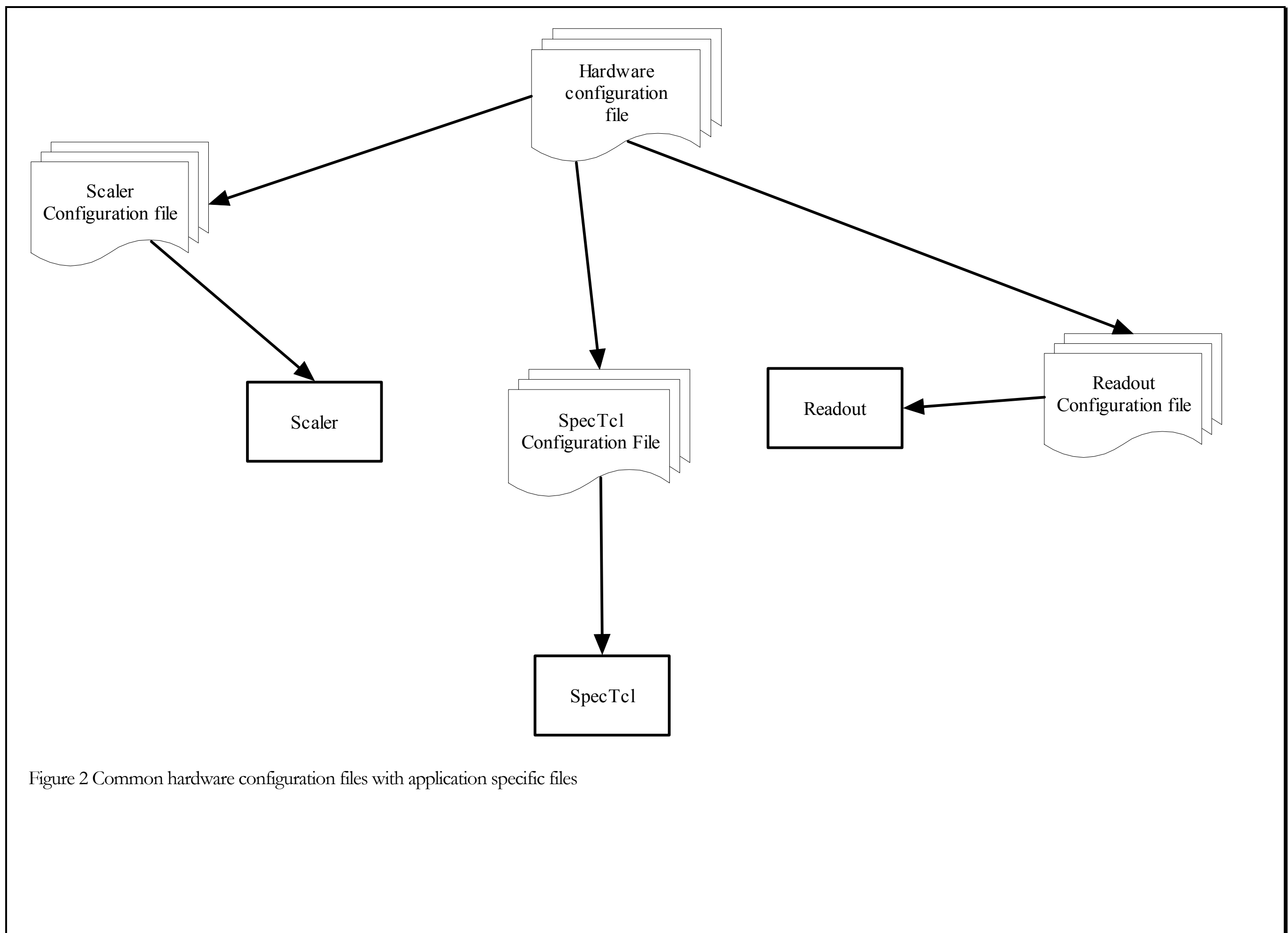


Figure 2 Common hardware configuration files with application specific files

This picture shows that each application specific configuration file includes a common hardware configuration file and augments it. Since the configuration files are written in an extension language to Tcl, this is easily done via the Tcl *source* command.

The Hardware Description File

The core of the acquisition analysis and monitoring configuration is a hardware configuration file. This file creates a "module" for each hardware device. The module is then provided with configuration parameters. These parameters describe how the software should configure the

hardware. The configuration parameters also provide textual names for each input channel. The application specific configuration file uses these names and the hardware modes of the module to deduce the structure of the data that are produced by the hardware. These names carry forward to produce parameter names (in the case of SpecTcl), and scaler channel names (in the case of Scaler).

The hardware description file is described completely in Chapter 5. It is called ~/config/hardware.tcl where ~ means the user's home directory. You can use the full power of the Tcl scripting language to create or modify this hardware configuration file.

Application Configuration files

Each application has its own application specific configuration file. The application specific configuration file includes the hardware description file via the Tcl *source* command. Each application specific configuration file interrogates the data produced by the hardware configuration file and turns it into an application specific configuration.

Readout's configuration file, sources the hardware configuration file. It's extended Tcl interpreter reacts to module commands by building up a set of device driver objects for each module defined and accumulating configuration parameters in a module specific configuration database. The readout configuration file specifies which modules are to be readout and in which order. It does this by querying the Readout program about the defined modules and scalers. Modules are then added to a "readout list" and a "scaler list". The order in which the modules are read is determined by the order in which they are added to the list.

When a run begins, the Readout program iterates through the readout list and the scaler list. Each module's Initialization method is invoked. The initialization queries the module's configuration database, and configures the hardware appropriately. When an event trigger is detected, the readout list is iterated and each module's read method is called to collect data from the module. If the software has declared a periodic scaler event, the scaler list is iterated and each module's read method is called to add scaler data to the scaler data packet.

The Scaler program's configuration file sources the same hardware configuration file. It's extended Tcl interpreter ignores the event data module definitions and only looks at scaler definitions. It uses the knowledge of the scaler readout order in conjunction with the channel parameter configuration to build a map between scaler channel names and offsets into the scaler data packet.

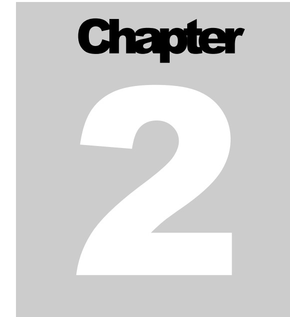
Additional commands in the scaler configuration file define:

- Display pages
- Display lines within the display pages.

SpecTcl's configuration file also sources the same hardware configuration file. In this case, the module commands produce software objects that know how to unpack the data from each module. The configuration database is created as for the Readout program, but the configuration

parameters are used to develop criteria for recognizing data from these devices in the event and subsequently decoding these data into raw parameters as specified by the device parameter configuration.

The SpecTcl configuration file must define additional parameter characteristics. It must also create an initial set of spectra, and bind them to the Xamine display program.



Chapter 2

The Hardware Description File

The hardware description file is a configuration file that describes the hardware and how to configure it. This chapter describes:

- The concepts behind the extensions to TCL that produce the configuration file language.
- The syntax of the *module* and *scaler* commands as well as a generic syntax for the commands that configure a module.
- The set of configuration parameters that each module type recognizes..

Basic Concepts

The extensions to TCL adopted for the hardware description file form an object based system. Two commands, *module* and *scaler* create objects that represent modules of particular types and their configurations (in object oriented terms, these commands are constructors).

Objects in the hardware description file are represented by commands. Thus the *module* and *scaler* command creates new commands. Those new commands in turn implement subcommands that allow you to configure, and examine the configuration of the module. For example:

```
module adc caenv785
```

Example 1 Creating a module.

Creates a module named adc. The module represents the configuration of a CAEN V785 peak sensing adc. The command shown in Example 1 also creates a new command *adc* that can be invoked to configure and inspect the configuration of this hardware module. Example 2 shows how to take the configuration of *adc* and store it in a Tcl variable for further processing:

```
set adcConfig [adc cget]
```

Example 2 Storing the configuration of a module for further processing.

The stuff inside the square brackets is a command *adc cget* the output of this command is substituted on the command line as a parameter. For any object representing a module, the *cget* subcommand returns the configuration of the module.

A module can be configured via its command as well:

```
adc config  base 0x400000
```

Example 3 Configuring a module

Example 3 configures the adc so that its base address is 0x400000. This tells the software that drives the adc to expect it to be at address 0x400000 in the VME address space. The *config* subcommand of any module command supplies module specific configuration information (in this case the base address) to the object.

The *module* and *scaler* commands are identical. The only difference is that objects created with *module* can be placed in Readout's readout list and SpecTcl's unpacker list while objects created with *scaler* can be placed in Readout's scaler readout list and registered with the scaler display program as having interesting channels.

Syntax

Now that we have seen samples of hardware configuration file commands, we will examine the detailed syntax of the *module,* and *scaler* commands. We will also look at the subcommands that all objects must implement. In the syntax descriptions below | is used to separate alternatives and italic case is used to describe a parameter that must be selected by you. For example:

```
module | scaler  name type
```

Example 4 Sample syntax definition

Example 4 describes a command that can consist either of the keyword **module** or the keyword **scaler** followed by a name and type that must be chosen by you. {}'s will be used to surround larger elements (they should not appear in the command itself. []'s will be used to surround optional elements (again, they should not appear in the command itself). A backslash \ is used to indicate that the syntax continues on the next line.

Having set the stage, the syntax of the module/scaler command is:

```
module | scaler   {name type [configuration-list]} | {-list [pattern]} | {-delete name } |
-types
```

Example 5 Syntax of the module and scaler command.

This syntax underscores the fact that the module and scaler commands are essentially the same command. The module/scaler command has four sub forms (from now on in examples I will use module for brevity). The first:

```
module name type [configuration-list]
```

Example 6 Creating a module

Creates a new module object. The module object has a name *name* and a module type *type*. The name of the module will also become the *module command*. The module command is used to operate on the hardware that is represented by the new object. The *type* parameter determines the type of module that is being created. The following module types are supported by the NSCLDAQ system.

Type name	**Hardware Type**
caenv775	A CAEN V775 32 channel multievent TDC.
Caenv792	A CAEN V792 32 channel QDC.
caenv785	A CAEN V785 32 channel multievent peak sensing ADC.
caenv830	A CAEN V830 32 channel multievent latching scaler. Note that this module is the only module type available to the scaler command.
packet	Create a grouping of modules that is optionally preceded by a word-count and identifying word.

Table 1 Supported module types

We will discuss the *configuration-list* parameter when we describe the module name command.

The second form of the module/scaler command is:

```
module –list [pattern]
```

Example 7 Listing defined modules.

This command will list the modules that have been created. Note that the module names and scaler names form distinct module namespaces. The optional *pattern* parameter specifies a "glob"

pattern that can be used to select the set of modules to list. The default pattern is "*", which matches all module names. "Glob" patterns can include wildcard characters like * that match any sequence of characters and ? which matches any single character.

The result of module –list is returned in a format that makes it easy to process by additional TCL commands: A list of items, each item in the list is a two item sub list containing the module name and type. New line characters separate the outer list items. For example:

```
module –list *dc
{adc caenv785}
{tdc caenv775}
```

Example 8 Sample module -list command and output

A sample snipped of TCL code that will process this list to produce a human readable output is:

```
set modules [module –list]
foreach module $modules {
   puts "Module named [lindex $module 0] is of type [lindex $module 1]"
}
```

Example 9 Processing the output of a module list.

Note how the foreach command is used to iterate over modules and then the lindex command is used to extract individual elements of each module's information. This is a recurring theme in the application specific configuration files.

Once a module has been created it can be manipulated via the module's command. The module's command is the same as the module name. For example:

```
module adc caenv785
adc cget
```

Example 10 Using the module's command.

In Example 10 a module named adc is created that is a CAEN V785 peak sensing ADC. The second command asks the module to return its configuration. The full syntax of the module command is:

```
name {config configuration-list …} | cget | help
```

Example 11 Configuring a module

The module command uses subcommands to determine the function that should be performed the subcommands are *config*, *cget*, and *help*. The subcommand that you will use to configure a module is *config*. This subcommand uses a set of parameters called a *configuration-list*. A *configuration-list* is a set of parameter keyword-value pairs. For example:

```
module adc caenv785
adc config base 0x400000  geo false
```

Example 12 Sample configuration list

In Example 12 above, the configuration list is "base 0x400000 geo false". This list contains two parameter keyword-value pairs. The parameter keyword describes what will be configured; the value describes how to configure the parameter. In Example 12, the first keyword-value pair "base 0x400000" sets the base address parameter to 0x400000, the second keyword-value pair "geo false" indicates that the module should not be set up using geographical addressing.

Each module supports a set of keywords that is specific to the module type. Keyword values have an associated data type. Table 2 describes the data types that can be taken by a value:

Data Type	**Meaning**	**Examples**
integer	A single integer number. The number can be supplied in decimal, hexadecimal (base 16) or octal (base 8) format.	15 - Decimal 15. 0x15 – hex 15 = decimal 21 015 - Octal 15 = 13.
bool	A Boolean value *true* or *false*	true - Set the parameter true/on false - Turn the parameter off.
string	A string of characters. Strings that contain spaces must be quoted with "" or {}. See TCL books for a description of the difference between these two types of quotation.	"yellow" - The string **yellow** "blue boy" – the string **blue boy** {red $book} – The string **red $book**
int array	A fixed length list of integers. The list can be enclosed in "" or {}, depending on whether or not you want to suppress variable substitution (see TCL books for more on this).	"1 2 3 4" – 4 element array {1 2 3 4} – Same 4 element array.
string array	A fixed length list of strings. The list can be enclosed in "" or {} depending on whether or not you want to suppress variable	"a b c" - 3 element array { a b $c} - 3 element array, but the third element is $c not the

	substitution. You must additionally quote items that have spaces in them.	value of the variable c {a b "have space"} also a three element array.

Table 2 Configuration value types

In the documentation that follows, we will list the set of configuration keywords each module accepts along with the data type of each associated parameter.

Hierarchy of the CAEN 16/32 channel digitizer modules

The CAEN V785 peak sensing ADC, CAEN V792 QDC and CAEN V775 TDC are all built on the same base module. Configuration support can therefore be thought of as a set of configuration parameters that are common to all CAENVxxx modules and additional configuration support that is specific to an individual module type. In the description of the configuration parameters that follows, we will first describe the base caenvxx configuration parameters. Next we will describe the extensions to that base, supported by each specific module type.

The CAENVxxx Base module

The CAENVxxx module implements the configuration subsystem for the CAEN V785 and CAENV775 digitizers. The base hardware fore these modules is a 32-channel peak sensing ADC. Daughter boards such as a time to peak converter or a charge to peak converter allow the base module to be marketed as a TDC and a QDC respectively.

CAEN also sells versions of these modules with NIM inputs. Due to front panel space limitations, the NIM input versions only implement 16 channels of the base module. It is important to know that the 16 channel units implement the *even* channel numbers (0,2,4,6), etc. When configuring channels on these modules, configure every other channel.

Table 3 describes the configuration parameters that are managed by all CAENVxxx modules. The table shows the name of the configuration keyword, the type of keyword value it expects, the meaning of the keyword-value pair and the default value for the keyword (what happens if you don't ever configure with this keyword).

Keyword	Parameter Type	Meaning	Default
crate	integer	Selects, which VME crate in a multi-crate system, houses the module. The parameter value must be in the range [0,7].	0 (This is the only crate in a single crate system).

slot	integer	Selects the geographical address of the module. This parameter interacts with the geo parameter. If geo is true, then this parameter must be the physical slot that houses the module. If geo is false, this parameter is used to program the module's geo address. Note that modules that support geographical addressing must use it, as their geo address is not programmable. This value must be in the range [1,21]	-1 – this parameter must be specified as it's initial value is illegal. The slot number is used to uniquely identify the data from this module
threshold	integer array	This 32-element array of integers defines the conversion thresholds for each channel of the adc. Remember that for 16 channel units the even channels are implemented. Each element of the array must be in the range [0,4000] and is in units of mv.	0
keepunder	bool	This parameter determines whether or not the module will suppress conversions that were below threshold.	False – suppress below threshold conversions
keepoverflow	bool	This parameter determines whether or not the module will suppress conversions that are overflows.	False – Supress overflow conversions.
card	Bool	This parameter determines whether or not the device is enabled. The module can be completely disabled. In that case it does not react to gates, and never supplies data.	True –module is enabled.
geo	bool	This parameter indicates whether or not the module will be configured via geographical addressing. Geographical addressing requires CERN JAUX VME back plane support, as well as the versions of the digitizer that support this option	true
enable	int array	This 32-element array allows you to control the individual channel enable mask. Each element of the array is either 0 (corresponding channel disabled), or 1 (corresponding channel enabled). Remember that for the 16	Array full of 1's all channels are enabled.

		channel units, the even channels are implemented.	
base	int	Configures the module's base address. This parameter is ignored if geo is true. If geo is false, this parameter value must reflect the module base address as stored set by the module's rotary switches.	0
multievent	bool	If false, the module is cleared after each read. If true, the module is assumed to be running in a system that allows it to buffer multiple events, in that case, the module is not cleared after being read.	False – Clear after each read.
parameters	string array	32-element array. Each element of the array provides a meaningful parameter name to the corresponding channel.	32 empty strings.
fastclearwindow	int	The fast clear window for the module (refer to the hardware documentation for more information).	0

Table 3 Common configuration parameters for the CAEN 32 channel digitizers.

The CAENV785 Module extensions to CAENVxxx

At this time, the CAEN V785 does not implement any additional configuration parameters.

The CAENV792 module extensions to CAENVxxx

The CAENV792 implements a single additional configuration parameter named "Iped". Iped takes an integer in the range [0,255] and programs the compensation current register. The purpose of the compensation register is to compensate for charge lost on the input stage of the QDC during the time that the gate is open. Refer to the hardware manual for more information about this.

The CAENV775 module extensions to CAENVxxx

The additional configuration parameters implemented by the CAEN V775 are shown in Table 4 below.

Keyword	Parameter Type	Meaning	Default
range	int	The full-scale conversion range of the module. This is in units of ns. Legal	500ns.

		values are in the range [140,1200]. Note that there are limits to the resolution with which the full-scale conversion range can be set. These are described in the CAENV775 manual. The configuration software will load the hardware with the nearest legal value.	
commonstart	bool	When true the module operates in common start mode. When false, common stop mode.	True.

Table 4 Additional configuration parameters implemented by the CAEN V775 TDC

Again note, that these configuration parameters are implemented in addition to those defined in Table 3.

The CAENV830 scaler module

The CAEN V830 is a 32-channel multi-event latching scaler module. It can be used in either the event-by-event readout (e.g. in conjunction with an oscillator to provide a time base). It can also be used as a periodic scaler. Which it is used as depends on whether or not it was defined via the **scaler** or **module** command.

The configuration parameters supported by the CAEN V830 are shown in Table 5 below. The table shows the parameter keyword, the type of the parameter, describes each parameter and shows the default value for each parameter.

Keyword	Parameter Type	Meaning	Default
base	int	Base address of the module. This only has meaning if geo is false.	0
slot	int	The geographical address for the module. If geo is true, and the crate supports geographical addressing, this must be number of the physical slot in which the module is inerted. If geo is false, this slot address is programmed into the module.	2
crate	int	In a multi-VME crate system, the number of the VME create in which the module is installed.	0
geo	bool	Determines whether or not the module is setup via geographical addressing	True
enables	int array	Array of 32 integers. Each integer corresponds to a channel in the module.	All 1's.

		A 1 enables the corresponding channel, while a 0 disables it.	
trigger	int	An integer value that represents the module trigger mode. The following values are supported: 0. Trigger is disabled. 1. Random triggering 2. Periodic trigger.	1 Random trigger.
wide	bool	True if the readout is wide (32 bits per scaler)	False (26 bit readout) This allows channels to be tagged with a channel number.
header	bool	If true insert the module header in the data read from the module. The format of the header word is described in figure 3.1 of the module documentation.	true
autoreset	bool	If true, triggers reset the counters to zero when the data are latched. Note that some firmware revisions fail to perform this function correctly.	True
fpclearmeb	bool	If true, the multi-event buffer is cleared on a front panel reset.	False
manualclear	bool	If true, the scaler counters are cleared by software after a aread. See autoreset.	False
packetize	Bool	If true, additional information is generated in software for the read. This information makes automated decoding of these data possible by the software.	False
vmetrigger	bool	Enable VME trigger if true, otherwise, front panel trigger is enabled.	False
parameters	String array	An array of 32 strings. Each string corresponds to a channel. The strings document the names of the channel inputs. SpecTcl and Scaler use these strings to build histograms and scaler displays.	32 empty strings.

id	int	If packetize is true, this is the id of the packet that is created for the scaler. Valid ids are in the range [0, 65535].	0.

Table 5 Configuration parameters supported by the CAEN V830 scaler.

The packet module

In larger experiments it is convenient to be able to impose a high level, hierarchical structure on the event data. At the NSCL this structure takes the form of *packets*. A packet is a subset of the event that has the form:

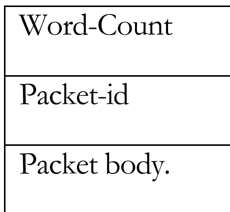


Figure 3 Structure of a packet.

The packet id together with the word-count supports modular experiments:

- Software can be written to pay attention only to the packets of interest, skipping those packet ids that are either not interesting or not known.

- Given a unique managed set of packet ids, experiments can be composed from several pieces of apparatus. The packet ids provide unique identifiers that enable you to distinguish between data from different apparatus. This has been successfully used in runs that combine the SeGA array with the S-800 spectrograph. Daniel Bazin (bazin@nscl.msu.edu) manages the top level packet ids at the NSCL.

- This event structure model can be extended hierarchically (nesting sub packets within packets and other sub packets) to create self-describing events.

The packet module type provides support for this event formatting. A packet module, in addition to implementing the *config, cget,* and *help* subcommand, supports subcommands that allow you to describe which modules are contained within the packet; *add, remove*, and *list.*

Before turning to these additional subcommands let's look at the configuration options supported by packet modules:

Name	**Type**	**Contains**	**Default**
packetize	bool	If true, the modules in the packet will be wrapped inside a packet (count/id words). When used with the readout program this generates the packet wrapper. When used with SpecTcl, this indicates a packet wrapper needs	false

		to be decoded.	
Id	int	If packetized is true, this value is the id of the packet. The value must be a positive integer in the range [0,65535]	None.

Table 6 packet module configuration options.

The purpose of a packet is to organize groups of modules into larger elements that can be treated as a single module and, optionally/usually, wrapped inside of a packet. To support this, packet module support additional subcommands. These subcommands are:

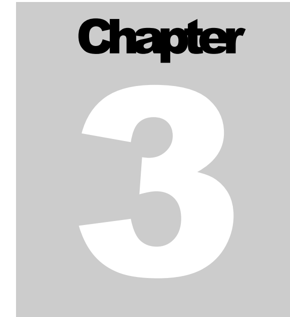
- *add* allows you to insert modules into the packet. Modules are always inserted at the end of the packet.
- *remove* allows you to remove a module from a packet if it is no longer required.
- *list* allows you to list the set of modules in a packet.

Example 13 shows a simple use of the packet module. A CAENV785, and CAENV775 adc and tdc are created and placed in a packet. The packet is configured to wrap the modules in a the word-count id structure of Figure 3 above.

```
module adc caenv785 slot 2
module tdc caenv775 slot3

module p packet id 0x1000 packetize true
p add adc tdc
```

Example 13 Specifiying a packet.



Chapter 3

The Readout configuration file

The Readout configuration file uses information in the hardware definition file (~/config/hardware.tcl) and additional commands provided by the user to configure the Readout program. Items that are configured in the NSCL readout program are:

- The order in which modules are read when an event fires.
- The order in which periodic scaler modules are read.

The readout configuration file is the command line argument to the Readout program. The readout configuration file is processed at the beginning of each run. If you make a change to this file it is not necessary to restart the readout program. Simply end and begin the run. In the event of an error in the configuration file, the readout program will report this error to the readoutGUI. Simply correct the file, stop and start the run again.

The readout command

The readout software uses the **readout** command to describe the order in which modules will be read when an event trigger is received. The form of the readout command is:

> readout {add *module* [*module.* …]} | {list [*pattern*]} | {remove *module*} {configure *params*}

Example 14 Syntax of the readout command.

Note that in essence, the readout command implements a packet as described in the section *The packet module* in Chapter 2 above. It accepts all the configuration options and subcommands as the packet module. Think of the readout command as the 'top level packet' of an experiment.

In the syntax above, *module* represents the name of a module created with the **module** command. Modules created with the **scaler** command cannot be added to the readout list.

The **add** syntax allows you to add a list of modules to the end of the readout list. Given modules named **mod1, mod2,** and **mod3,** the following two script fragments do the same thing:

```
readout add mod1
readout add mod2
readout add mod3
```

Example 15 One way to add three modules to the readout list.

```
readout add mod1
readout add mod2 mod3
```

Example 16 A second way to add three modules to the readout list.

In Example 14, *pattern* is a pattern that can contain wild card characters like * and ?. If omitted, the pattern defaults to *. The list subcommand returns a list of modules whose names match the pattern. The modules are listed in the order they are read out. The return list is a valid Tcl list. Each element is a two-element sub list containing the module name and type. The following script fragment produces a human readable module listing:

```
set read [module -list]
puts "Module     Type"
puts "=========="
foreach module $read {
    set name [lindex $module 0]
    set type [lindex $module 1]
    set line [format "%6s   %6s" $name $type]
    puts $line
}
```

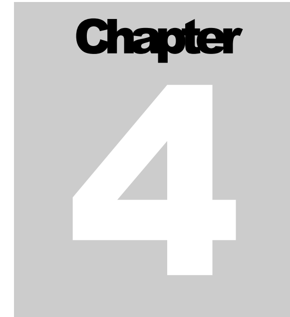
Example 17 Using the **readout list** command

The **module remove** subcommand removes a module from the read list. Note that if a module is deleted via **module –delete** it is automatically removed from the read list as well.

The scalerbank command

Periodically, the Readout program reads a set of scalers. The modules that can be read by this are defined by the **scaler** command in the hardware configuration file. The order in which these scaler modules are read is defined by the **scalerbank** command.

The syntax of the **scalerbank** command is identical to that of the **readout** command. The only difference is that only modules defined via the **scaler** command can be added to the scaler readout list. The **scaler** command, in turn, only allows you to create caenv830 modules. This prevents you from accidentally reading a module that is not a scaler at the periodic scaler readout time.



Chapter 4

The Scaler Display configuration file

The NSCLDAQ software includes support for periodically reading scalers and displaying rates, totals and ratios of selected scaler pairs. Applications of this subsystem include:

- Monitoring detector rates to ensure that beam intensities are not so high that detector pile-up or electronics pile-up is a problem.
- Monitoring the number of events that occurred in nature vs. the number of events that the acquisition system was able to accept (live time ratio).

The Scaler display program uses a scaler configuration file to define its display. The scaler configuration file uses the hardware configuration file to determine the scaler channels and to configure the display program to be able to extract specific channels from scaler buffers.

The scaler configuration file does this by asking **scalerbank** to list the set of scalers that will be read out, and then interrogating each module for the names of the channels configured into it via the parameters configuration option.

The scaler configuration file is read when the scaler program starts. It is therefore necessary to restart the scaler program if you make changes to its configuration file

The scaler display program manages a set of pages. These pages are displayed in a *tabbed notebook*. A page may be displayed by clicking it's tab. You can define as many pages as you want and place arbitrary combinations of single scalers and scaler ratios on lines in each page of the notebook.

The scaler program extends the TCL interpreter by adding the commands:

- channel – makes a correspondence between a named channel and a slot in scaler buffers.
- page – defines a page on the scaler display.
- display_single – defines a scaler display page line that contains a single scaler channel.
- display_ratio – defines a scaler display page line that contains a pair of scaler channels and their ratio.

The Channel Command

Scaler buffers have a fixed format. In order to display scaler channels, the display program must establish a correspondence between scaler values in this buffer and named channels that you specify in display_single and display_ratio commands. The **channel** command is used to create this correspondence. The syntax of the **channel** command is:

```
channel name offset
```

Example 18 The **channel** command syntax

name is the name given to a channel, *offset* is the offset into the block of scalers at which this channel can be found. The nice thing about having a central hardware configuration file is that the channel names can be determined from this file, and the offsets to them can be calculated automatically. The script fragment in Example 19 below shows how to do this:

```
set scalers [scaler -list ]
#  Procedure to get a parameter list from a scaler module.
proc getParams {module} {
    set config [$module cget]
    set params ""
    foreach item $config {
        set configname [lindex $item 0]
           if {$configname == "parameters"} {
           set channels [lindex $item 1]
           foreach channel $channels {
               if {$channel != ""} {
                   lappend params $channel
               }
           }
        }
    }
    return $params
}

foreach scaler $scalers {
    set name [lindex $scaler 0]
    set channels [getParams $name]
    set index 0;                        # Channel number.
    foreach channel $channels {
       channel $channel $index
       incr    index
    }
}
```

Example 19 Creating scaler channels from the hardware.tcl file.

The Page Command

The **page** command creates a scaler display page. A scaler display page is a page in a tabbed notebook. It has the following elements:

- A verbose title that appears at the top of the page when it is displayed.
- A brief title that appears on the tab that is used to select the page for display.
- A set of display lines that are defined via the **display_single** and **display_ratio** commands.

Any number of pages can be created. The syntax of the **page** command is

```
page brief-title verbose-title
```

Example 20 Syntax of the **page** command.

The *brief-title* should not have any spaces. The *verbose-title* should be enclosed in either quotes or curly brackets {} if it has spaces. The following script fragment creates 10 pages numbered 0-9:

```
for {set p 0} {$p < 10} {incr p} {
   page Page$p "This is page number $p"
```

Example 21 Creating 10 pages with silly titles.

The Display Commands

The **display** command creates scaler display lines on a scaler page. Two types of scaler display lines can be created:

- *Single* lines display the rates and total counts for a single scaler value. Single lines are created using the **display_single** command.
- *Rato* lines display the rates totals and ratios of rates and totals for a pair of scalers; a *numerator-channel* and a *denominator-channel*. Ratio lines are created using the **display_ratio** command.

Lines are displayed on a page in the order in which they are created.

The syntax of the **display_single** command is:

```
display_single brief-title channel
```

Example 22 Syntax of the **display_single** command.

In this command, *brief-title* is the brief title of the display page on which the scaler will be displayed. *channel* is the name of the channel to display. *brief-title* is defined by the **page** command. *channel* is defined via the **channel** command.

The syntax of the **display_ratio** command is:

```
display_ratio brief-title numerator-channel denominator-channel
```

Example 23 Syntax of the **display_ratio** command.

In this command:

- *brief-title* is the brief title of the page defined in the **page** command.
- *numerator-channel* is the name of the channel (as defined by the **channel** command) that will be displayed as the ratio's numerator. The numberator channel is also the left of the pair of channels in the individual channel display columns.

- *Denominator-channel* is the name of the channel (as defined by the **channel** command) that will be the denominator of the ratio. The denominator channel is also the rightmost of the pair of channels displayed in the individual channel display columns.

Chapter
5

Chapter 5

The SpecTcl Configuration File

SpecTcl is an advanced, highly configurable, and customizable event sorting/histogramming program. SpecTcl can be programmed in C++, or in Tcl/Tk or a mixture of the two. In this chapter, we will describe the extensions to Tcl implemented by SpecTcl's configuration file, and how to use these to obtain histograms of raw parameters.

In order to understand SpecTcl's configuration file, it is important to understand SpecTcl's data

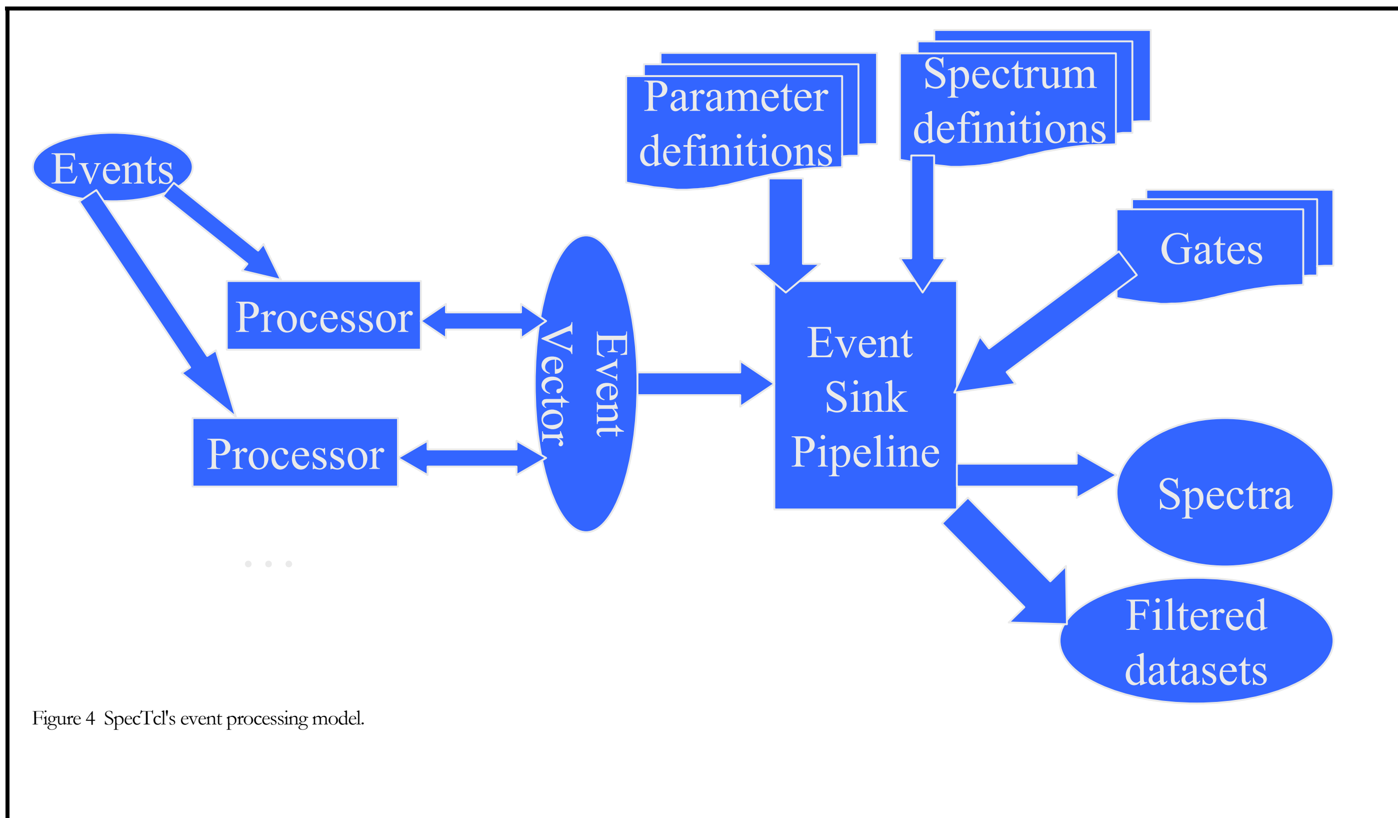


Figure 4 SpecTcl's event processing model.

processing model. This model is shown in Figure 4.

Events enter the system in some raw format. Event processors are invoked sequentially to *unpack* events into a list of parameters of interest, the *Event Vector*. The event vector is passed to an *event sink pipeline*. The main element of the event sink pipeline is a histogrammer. The histogrammer

uses parameter, spectrum and gate definitions to produce spectra that are displayed by the Xamine display program.

The SpecTcl configuration file must configure the first Event processor so that it can automatically unpack the raw event into the event vector. Subsequent event processors can operate on the parameters in the event vector to produce computed parameters.

SpecTcl implements three commands that are important to the configuration file:

- **unpack** is the inverse of the Readout program's **readout** command. It registers an unpacker for a segment of the event.

- **parameter** defines the properties of a SpecTcl parameter. Parameters are named entities that can be histogrammed.

- **spectrum** defines a spectrum. SpecTcl supports a rich variety of histogram types. In this document we will only describe 1-d and 2-d histograms.

- **sbind** makes a spectrum viewable in Xamine

Full SpecTcl documentation is available online at:

http://docs.nscl.msu.edu/

Click on the SpecTcl link.

The unpack command

The SpecTcl configuration file sources the hardware definition file. The modules that are defined and configured there create unpackers for SpecTcl. An unpacker knows how to decode the event data from a particular hardware module. The parameter configuration for each module creates a correspondence between channels in the module and SpecTcl parameters (see next section The parameter command). The syntax for the unpack command is:

unpack {add *module* [*module.* …]} | {list [*pattern*]} | {remove *module*}

Example 24 Syntax of the unpack command.

The *add* subcommand allows you to add a module to the unpacker list. Modules must be added to the unpacker list in the same order in which they were added to the Readout's readout list.

The *list* subcommand produces a list of the modules that are currently in the unpacker list. The list has the same format at the Readout configuration file's *readout list* command.

The *delete* subcommand removes a named module from the unpacker list.

Note that the syntax for the **unpack** command is essentially identical to that of the Readout **readout** and **scalerbank** commands.

The parameter command

The **parameter** command defines the properties of a SpecTcl parameter. The default SpecTcl configuration file queries the modules for their parameter names and creates a reasonable set of default parameters. A subset of the **parameter** command syntax is shown in Example 25 below. For full documentation see the SpecTcl documentation:

```
parameter {name id resolution} | -list | {-delete name}
```

Example 25 Partial parameter command syntax.

The first form of the command creates a new parameter definition:

- *name* is the name of the parameter and, for raw parameters, should be the name of a channel in a module.
- *id* is an integer parameter id you assign to the parameter. All parameter id's must be unique. The parameter id is the index into the event vector of the parameter.
- *resolution* an integer that represents the number of bits of resolution of the parameter. (e.g. a CAEN V785 provides 12 bit digitized values, therefore this number is 12 for all modules).

For example:

```
parameter  energy1 1 12
```

Example 26 Sample parameter definition

The second form of the parameter command *parameter –list* lists the set of defined parameters. The command actually returns the parameter list as its value making it possible to capture this list and manipulate it within Tcl scripts. Each parameter definition is a single list item and is itself a list of the form:

```
name id resolution mapping-information
```

Example 27 Format of spectrum -list output

In Example 27 above, the elements of this list have the following meanings:

- *name* is the parameter name.
- *id* is the id number of the parameter
- *Resolution* is the resolution in bits of the parameter

- *Mapping-information* is information that maps the parameter space to a calibrated real parameter space. In the examples in this manual, this is a list containing three empty sub lists.

Example 28 below shows a parameter definition and the resulting listing for it. It also shows how to produce a human readable parameter list. The bold face lines were typed by a human, the normal faced type by the computer.

```
parameter george 12
george
parameter –list
{george 100 12 {{} {} {}}}
foreach parameter [parameter –list] {
  puts [format "%s  %d  %s" [lindex $parameter 0] \
                            [lindex $parameter 1]  \
                            [lindex $parameter 2]]
}
george   100   12
…
```

Example 28 Sample output from parameter -list, and using it programmatically

The final form of the **parameter** command deletes parameter definitions. If you delete a parameter definition for a parameter that is bound to a module channel, that parameter is not unpacked and will not be analyzed. It is still stored in the event file for the run, and may be analyzed off-line. Example 29 below shows how to delete the sample parameter from Example 28.

```
parameter –delete george
```

Example 29 Deleting a parameter definition.

The spectrum command

The **spectrum** command creates, lists and deletes spectra. Spectra are defined on sets of parameters. SpecTcl supports many spectrum types. In this manual, we will only describe how to create one-dimensional and two-dimensional spectra. We will also only talk about simple integer channeled spectra. Refer to the online SpecTcl documentation at http://docs.nscl.msu.edu/ to learn more about SpecTcl and the spectra it supports.

The syntaxes of the **spectrum** command we will describe are shown in Example 30 below.

```
spectrum  {name type parameter-list axis-speclist} | {-list [name]} | {-delete name}
```

Example 30 The syntax of the **spectrum** command

The first form of the command creates a new spectrum. The parameters of this form are:

- *name* – the name of the spectrum. Each spectrum must have a unique name. The name can be any combination of characters, however white space and characters special to tcl must be appropriately quoted or escaped.
- *type* – the type of spectrum being created. In this manual we will only describe the types:
    - **1** – One dimensional spectrum
    - **2** – Two dimensional spectrum.
- *parameter-list* – A tcl list containing the parameters that will be histogrammed. For a one-dimensional histogram, this is the name of a single parameter. For a two-dimensional histogram, this is a two-element list containing the parameter on the X-axis followed by the parameter on the Y-axis.
- *Axis*-speclist – A list of axis specifications. In this manual, we will only describe the axis specifications that consist of bits of resolution. In the case of a 1-dimensional spectrum, the single axis specification is an integer that represents the number of bits of resolution the spectrum X-axis has. For a two dimensional spectrum, this is a list containing a pair of integer resolutions.

Example 31 below creates two spectra. A 1-dimensional spectrum of the parameter e1 that has 1024 channels, and a 2-dimensional spectrum of e1 on the x-axis and e2 on the y-axis that is 512x512 channels:

```
spectrum e1 1 e1 10
spectrum e1-vs-e2 2 {e1 e2} {9 9}
```

Example 31 Creating spectra..

The second form of the spectrum command lists the characteristics of all of the spectra or, if a spectrum name is given, the characteristics of that single spectrum. The spectrum list is returned in a format that can be used by further TCL commands: A list of spectrum descriptions. Each description is a sub list containing:

- *Id* – A spectrum id that is used only internally within SpecTcl.
- *Name* – The name of the spectrum.
- *Type* – The spectrum type.
- *Parameters* – list of parameters on the spectrum axes.
- *Axis-speclist* – List of axis specifications in a canonical form.

- *Datattype* – the data type of each channel of the spectrum.

Example 32 shows:

- The list produced of the spectra definedor the spectra defined in Example 31.
- How to use this list to produce a user readable listing of the spectra and their parameters.

User input is **bold-face** while computer output is in normal type.

```
spectrum –list
{1 e1 1 {e1} {{0 1024 1024}} long}
{2 e1-vs-e2  2 {e1 e2} {{0 512 512} {0 512 512} word}
foreach spect [spectrum –list] {
   set name [lindex $spect 1]
   set type   [lindex $spect 2]
   set params [lindex $spect 3]
   if {$type == 1} {
      puts [format "%s    %s" $name  $params]
   } else {
       puts [format "%s %s vs  %s" $name [lindex $params 0] [lindex $params
1]]
   }
}
e1    e1
e1-vs-e2  e1 vs e2
```

Example 32 Output from Spectrum -list and using it in a script.

The final form of the spectrum command deletes a spectrum definition:

```
spectrum –delete e1
```

Example 33 Deleting the spectrum e1

A deleted spectrum is removed from SpecTcl. If it was bound to the Xamine display program, it is unbound. If it was visible in the Xamine window it will disappear.

The sbind command

SpecTcl supports the creation and maintenance of an arbitrary number of histograms with size limited only by the available virtual memory on the system. The Xamine display program, however only supports a fixed number of channels of storage and a limited (but large) number of spectra. The **sbind** command makes spectra available for display by Xamine, by binding them in to Xamine's spectrum memory. **sbind** copies the channels that make up a spectrum into

Xamine's memory, removes the old channels and directs SpecTcl to continue incrementing channels directly in Xamine's memory for bound spectra.

The **sbind** command has the syntax shown in <example>

```
sbind {name [name …]} | -all | -list
```

Example 34 Syntax of the **sbind** commnd.

The first form of thi sbind command binds a set of spectra in to Xamine's display memory. The second form attempts to bind all unbound spectra into Xamine's display memory.

The final form of the sbind command lists the set of spectra that are currently bound to Xamine's display memory. The bindings are returned as a TCL list. The elements of the list are:

- *Id* - The SpecTcl id for the spectrum (same id as seen in spectrum –list).
- *Name* – The name of the spectrum.
- *Xid* – The slot number in the Xamine spectrum memory that has been assigned to the spectruml.

<Example> below shows sample output from **sbind –list**. It also shows how to use this output to list each bound spectrum and its type. User input is **bold** compute output is in normal typeface.

```
sbind –list
{0 e1 1}
{01 e1-vs-e2 0}
foreach binding [sbind –list] {
   set name [lindex $binding 1]
   set info    [spectrum –list $name]
   set type   [lindex $info 2]
   puts "$name is spectrum type $type"
}
e1 is type 1
e1-vs-e2 is type 2
```

Example 35 sbind -list and using the output of sbind –list

Note how the spectrum name is extracted from the sbind –list command and then used to get the spectrum description via the spectrum –list command.