VM-USB based Data Acquisition System

Ron Fox
VM-USB based Data Acquisition System
by Ron Fox

Revision History
Revision 1.0 November 25, 2007 Revised by: RF
Original Release
Revision 1.1 February 15, 2008 Revised by: RF
Describe SIS3804 support and installation from tarball
Revision 1.2 June 27, 2008
Describe MADC32 support and other STUK custom software
Revision 1.2-001 July 9, 2008
Add -nimbusy description to madc command, this exposes the programmability of the NIM output, and allows monitoring the internal
Revision 1.2-002 July 16, 2008
Fix sectioning issues with the SpecTcl plugin docs.
Revision 1.2-003 October 4, 2008
Add documentation for the CAENV1x90 support
Revision 1.2-004 August 27, 2009
Document Hytec timestamp and id tagging
Revision 1.2-005 September 28, 2009
Document udev stuff
Revision 1.3 November 25, 2009
Document adding event processors to SpecTcl
Revision 1.4 May 21, 2010
Document support for CAEN V977 input register.
Revision 1.5 August 31, 2010
Changes to madc command to support chaining and madcchain config command.
Revision 1.6 September 22, 2010
Document the mase command and how to configure parameters for IUCFMaseSpecTcl
Revision 1.7 November 26, 2010
• Add support for CAENV965 and other dual range CAEN modules. • Added support for vmusb control module for generalized
Revision 1.8 May 30, 2011
Added support for the CAEN V6533 HV control
Revision 1.8 October 8, 2011
Completed documentation for CAEN V1x90 TDCs
# Table of Contents

1. Introduction .................................................................................................................. 1
   1.1. Installing the software ............................................................................................ 1
   1.2. VM-USB description .............................................................................................. 2
   1.3. Supported Devices ................................................................................................. 3
   1.4. Script organization ............................................................................................... 3

2. Getting Started ............................................................................................................. 5
   2.1. Preparing a data acquisition account ................................................................. 5
   2.2. Data acquisition system components and how to start them ......................... 6
   2.3. Operating the NSCL DAQ components ............................................................... 7
      2.3.1. ReadoutGUI ..................................................................................................... 8
      2.3.2. SpecTcl ........................................................................................................ 11
      2.3.3. Xamine ......................................................................................................... 32
      2.3.4. ScalerDisplay ............................................................................................... 47
   2.4. Writing event data to DVD .................................................................................... 47

3. Configuration Files. ..................................................................................................... 48
   3.1. daqconfig.tcl The readout config file ............................................................... 48
   3.2. Configuring SpecTcl ............................................................................................ 48
      3.2.1. A note about the Hytec 2530 ADC ......................................................... 50
   3.3. Configuring the Scaler Display ......................................................................... 50
   3.4. Script Reference .................................................................................................. 52
      adc ......................................................................................................................... 52
      caenchain ............................................................................................................ 55
      sis3820 ............................................................................................................... 57
      v830 ...................................................................................................................... 58
      v977 ...................................................................................................................... 61
      sis3804 ................................................................................................................. 63
      hytec ................................................................................................................... 65
      madc ................................................................................................................... 69
      madcchain ........................................................................................................... 74
      madcscale ...................................................................................................... 75
      mase ................................................................................................................... 75
      tdevv1x90 ........................................................................................................... 78
      v1729a .............................................................................................................. 82
      stack .................................................................................................................. 84
      variables .......................................................................................................... 88

4. SpecTcl .......................................................................................................................... 92
   4.1. Using daqconfig.tcl to drive event decoding ................................................ 92
   4.2. SpecTcl Plugins ................................................................................................ 94
      4.2.1. The firstof Plugin ....................................................................................... 95
      4.2.2. The calibrated parameter plugin ............................................................. 97
5. Scaler Display ................................................................................................................................. 103
6. Converting event data to XML files .............................................................................................. 105
   6.1. Background ................................................................................................................................. 105
         metadata .................................................................................................................................... 106
   6.2. Usage .......................................................................................................................................... 107
   6.3. Output description ....................................................................................................................... 107
   6.4. Fragments from an XML event file ............................................................................................ 111
7. Slow control server ......................................................................................................................... 114
   7.1. Application protocol .................................................................................................................... 114
         Set request ................................................................................................................................ 114
         Get request ................................................................................................................................. 115
         Update request ........................................................................................................................... 115
   7.2. Controls Configuration file ......................................................................................................... 116
         Module ......................................................................................................................................... 116
         ggd - Jtec/Wiener Gate and delay generator ............................................................................. 118
         caenv812/caenv895 .................................................................................................................... 119
         vmusb .......................................................................................................................................... 121
         v6533 .......................................................................................................................................... 122
   7.3. Slow Control clients .................................................................................................................... 125
         7.3.1. Jtec MDGG-8 control panel ............................................................................................. 125
         7.3.2. CAEN discriminator control panel .................................................................................. 125
         7.3.3. VMUSB remote access support classes .......................................................................... 125
         7.3.4. The CAEN V6533 high voltage control panel ............................................................... 127
A. Reference material for the NSCL Scaler Display Program ............................................................... 129
   sclclient .......................................................................................................................................... 129
   ScalerDisplay .................................................................................................................................. 132
B. Validity checks on configuration parameter values ......................................................................... 141
C. Using the TreeParameterPackage to add Event Processors to SpecTcl ........................................ 144
   C.1. SpecTcl’s event processing pipeline ......................................................................................... 144
         C.1.1. The Event Processing Pipeline ......................................................................................... 144
         C.1.2. Writing event processors ................................................................................................. 154
         C.1.3. Locating dependent parameters ........................................................................................ 157
         C.1.4. Computing the parameter ................................................................................................. 159
   C.2. TreeParameter ............................................................................................................................ 160
         C.2.1. TreeParameter objects ..................................................................................................... 161
         C.2.2. TreeVariable objects ......................................................................................................... 164
   C.3. Source code for the examples ..................................................................................................... 167
   C.4. Non-Tree Parameter code .......................................................................................................... 167
   C.5. TreeParameter version ............................................................................................................... 182
   C.6. TreeVariable version ................................................................................................................. 197
D. IUCFMaseSpecTcl ............................................................................................................................. 213
List of Figures

2-1. Readout GUI main window ................................................................. 8
2-2. Folder GUI ....................................................................................... 12
2-3. Spectrum Context Menu ................................................................. 13
2-4. Folder GUI Parameter context menu ............................................... 13
2-5. Initial Spectrum Definition Dialog .................................................. 14
2-6. Spectrum editor for 1-d spectra ....................................................... 15
2-7. Spectrum definition dialog after expanding the parameter tree .......... 17
2-8. The folder GUI with a 1-d spectrum expanded .................................. 18
2-9. The generic gate definition dialog .................................................... 19
2-10. The slice definition dialog (initial) ................................................... 19
2-11. Or Gate Definition Dialog ............................................................. 21
2-12. Or gate dialog after the gate tree has been expanded. ....................... 22
2-13. The folder GUI after expanding the Gates folder ............................. 23
2-14. Gate application dialog ................................................................. 23
2-15. Filled in Gate Application dialog ................................................... 24
2-16. Save definition dialog ................................................................. 26
2-17. Attach online dialog ................................................................. 29
2-18. File data source dialog ................................................................. 30
2-19. Pipe data source dialog ................................................................. 31
2-20. Cluster File data source dialog ....................................................... 32
2-21. Folder GUI after creating the test spectra ....................................... 33
2-22. The Xamine window ................................................................. 34
2-23. The Geometry dialog box ............................................................ 36
2-24. Xamine’s spectrum chooser dialog box ......................................... 38
2-25. Xamine with the upper left pane loaded ....................................... 38
2-26. Xamine fully loaded with spectra ............................................... 40
2-27. Expansion of RAW.04 ................................................................. 42
2-28. The gate point acceptance dialog ................................................ 43
2-29. demo.slice in the folder GUI ....................................................... 44
2-30. Applying a gate when creating a spectrum ................................... 45
2-31. Effect of applying a gate to a spectrum .......................................... 45
7-1. CAEN V6533 control panel .................................................................. 127
Chapter 1. Introduction

This chapter is an orientation to the VM-USB data acquisition system.

- The section "Installation" describes how to install the Readout software on a system that is running the NSCL DAQ software. The NSCLDAQ software is a prerequisite to the successful installation of the VM-USB data acquisition system.
- The Section "VM-USB description" provides an overview of the VM-USB controller module
- "Supported Devices" provides information about which data taking devices are supported by this release and any support restrictions.
- "Script organization" is an overview to how configuration scripts cooperate to supply configuration for readout, SpecTcl and Scaler display programs from a common base configuration script.

1.1. Installing the software

The software can be downloaded from the NSCL DAQ project in sourceforge.net. The project is at: http://www.sourceforge.net/projects/nscldaq (http://www.sourceforge.net/projects/nscldaq). In the downloads tab, click on browse all files. In the package list, click VM-USB. Download the most recent llnlReadout-x.y-nnn.tar.gz tarball.

For the sake of specificity, the remainder of this section assumes you have downloaded llnlReadout-2.1.tar.gz to some working directory, you should use the most recent revision available at sourceforge. Unwrap the tarball:

```
tar xzf llnlReadout-3.0.tar.gz
```

This will create a subdirectory llnlReadout-2.1.tar.gz which contains the source code tree. The normal build procedure is:

```
cd llnlReadout-3.0.tar.gz
./configure --prefix=install-dir --with-spectrodaq-home=spdaq-home
make install
```

In the commands above, install-dir is a writable directory into which you will be installing the software. For shared installations, I recommend /usr/opt/vmusbReadout or some similar name. The Readout executable will be installed in the bin subdirectory of install-dir.

spdaq-home is the top-level directory of the NSCL spectrodaq installation. Typically this will be something like /usr/opt/spectrodaq.
Chapter 1. Introduction

The `configure` script has many more options and features.

```
./configure --help
```

describes them in detail.

Many versions of linux use the `udev` device permission management system. These linuxes require an edit to the configuration files for `udev` in order to ensure that non-root users can use the VM-USB.

`udev` configuration files are located in `/etc/udev`. The specific file to edit will depend on the linux distribution and linux distribution version. Locate files with lines like:

```
SUBSYSTEM=="usb_device", MODE="0664"
```

Change these to read:

```
SUBSYSTEM=="usb_device", MODE="0666"
```

### 1.2. VM-USB description

The VM-USB is a VME interface that connects to a host system via a USB-2 interface. USB-2 is capable of transfer rates of 480Mb/sec, however the USB protocol has a very high transfer initiation latency. This latency means that high performance data taking with the VM-USB must be done in a way that does not require event by event interaction with the host computer.

The VM-USB implements local intelligence via a capability described in its manual as a `stack`. A stack is essentially a list of VME actions stored in the VM-USB and triggered by an external condition. Stacks run without host interaction, filling buffers of data which can then be block transferred to the host computer. Stacks are the VM-USB’s way to work around the high latencies of USB-2.

The VM-USB supports eight stacks. Stack 0 is triggered by a NIM pulse on IN1. Stack 1 is intended to be periodically triggered to read out scaler data. The other stacks can be triggered via arbitrary interrupt conditions on the VME backplane. The host computer can also pass an ‘immediate’ stack to the VM-USB which is executed on receipt.

Stacks are rather hard to construct by hand. The VM-USB data acquisition system automates stack construction using a three level scheme:
1. A class library provides software that constructs stack elements (called lines in the VM-USB manual) from meaningful member function calls. The library supports both individual VME actions as well as the creation of stacks which can be executed immediately or downloaded to the VM-USB.

2. A C++ driver class is provided for each electronics module supported by the VM-USB Readout program. This driver knows how to interact with the VM-USB to initialize a module according to an attached configuration module, as well as how to contribute the necessary stack lines to read the module out, when the Readout framework is building stacks for download.

3. Each module is also represented by an extension command to a captive Tcl interpreter. When the run starts, this interpreter is used to interpret a configuration script that describes the hardware, desired configuration and the stacks to be created by the Readout software.

This flexible, and extensible scheme allows you to take data from the VM-USB without needing to know very much about it or about its hardware.

1.3. Supported Devices

The following devices are supported:

1. CAEN 32 channel digitizers, including the V785, V775, V792, and V862 are supported. The QDC digitizers do not include support (at this time) for setting the compensation charge. 775 support does not include support for setting the full scale time range.

If desired or required, these can be easily added.

2. Chained block transfers from a set of related digitizers is supported.
3. CAEN V830 multi-event scaler modules.
4. CAEN V977 I/O register (Input only).
5. The SIS 3820 scaler.
6. The SIS3804 8 channel scaler.
7. The Mesytec MADC32 32 channel peak sensing ADC. The Mesytec ADC is supported as a digitizer as well as a pseudo scaler from which dead time information can be retrieved.
8. The CAEN V1190 and CAEN V1290 multihit TDC collectively referred to as the CAEN V1x90.
1.4. Script organization

Configuration files that drive the VM-USB data taking application are stored in the ~/config directory of the account that is taking data. This allows applications to locate them, and scripts that depend on each other to locate each other.

~/config/daqconfig.tcl is the VM-USB Readout configuration file. It is directly interpreted by the Readout software whenever a run starts. You can therefore tweak the configuration of the experiment, and this change in configuration will be applied the next time the run starts. See the Configuration Files for a complete description of the configuration file format.

Several other applications interpret this configuration file via scripts that pick out the pieces of the configuration they need to operate. These applications may also require that metadata, in the form of specific Tcl variables be defined.

SpecTcl, for example, picks out the set of modules and the stack order, as well as parameter naming metadata, and uses it to configure the unpacking of VM-USB events to named parameters that can then be histogrammed. For some configuration, SpecTcl may also be programmed to create raw histograms for the parameters defined in the configuration file. See the SpecTcl chapter for more information.

The scaler display configuration file uses a script to pick out the set of modules that are loaded into the scaler stack. Metadata stored in specific Tcl variables defines the channels that can then be displayed on various tabs of the scaler display by the remainder of the configuration file. See the Scaler Display chapter for more information.

The system delivered to STUK includes a filter that translates raw event files to XML so that they can be loaded into a database. This program; evttoxml uses a metadata package to interpret the script so that it can write out a header that contains metadata describing the experimental setup. evttoxml also interprets the SpecTcl metadata to drive its own event unpacking software so that it can produce the XML version of each event. Finally the experimenter can supply an ancillary metadata file that provides arbitrary extensions to the metadata segment of the XML file via the metadata configuration command. See the Evttoxml chapter for more information.

Finally the Getting Started chapter describes how to get up and running quickly with the system as well as how to operate the system during a run. Additional documentation that describes the NSCL DAQ system is located at: http://docs.nscl.msu.edu/daq. SpecTcl documentation is at http://docs.nscl.msu.edu/daq/spectcl/.
Chapter 2. Getting Started

For people who don’t want to wade through a bunch of documentation, this chapter is a how to to get you up and taking data in a hurry. Where relevent, references to more detailed descriptions of the software are provided.

Much of this chapter is specific to systems that I have installed. If you have installed this software on your own, there may be differences between the procedures outlined here and what you will actually have to do.

When you’ve finished reading this material, you should understand

• How to prepare a data acquisition user account.
• How to start the components of the data acquisition system.
• How to operate the NSCL DAQ components.
• How to save data to DVD

2.1. Preparing a data acquisition account

The systems come with two pre-created accounts. The first, is the root account, which should only be used for system management and maintenance. The second, the daquser account is an ordinary account that should not be used for data acquisition, but can be used to login to the console and, via root consoles, or su, create data acquisition accounts. The initial root password is nscldaq. The initial daquser account is daquser. You should change both of these passwords at your earliest convenience.

Let’s go through the steps needed to create a data acquisition account. Fist login to the system and gain root access. You can either do this by logging in via ssh directly to root, or logging into the nscldaq account and using su to gain access to the root account. If you log in to the console, you can also use the Accessories → Root Console menu entry to gain root access. In either case, you’ll need to provide the root password.

Once you have gained root access enter the command:

```
adduser username
```

where username is the name of the new user. The adduser command has been extended so that:
Chapter 2. Getting Started

• An event area is created on the data disk mounted on `/events`, and a `~/stagearea` link created to point to it. The NSCLDAQ system uses the `stagearea` link to locate the event area when recording data.

• Sample configuration files are copied to `~/config`. For more information about configuration files see the chapter Configuration Files.

• Scripts are copied into `~/bin` so that you can easily start components of the data acquisition system without needing to know parameters each component needs.

• A `~/spectcl` directory is created and a `SpecTclRC.tcl` file planted there that will correctly initialize SpecTcl to analyze data from the VM-USB.

• Various configuration files are placed that inform the data acquisition system and SpecTcl that the default buffersize is 26656 bytes. This buffer size maximizes the data transfer performance of the VM-USB.

• SSH keys are created to allow you to ssh to `localhost` without providing a password. The NSCL data acquisition system is a distributed system. The graphical user interface that wraps the Readout program assumes that the Readout software might need to run anywhere on the network. It therefore uses SSH to start the Readout component even if it runs on the local system.

Once the account is created, login to it. The first time ssh is used with a new system (even the local one), you will be prompted to accept a host key that identifies that system. When the Readout program is run, it won’t be able to respond to that prompt therefore you should

```
ssh localhost
```

and accept the host key manually. Logout of the ssh session you started. You are now ready to start the data acquisition system components.

2.2. Data acquisition system components and how to start them

In addition to a persistent server called spectrodaq responsible for distributing data from the Readout program to client application, the data acquisition system consist of the following user started components.

ReadoutGUI

ReadoutGUI is a graphical user interface that controls the VM-USB readout program. It is responsible for starting the Readout in the system that is connected physically to the VM-USB (for STUK this is `localhost`). The ReadoutGUI also provides controls for starting and stopping runs as well as for controlling whether or not data from a run is recorded to disk.
Chapter 2. Getting Started

The script `~/bin/goreadout` is installed when the account is created. It starts the ReadoutGui telling it to start the VM-USB Readout software in the localhost.

SpecTcl

SpecTcl is the NSCL DAQ histogramming program. SpecTcl includes a visualization component called Xamine. Information about how SpecTcl is configured is provided in SpecTcl, as well as in the chapter Configuration Files.

The script `~/bin/gospec` is installed by the account creation scripts. It starts SpecTcl in a way that the configuration files are interpreted to define the initial set of parameters as well as some raw spectra.

SpecTcl’s GUI and commands can be used to define additional histograms and parameters. The GUI can be used to save histogram, gate and, and gate application definitions. On subsequent runs, this information can be reloaded.

ScalerDisplay

ScalerDisplay is a program that monitors the rates and totals of the run time scalers. Scaler Display describes how to configure the display pages and chart recorder provided by this application.

For the STUK installation of the system, the deadtime counters in the Mesytec ADC’s are used as pseudo scalers. They will be periodically read and the scaler display will provide dead-time displays online.

For more information about the scaler display and how to configure it, see the Scaler Display chapter.

The script `~/bin/goscaler` was installed when the account was created. It will start the scaler display program so that the scaler channels are for you by interpreting the `~/config/daqconfig.tcl` file. You may still need to configure the scaler display itself. See the Scaler Display chapter and Scaler display reference material for more information about that.

In addition to the individual startup scripts described above, the script `~/bin/startdaq` is installed when the account is created. This script starts all of the components described above.
2.3. Operating the NSCL DAQ components

This section describes at a high level, how to operate the NSCL DAQ components. The following subsections provide information about each of the components:

ReadoutGUI
   Describes how to operate the Readout GUI.

SpecTcl
   Describes how to operate the SpecTcl program.

ScalerDisplay
   Describes how to operate the scaler display program.

2.3.1. ReadoutGUI

This section describes how to use the ReadoutGUI. Note that when the ReadoutGUI starts, it will run the VM-USB Readout program in the target (localhost) system.

The figure below shows the Readout GUI main window:
The top part of the GUI contains menu bar with File and Scalers drop down menus.

The file menu has the following commands:

**File New...**

Brings up a dialog that allows you to select a new readout host and path to the readout program. Note that this path will get saved so that it is remembered for subsequent starts of the gui.
Chapter 2. Getting Started

File —→ Restart...

Restarts the readout program. This is only enabled if the readout program is actually running in the target system.

File —→ Start

Starts the readout program if it is not yet running or if it has failed and exited.

File —→ Source...

Prompts for the name of a file of commands to send to the readout program. Since the production readout software runs a Tcl interpreter it may be a perfectly reasonable thing to do to want to send a script to the Readout program. This entry does that.

File —→ Exit

Prompts for confirmation and, if it receives it, exits. Any readout program that is running is forced to exit after first being asked nicely to exit.

The Scalers menu is not used by the VM-USB Readout program. Scaler counts and readout period are set in the configuration file.

Immediately below the menu bar is a pair of boxes that display the name of the host in which the readout program will be run as well as the full path to the readout program. This ensures that for any readout program you will know what readout software is being run and where it is being run.

Below the Host/Readout Program documentation section is a set of controls that set the state variables for runs. This section of controls is disabled in the event a run is active, as run parameters can only be set if the run is halted.

The control labeled Title is a long entry widget. The entry widget in that frame displays the current title of the run. Simply edit the contents of that box to change the title.

Below the title control at the left side of the GUI is the Run Controls panel. This pane has two push-buttons and a checkbox. The Begin button starts the run and enables the Pause button. When the run is active, the Begin button changes to a End button and ends the run. If the run is paused, the Pause button similarly changes to a Resume button. Checking the Record button prior to starting the run enables event recording by running the EventLog application prior to starting the run.

To the right of the Run Controls panel is the Run Number control. When the run is not active, you can set the run number for the next run in that entry. The entry ensures that it always contains an integer number. This also implies that you can never clear the run number entry. To change a one-digit run number, type in the new run number and delete the old one.
Below the run number control, the **Elapsed Active Time** box displays the elapsed run time. This time should be treated as an approximation as it is just a clock that starts when the **Begin** or **Resume** buttons are clicked and stops when the **Pause** or **Resume** buttons are clicked.

Below the Run Controls panel, is a set of controls that allow you to do timed runs. Timed runs are performed by enabling the **Timed Run** checkbox and setting the desired length of the run in the boxes day, hour, minute and seconds boxes to the right of the **Timed Run** button prior to starting a run. When the elapsed run time is greater than the desired length of the run, the run automatically halts.

The large frame labelled **Readout Output** captures output from the Readout program. All output is captured uninterpreted. The Gui will also insert time stamped messages in that output indicating e.g. when runs are started or stopped.

The blank area below the **Readout Output** is a status display when the run is active. It displays the number of event file segments (event files are not allowed to be more than 2Gbytes long), and the total amount of data recorded.

The status display also uses a simple bar chart to show the fraction of the buffer pool available for data. If this gets low, the green bar will turn yellow and, eventually, red. To get a detailed picture of which nodes are clients and the buffers owned by each client, you can click the **Details...** button to bring up an additional window that describes that information.

### 2.3.2. SpecTcl

SpecTcl displays several windows. The three you will interact with the most in order of decreasing frequency are probably:

1. **Xamine**, which display spectra, allows you to graphically enter primitive gates, and lets you get simple statistics from areas of interest

2. The folder **GUI** which allows you to create spectra, and compound gates, as well as to apply those gates to spectra. Since the folder GUI supports saving and restoring its definitions, usually you will use this at the start of a set of related runs, and then read in the definition whenever you need to restart SpecTcl.

3. The **TkCon** window, which provides access to SpecTcl’s command; an extended Tcl/Tk interpreter.

Since you need to understand the Folder GUI in order to set up you analysis, we will start there. Once you’ve set up the initial analysis, you’ll want to look at spectra and perhaps create gates. This is done in the Xamine window, so we will describe that next.

The TkCon window will not be described, as you can think of it as just a terminal window that is connected directly to SpecTcl. Type commands at it and SpecTcl will execute those commands. Tkcon
supports limited command line editing and recal.

2.3.2.1. Introduction to the SpecTcl folder GUI.

The Folder GUI is a direct manipulation GUI that represents parameters, selected Tcl variables, gates, and spectra as a folder hierarchy. The assumption is that the names you choose will represent a position in the hierarchy where levels of the hierarchy are separated by periods. For example, the name \texttt{a.b.c} represents the item \texttt{c} in a folder named \texttt{b} that is in turn inside a folder named \texttt{a}.

When you start SpecTcl using the folder GUI, the folder GUI window will look like this:

\textbf{Figure 2-2. Folder GUI}

The folders that have a + to their left can be expanded to show more detail. Items may also be expandable to show a full graphical description of the item.

In this section we’re only going to look at a subset of the capabilities of the folder GUI. I urge you to explore the menu options and the online help the GUI offers. The online Help is available through the menu: \texttt{Help $\rightarrow$ Topics}. This section is sub-divided as follows:

1. Context Menus describes what context menus are and how to bring them up.
2. Creating Spectra describes how to create the various spectrum types using the folder GUI.
3. Creating Gates describes how to create gates using the folder GUI.

4. Saving and Restoring Definitions describes how to use the menu GUI to save and restore the definitions you have created both with the GUI and in any other way.

5. Selecting Data Sources describes how to select a data source for SpecTcl using the folder GUI. SpecTcl can analyze data from the online system, in which case it samples as much data as it is able to keep up with without slowing down the online system. SpecTcl can also analyze data from saved event files, and from the output of programs via a pipe data source.

2.3.2.1.1. Context Menus

The folder GUI uses context menus to allow you to do the most common actions on an item in the folder hierarchy. A context menu is a menu that pops up in response to a click of right-most button on the mouse. Use a context menu by holding down mouse-button 3, moving the pointer over the desired selection and then releasing the button. If you decide not to select any of the menu choices, just drive the pointer off the menu before releasing the button.

The selections available in a context menu depend on where the mouse cursor is when mouse button 3 is clicked. The context menu you will get when you are over the spectrum folder looks like this:

![Figure 2-3. Spectrum Context Menu](image)

In this case, the New... menu selection will create a new spectrum, the Clear all selection will clear the counts in all the spectra. For each context menu, the Help selection will pop up help that describes the relevant part of the folder GUI and the context menu you have popped up.

By contrast, the context menu that will pop up when the pointer is over the Parameters folder looks like this:
Figure 2-4. Folder GUI Parameter context menu

To learn more about the context menus, and what they can do, experiment by seeing what you get when you right click over various things. There are context menus for items as well as folders. Look at the online help for each menu if the function of its items is unclear.

If you learn better by reading, you can go to the online help by clicking Help→Topics... In the help browser that pops up select the Topics→Browser item. At the bottom of that page are links to the help for each of the context menus.

2.3.2.1.2. Creating Spectra

The folder GUI allows you to create and edit spectrum definitions. Be aware, however that when you edit an existing spectrum definition, what the GUI will actually do is delete the old spectrum and create a new spectrum with the same name as the old spectrum and the new definitions. You will lose all the counts that had been accumulated in the old spectrum.

There are two ways to create a spectrum with the Folder GUI.

1. Select Spectra→Create... using the menu bar at the top of the gui window.
2. Use the context menu New... on the context menu for the Spectra folder.

Regardless of how you get there, you will be presented with an initial spectrum definition dialog:
Figure 2-5. Initial Spectrum Definition Dialog

Let’s look at the procedure to make a spectrum named test.1d (This should create a folder test under the Spectra folder with the item 1d). Note that the images screenshots you see will be from an unconfigured SpecTcl. If you want to follow along you will need to create an empty directory, in that directory, type the commands:

Example 2-1. Creating an unconfigured SpecTcl

```
cp /usr/opt/spectcl/current/Skel/* .
make
./SpecTcl
```

Do not attempt to use this unconfigured SpecTcl to analyze online or event data taken with the STUK system. It does not have the correct event processors and startup scripts.

Enter the name of the new spectrum test.1d in the Spectrum Name: text box.

Different spectrum types require different sorts of information. Initially, the dialog does not know what type of spectrum you are creating. The Spectrum Type button displays a drop down menu. Select 1-d from that menu to tell the GUI you are creating a 1-d spectrum.

The GUI expands to the spectrum specific editor for 1-d spectra:
Chapter 2. Getting Started

Figure 2-6. Spectrum editor for 1-d spectra

On the left 1/2 of the dialog is subset of the folder hierarchy. A 1-d spectrum requires a parameter and may optionally have a gate applied to it. The parameter and optional gate are chosen by double clicking them on the subset browser.

Once a parameter has been selected, the right side of the dialog displays it and allows you to set the spectrum axis limits and bin count. If the parameter has been defined as a Tree parameter it may have range and resolution information. If so, the dialog will use that to suggest values for the axis limits and bin count. If the parameter has a units string associated with it, the units will be displayed as well.

We don’t have any gates defined so we are only going to select a parameter and set the range and binning of the spectrum X-axis. Click the + to the left of the parameters folder (left mouse button). The Parameter folder expands to reveal an event folder. Click its + to expand it. Click the + of the raw folder this revealed.
Chapter 2. Getting Started

The spectrum definition dialog should now look like this:

Figure 2-7. Spectrum definition dialog after expanding the parameter tree

![Figure 2-7. Spectrum definition dialog after expanding the parameter tree]

Each of the green square icons that has a script  \( T \) with a lowercase  \( p \) kerned inside it represents a parameter. The last part of the parameter name is displayed to the right of its icon, and the column of the tables to the right of the folder tree give the properties of each parameter.

Recall that periods separate parts of the hierarchy. The Parameters root folder is not part of a parameter name, but simply indicates its subfolders contain parameters. Therefore, the icon labelled 00 really represents the parameter named `event.raw.00` (Parameter 00 living in folder `raw` that itself lives in folder `event`).

Double click on the `event.raw.00` icon. Note the changes in the right 1/2 of the spectrum definition dialog.

Note how the parameter you select has suggested an axis definition. Let’s change that to an axis that runs
from 0 to 1023 with 1024 bins. Enter 0 in the Low text entry, 1023 in the High box and 1024 in the Bins box.

Finally, create the spectrum and dismiss the dialog by clicking its Ok button. (Accept creates the spectrum but leaves the dialog displayed so you can create more spectra).

Note that there is now a + next to the Spectra folder in the main Gui. Click that +. Since we named the spectrum test.1d, we get a folder named test that also has a +. Click that + to see the spectrum (an icon that looks like a spectrum with a few peaks). Note that:

• The spectrum type, and axis definition appear in the columns to the right of the spectrum icon.
• The spectrum icon itself has a + to its left indicating it can be expanded to display more detail. If you click that + The parameter histogrammed will be displayed.

If you click the + to the left of the histogram definition, the folder GUI will look like this:

Figure 2-8. The folder GUI with a 1-d spectrum expanded.

Expanding different spectrum types will provide different information about the spectrum details. For
example, a 2-d spectrum will an item for each axis describing the parameter on that axis, and the limits and binning of that axis.

I conclude this section by encouraging you to play around creating different spectrum types, exploring the types of spectrum definition dialogs you’ll see. You may also want to click on each spectrum definition dialog’s Help button to explore the on-line help for each of these spectrum editors.

2.3.2.1.3. Creating Gates

If you have read the section on creating spectra, you already have most of the skills needed to create a gate with the folder GUI. Please note that primitive gates like slices, contours of bands can be most easily entered by clicking on spectra displayed by Xamine. Compound gates are most easily entered using the folder GUI.

In spite of this, in this section we will:

1. Create a pair of slice gates, on event.raw.00 and event.raw.01 (recall that gates are defined on parameters even though they can be created on spectra that are created on those parameters).
2. Create an OR gate that will be true whenever an event is in at least one of those gates.
3. Apply the OR gate we made in the previous step to the spectrum we created in the previous section (test.1d).

Ok, let’s start by making the two slice gates. Gates have a gate definition dialog that, like the spectrum definition dialog consists of a generic and gate specific section. Bring up the gate definition dialog either by selecting Gate → Create... or by selecting New... from the Gates folder context menu.

This will bring up the generic part of the gate definition dialog:

Figure 2-9. The generic gate definition dialog

Fill in the gate name as slices.01 and select the gate time to be a slice. This will expand the gate definition dialog to include the slice editor. Note the reminder at the top of this dialog that you can enter slice gates graphically as well.
Figure 2-10. The slice definition dialog (initial)

The top part of the new area is a generic parameter chooser that is used in a lot of gate definition dialogs. The top left part of the newly displayed part of the dialog is once more a subset browser. In this case the subset browser allows you to select a parameter. The right hand top part of the gate definition dialog is a list of the parameters on which the gate will be defined. In this case the dialog will only allow one parameter to be chosen.

Using the + open the Parameters folder and underneath it, successively, the event and raw folders. Chose the parameter icon that corresponds to event.raw.00. Note that

1. That parameter is removed from the list of available parameters preventing you from specifying it twice in gates that use more than one parameter.
2. The full parameter name is now in the Parameter(s) selected listbox.

See what happens if you now select a different parameter (e.g. event.sum). Finally, select event.raw.01.

Enter the slice limits at the bottom of the dialog in the text boxes labelled Low Limit: and High Limit: respectively.

Since we are going to create another slice, click the Accept button to accept the gate.
The default new gate type is another slice, so now let’s enter the gate name, and select parameter. In the Gate Name: text box enter slices.02. Select the parameter event.raw.02, and enter whatever limits you feel like entering, then click Accept again.

Next we’ll make a gate named OrGate which will be true when at least one of slices.01 or slices.02 are true. Enter OrgGate in the Gate Name text box. Click on the Slice button to drop down the gate type menu and select Or (+). Your dialog should now look like:

Figure 2-11. Or Gate Definition Dialog
Or gates depend on other gates, rather than parameters. Therefore, the gate definition dialog consists of a browser for the existing gates, and a listbox showing the gates that have been accepted.

Open the Gates folder and then the slices folder underneath it. You should see something like this:

**Figure 2-12. Or gate dialog after the gate tree has been expanded.**

![Or gate dialog after the gate tree has been expanded.](image)

Double click on the two gates (`slices.01` and `slices.02`) in any order to add them to the `Dependent Gates` listbox. Click the `Ok` button to accept this last gate and dismiss the dialog.
Fully expand the Gates folder, all subfolders and items in all the subfolders. This should give you something like this:

Figure 2-13. The folder GUI after expanding the Gates folder

Expanding a gate provides you with the gate dependent information about how that gate has been made.

As we know, gates are only useful when they are applied to a spectrum. Let’s apply OrGate to The test.1dspectrum. There are actually three ways to get to the Gate application dialog.

1. Select Gate→Apply... which brings up the gate application dialog completely uninitialized.
2. Select the Gate... entry from a Spectrum context menu. This brings up the same dialog with the spectrum already selected.
3. Select the Apply To... entry from a Gate context menu. This brings up the same dialog with the gate already selected.

In addition, we have already seen that the dialog for creating a spectrum allows you to specify an initial gate.

So that we can go through the entire process we’ll select Gate→Apply... from the menu bar. This brings up the uninitialized gate application dialog:
Application is the process of selecting a single gate that will be applied to at least one spectrum. Open the Spectra folder and its test subfolder. Double click on the 1d spectrum to set test.1d as the only spectrum that will be applied to. If you have more than one spectrum you want to apply the same gate to, you can keep double clicking spectra until all the target spectra are listed in the Spectra: listbox.

Select that gate to apply by opening the Gates folder and double clicking on the Orgate gate icon. Your Gate application dialog should now look like:
Figure 2-15. Filled in Gate Application dialog.

Apply the gate by clicking the Ok button.

I will conclude this section by once more encouraging you to play with the gate definition editors. See what each gate asks you to provide, and what details each gate type provides in the folder Gui. Once more all the gate definition dialogs have a Help button that takes you to a help page appropriate to that type of gate editor.
2.3.2.1.4. Saving and Restoring Definitions

The set of parameter, spectrum, and gate definitions and gate applications that make up a realistic analysis can be quite large. Re-creating them each time is error-prone, and just plain no fun. SpecTcl’s commands offer enough introspection capability to allow the folder GUI to write scripts that can be read in to restore these definitions at a later time.

Recognizing that you may not remember to save your definitions, or that there may be failures in the hardware or software that leave you with unsaved definitions, the folder GUI automatically writes a definition file named `failsafe.tcl` whenever you change the definitions.

You may also explicitly save and restore definitions from a named file. To save your definitions to file, click the File —> Save... selection from the menu bar. This brings up the following dialog:
The set of checkboxes at the top of the widget describe which definitions will be saved to file. By default all definitions are saved. The remainder of the dialog box allows you to choose a file to which the definitions will be saved. Once you have done that, click Ok to save the file.

For the sample spectra we have been creating, this setup file will look like:
Example 2-2. Sample folder gui saved settings file

```
# SpecTclGUI save file created Fri Jan 04 04:31:12 PM EST 2008
# SpecTclGui Version: 1.0
#        Author: Ron Fox (fox@nscl.msu.edu)

#New Tree Parameters:

#Modified Tree Parameters:

# Pseudo parameter definitions

# Tree variable definitions:

# Spectrum Definitions

catch {spectrum -delete test.1d}
spectrum test.1d 1 event.raw.00 {{0.000000 1023.000000 1024}}

# Gate definitions in reverse dependency order

gate slices.01 s {event.raw.01 {100.000000 200.000000}}
gate slices.02 s {event.raw.02 {0.000000 150.000000}}
gate OrGate + {slices.01 slices.02}

# Gate Applications:

apply OrGate test.1d

# filter definitions: ALL FILTERS ARE DISABLED!!!!!!!
```

A few observations:

- The configuration file is just a Tcl script you could just source it in and refresh the tree parameters.
- Spectra are deleted before being created, the *catch* command ensures that if the spectrum does not exist the script will continue to execute.
- Gates are written out in reverse dependency order. That is A gate will not be written to the configuration file until all gates it depends on have first been written. This ensures that loading the file will always work (consider what would happen if OrGate had been written prior to slices.02).

While definition files can just be sourced, the folder GUI also provides a File —→ Load... Menubar menu item. This just brings up a file chooser dialog. When you select the desired file, SpecTcl will source it in, refresh the object browser and sbind all spectra.
2.3.2.1.5. Selecting Data Sources

SpecTcl analyzes data from a data source. The current version of SpecTcl understands two types of data sources:

- File; File data sources in turn can be either raw event files or filtered data files.
- Pipe; Pipe data sources are programs that write data to their standard output which is connected by means of a pipe to SpecTcl.

There are also two special cases of piped data sources that are important:

- Connecting to the online system (via spectcldaq, a pipe data source in the NSCL data acquisition system.
- Connecting to a list of run files (called cluster files by some.

The folder gui Data Source menu has menu entries that allow you to connect to any of these types of data sources. In this section, we will examine each of the option on the Data Source menu and the dialogs they bring up.

**Attaching to the online system.** The Data Source → Online (spectrodaq)... menu entry attaches SpecTcl to a pipe data source that connects to the online system. For this to work, the NSCL data acquisition software must be installed on the local system. SpecTcl will look for the pipe adapter in various directories rooted in `/usr/opt/daq`. If your installation of the NSCL DAQ system is elsewhere, you can use the Edit → Preferences... menu entry to tell SpecTcl where the NSCL DAQ system is located.

Clicking on **Data Source → Online (spectrodaq)...** displays the following dialog:

**Figure 2-17. Attach online dialog**

![Host Prompt Dialog](hostprompt.png)

The Host entry box is where you will enter the name of the computer that is attached to the experiment (from which you are accepting) data. If the default buffer size is not correct you can adjust it with either the up and down arrows or by typing the correct size in the Buffer size in bytes: spinbox.
Click the Ok when you are ready to take data. The Help provides more information about the dialog. The default buffer size can also be set in the Edit → Preferences... dialog.

**File data source dialog.** The Data Source → File... dialog prompts for a file from which to read event data. In the NSCL data acquisition system, the buffersize used to acquire data is encoded in the filename. E.g. run123-4096.evt contains data from run number 123 taken with a buffersize of 4096 words (8192 bytes). Note that SpecTcl’s buffersizes are always specified in bytes.

In the dialog below:

**Figure 2-18. File data source dialog**

![File data source dialog](image)

use the file chooser section of the dialog to select the event file to read. If the buffersize of that event size is not the same as the buffersize in the Buffer Size: spinbox, select the correct buffer size using the spin box arrows or by typing a value in to the entry. When the dialog is filled in correctly, click the Ok to start analyzing data from the selected file.

**Attach Pipe Dialog.** The Data Source → Pipe... menu selection allows you to attach SpecTcl data from the output of any program. One possible use would be to attach to gzcat to analyze compressed
event data without the need to create a decompressed data file.

Attaching to the online system, and attaching to a list of runs (cluster file) is a special case of attaching to a pipe data source. When you attach to a pipe data source, you get the following dialog:

**Figure 2-19. Pipe data source dialog**

Use the file browser to select the command or type in the command in the **Selection** text box. Enter any command line arguments or switches the command should receive in the **Parameters** text box. If necessary use the **Buffer Size** spinbox to set the correct buffersize.

When you have filled in the dialog correctly, click the **Ok** to start analyzing data from the program.

**Reading Cluster Files.** A special case of reading data from a file is the analysis of **cluster files**. A cluster file is a file that contains the names of event files. When you start analyzing a cluster file, the files named in the cluster file are analyzed sequentially.
The Data Source—List of runs... selection brings up the dialog shown below:

Figure 2-20. Cluster File data source dialog

Simply select the cluster file (the file with the list of event file names), and click Ok to start analyzing. The file will be analyzed with the default buffer size which can be set using the Edit—Preferences... menu selection.

Filter Files. A filter file is an event file with a subset of parameters from a subset of events. Filters are created using the Filters menu. Filter files are written in a special file format. You must have prepared your SpecTcl to read a filter file before you can process it.

To open a filter file data source with the folder Gui, you should click on Data Source—Filter File.... You will first see a dialog that reminds you that you must have built SpecTcl properly to accept filter files. Click on Ok to continue or on Cancel if your SpecTcl has not been built to handle filter files.

If you click on Ok You will get a file chooser from which you can select the filter file to use as the data source.
2.3.3. Xamine

Once you have created a set of spectra, you will naturally want to Xamine them. Xamine is a Motif program that communicates with SpecTcl via shared memory. SpecTcl’s `sbind` command loads spectra into shared memory. Xamine displays them from that shared memory.

In addition, Xamine allows you to graphically create and accept primitive gates, and much more.

In this section, we will confine ourselves to the following tasks:

1. Creating a layout or geometry, as Xamine calls it.
2. Populating the layout with spectra
3. Saving the layout (or window file) and later reloading it.
4. Graphically creating and accepting a gate.

Xamine has read-only access to spectra so you should not be able to damage anything. I would therefore like to encourage you to play with Xamine further to explore its features.

The starting point for all of the subsections in this section assumes that we are running the unconfigured SpecTcl, and have made raw spectra for each of the parameters. We will also have made a 2-d spectrum of the first pair of parameters.

If you are following around, create the spectra and populate them with some using SpecTcl’s test data source. In the TkCon window type `start` wait for a bit and then type `stop`. The test data source is used if analysis is started without selecting a data source. It provides fixed length events of several parameters that are sampled from different gaussian distributions. The parameters are in the range 0-1023.

The figure below shows what the folder GUI looks like when you have created these spectra. Try using the array checkbox to create all of the raw spectra in one operation.
Figure 2-21. Folder GUI after creating the test spectra.

Before we get on with the detailed task descriptions, let’s look at the Xamine window, and its components.
As with most of today’s graphical user interfaces, the top strip of the window consists of a menu bar. Pull down some of the menus to see what they contain. Note that the **Options** menu allows you to set the default display characteristics of spectra displayed by Xamine. The **Spectra** menu modifies the characteristics of the individual selected spectrum.

The large empty area below the menu bar is the spectrum display window. When we configure Xamine, this area will be where spectra will be displayed. The status bar below that provides information about the spectrum, including the location of the cursor in spectrum coordinates if the cursor is hovering or moving over the spectrum display window.

Below the status bar is a set of grouped buttons. Think of this area as containing several toolboxes containing buttons that get you directly at the most common Xamine operations. All of the buttons in this
area duplicate functionality available in the menus in a much more accessible way.

One more note before we continue. Most Xamine dialogs are not modal. This means that you can interact with other elements of the Xamine user interface while a dialog is displayed.

### 2.3.3.1. Creating a layout and populating it

The Spectrum Display window can be subdivided into a rectangular array of *panes*. You can then load each of these panes with a spectrum.

We’ve made 12 spectra. One for each of ten raw parameters, one for the sum parameter, and a 2-d spectrum. Therefore, let’s subdivide the spectrum display window into four rows and three columns.

Click on the **Geometry** button. This displays the geometry selection dialog. Select four rows and three columns from the two columns of radio buttons. When you are done, the dialog should look like this:
Chapter 2. Getting Started

Figure 2-23. The Geometry dialog box.

Clicking on the Ok button of the dialog subdivides the spectrum display window into the desired panes.

Note that the upper right pane looks different, like it’s pressed in. One pane will always look like this. This pane is called the selected pane, or selected spectrum. Any operation that can only affect a single pane, operates on that pane.

Click in a different pane. That pane becomes the selected pane. As a simple example of an operation on a selected pane, double click in the selected pane. Note that the zoom check button in the left most set of buttons in the toolbox is now selected. The double click zoomed the selected pane to fill the entire spectrum display window. Double click in the spectrum display window. That un-zooms (note that the zoom is no longer selected.

Loading a spectrum into a pane is an operation that affects a single pane. Let’s load the upper left pane. Click in the pane to select it. And click the Display button. This displays the spectrum selection dialog
Figure 2-24. Xamine’s spectrum chooser dialog box.

Double click on the RAW.00 spectrum.
We could continue in this manner, selecting panes, clicking the Display button selecting spectra and so on. That would be tiresome, however. Let’s look at a faster way to load the panes.

Click the next pane over to the right (second from the left top row). This time, to load the pane, click the Display+ (that’s the button below the Display button).

Double click the RAW.01 spectrum. Note that:
- The spectrum choice dialog remains displayed
- The selection has automatically moved to the right by one box.

This allows us to efficiently load the entire set of panes.
Do this by double clicking on the `RAW.02` through `RAW.09` spectra (note how the selection progresses through the rows), the `SUM` and the `XY` spectra. Only five parameters have a value for each event. Therefore the resulting display should be:

**Figure 2-26. Xamine fully loaded with spectra**

Dismiss the spectrum chooser by clicking the **Cancel** button.

### 2.3.3.2. Saving and restoring a window file.

While we have seen that it is very easy to create geometries, and to populate them with spectrum displays, in many cases an experiment will need several views of the data. Furthermore, it would be rather onerous to have to populate even one view everytime we started SpecTcl and Xamine.
Chapter 2. Getting Started

Fortunately, Xamine has the capability to save and read back *window files*. A window file is the current geometry, the set of spectra populating it, and the display properties of each pane. (If a spectrum has been expanded, the window file will store the expansion. If, when the window file is saved, one of the panes is zoomed, the window file, when read in will cause the layout to be zoomed to the same spectrum.

Let’s save the layout we made previously. Pull down the *Window* menu on the menu bar and click *Write Configuration...* (as you might guess you can also find this function in the *File* menu). In the file selection box that is displayed, the focus is in the filename text entry, and the insertion point is at the end of the directory path. Type `all.win` and click *Ok* to save the layout as `all.win` in the current working directory.

If you want to overwrite an existing file, simply locate it in the file/directory boxes that make up the majority of the file selection box, and double click it. Confirm when prompted if you are sure you want to overwrite.

Next; set the geometry to a single pane (one row, one column). Setting a new geometry always leaves you with empty panes.

To read in the window file, select: *Spectra* → *Read Configuration*... Double click the `all.win` file. Note that the layout has been restored.

### 2.3.3.3. Creating gates with Xamine, and applying them to spectra.

In this section we will create some gates on existing spectra, and create new spectra to which those gates are applied.

A gate is a condition that is checked for each event. When a gate is applied to a spectrum, that spectrum can only be incremented for events that make its gate true. Each spectrum can only have a single gate applied to it, however a gate can be applied to many spectra.

SpecTcl has a rich set of gate types that include primitive and compound gates. Primitive gates are defined on a parameter and include slices (low/high limits on a single parameter), bands (polylines drawn in a two parameter space which are true when the parameters form a pointe below the band), and contours (closed polygonal figures drawn in a two parameter space that are true when the parameters forma point within the polygon). Compound gates are gates that are defined as a combination of existing gates (primitive or other compound gates). These include the and gate (true if all of the dependent gate are true), the or gate (true if any of the dependent gates are true), and the not gate (true if its single dependent gate is false).

In the chapter that describes the folder GUI ([see the subsection on making gates](#)), we learned that it is possible to make gates using the folder GUI. Primitive gates, however are more easily defined by accepting points on a spectrum that displays the parameter(s) on which the gate is defined.
Let’s start by making a slice gate using Xamine. A slice is a pair of limits on a parameter. Slice gates define a half open interval closed at the lower end. We will set the slice gate on the parameter \texttt{event.raw.04}. It is important to keep in mind that while gates may be drawn on spectra, they are defined on parameters.

The \texttt{RAW.04} spectrum displays the spectrum of \texttt{event.raw.04}. Double click on it so that we can see it more clearly (you don’t need to zoom a spectrum to draw a gate in it, but it’s often easier to see where to put the limits if you do).

Since the peak is rather narrow and close to the origin, let’s expand the spectrum first. Click the \texttt{Expand} button in the middle toolbox. The expand dialog pops up to allow you to type in the expansion limits, instead, just click the mouse close to the Y axis and somewhere past the end of the peak. Dashed vertical lines show you where the expansion limits are. You can use the \texttt{Delete} buttons in the dialog or the right mouse button to remove the limits if you don’t like them. When satisfied, click the \texttt{Ok} button in the expansion dialog to dismiss the dialog. The pane now only shows the portion of the spectrum between the limits you selected. For example:
The `UnExpand` restores the display of the full spectrum. You may further expand an expanded spectrum.

Now let's create the slice gate. For historical reasons, Xamine calls slices `Cut` gates. Click the `Cut` button to start the procedure. The gate point acceptance dialog is displayed:
In the text entry labeled Object Name, type the name of the gate for example demo.slice. Note that when the gate is displayed in the folder GUI, periods will be interpreted as path separators in the folder hierarchy so the name demo.slice will create a folder, demo in the gates folder. The demo folder will have a gate named slice in it.

Use the mouse pointer to click in the limits of the slice on the spectrum. Note that when accepted, the actual slice points will be converted to values in raw parameter space if the spectrum is compressed relative to the parameter. Choose a relatively narrow gate that does not cover the entire peak.

Click the Ok button on the gate point acceptance dialog to finish defining the gate. Note that the spectrum displays the gate. Primitive gates are displayed on any spectrum that is defined on its parameter(s). In the folder GUI window, right click anywhere in the folder hierarchy and accept Refresh Tree.

If you expand the Gates and then the demo folder below it, and then the slice gate in the demo folder, you’ll see something like this:
Chapter 2. Getting Started

Figure 2-29. demo.slice in the folder GUI

In order to see very clearly what it means to apply a gate to a spectrum, use the folder GUI to create a spectrum named raw.04-gated, that displays event.raw.04. Before creating the spectrum, apply the gate to it by opening the Gates folder in the spectrum creation dialog drilling down to the demo folder and double clicking the slice gate. The gate applied to the spectrum should be displayed at the top of the spectrum creation dialog:

Figure 2-30. Applying a gate when creating a spectrum.

You can also apply an existing gate to an existing spectrum. In the folder GUI, if you right click a spectrum, and choose Gate... the resulting dialog will allow you to apply any gate to that spectrum. You can apply the same gate to several spectra in one operation by selecting the Gate→Apply... menu command.

In the TkCon window type start wait a bit and then stop to accumulate a bit more test data (this is not necessary... you could do all of this with data continuously flowing into SpecTcl as SpecTcl and Xamine’s user interfaces are fully live while analyzing data).

Set the geometry to be a 1x1 (single pane), and display the RAW.04-GATED spectrum in that pane. Note that it only has counts within the gate lines.
2.3.3.4. More playing around with Xamine

As an exercise, create a contour gate on the spectrum X#Y#, create a new spectrum on event.raw.00, apply the contour to that spectrum. Clear all the spectra (folder GUI Spectra → Clear All), and take more data. Display both of the spectra on event.raw.00, and use the Xamine Summing Region buttons to set summing regions on each of the spectra, then the Integrate button to see how many fewer counts the gated spectrum accumulated. If your contour was completely inside the peak in the X#Y# spectrum you should also see the tails of the gated spectrum cut off.

In Xamine, select the RAW_00 spectrum. Next, from the menu select: Spectra → Superimpose...\ In the resulting dialog, double click the gated spectrum. Does this help you see the differences between the two spectra clearly?
2.3.4. ScalerDisplay

The NSCLDAQ Scaler Display program provides the ability to display the rates and total counts of individual scalers, as well as the ratios of pairs of scalers. This is used in the STUK system to display the dead-time total time, and dead-time to total time ratio.

When a single page of scalers is displayed, the display program has no controls. It is simply a passive program.

2.4. Writing event data to DVD

Having taken data you will eventually want to write it to DVD. If you put a blank DVD into the drive, a DVD icon will appear on the desktop. Double clicking that icon brings up the GNOME DVD writing application. You can drag the event area to the DVD window and then burn the DVD. The event area is named /event/username where username is the name of the account that took the data.

If you have taken more event data than will fit on a disk, you may need to drag subsets of the event area for each dvd you burn. The best way to do that is to drag run directories from the experiment subdirectory of the event area to the DVD. All other directories in the event area only contain symbolic links.
Chapter 3. Configuration Files.

This chapter describes the configuration files that drive Readout, SpecTcl and the Scaler display. The end of this chapter contains reference material that describes the extensions to Tcl that have been incorporated into the Readout configuration script engine to support building VM-USB stacks directly from the configuration file `~/config/daqconfig.tcl`.

3.1. daqconfig.tcl The readout config file

The `~/config/daqconfig.tcl` file is the basis for configuring the Reaout (directly), SpecTcl the ScalerDisplay, and the event file to XML converter (indirectly). This file is a script for a Tcl interpreter that has been extended with additional commands that support describing the hardware to be readout and how to aggregate that hardware into stacks. It may be better to think of `daqconfig.tcl` as a configuration program, rather than a simple configuration file.

Extensions to Tcl for configuration are a group of commands. The command keyword for each command indicates the type of device or item it manipulates. For example the `adc` command creates, configures or gets the configuration of CAEN V785, V775, V792 or V862 digitizers.

The extensions to Tcl are what Tcl calls *command ensembles*. A command ensemble is a command with subcommands. Each extension has three subcommands:

create

Creates an item of the type associated with a command. Many of the device creation subcommands also support decoding device configuration information on the same line.

config

Configures an item of the type associated with a command. Configuration determines the way the module is initialized for data taking.

cget

Returns configuration information about an item of the type associated with a command. This allows intelligent configuration scripts to be written that can introspect the configuration of a device and take action depending on that configuration.

See the command reference at the end of this chapter for a list of the commands that have been added to Readout’s configuration Tcl interpreter to support configuring the readout of your experiment.
3.2. Configuring SpecTcl

Using the helper files `~/config/configFile.tcl`, and `~/config/spectclSetup.tcl`, SpecTcl's startup scripts are able to process enough of the `~/config/daqconfig.tcl` to be able to create a correspondence between module channel numbers and parameter names. The scripts create global variables that SpecTcl uses to determine the order and type of each element of each stack. You provide global array element definitions that associate SpecTcl parameter names with each module you are using.

From this information, SpecTcl is able to configure its event decoding modules so that the proper module decoders are invoked in the order required by the event structure and so that the data from each module is unpacked into the correct set of parameters for histogramming.

The key to this is the `adcChannels` array. Tcl arrays are indexed by strings. In this case the `adcChannels` array is indexed by the names of each module. The value of each of these elements is a list of the parameter names for each channel of that module.

Let's look at an example of a configuration file fragment that defines the parameters for the X strips of a double sided silicon strip detector (dsssd).

**Example 3-1. Sample configuration and `adcChannels` definitions**

```tcl
# Define and configure the hardware:

madc create dsssd1.x -base 0x40000000 -id 4 -ipl0
madc config dsssd1.x -gatemode common -gategenerator disabled
madc config dsssd1.x -inputrange 8v
madc config dsssd1.x -timestamp on -timingsource vme -timedivisor $madcTimeDivisor
madc config dsssd1.x -thresholds $thresholds

...  

# Tell spectcl how to decode the modules:

set adcChannels(dsssd1.x) [list x.00 x.01 x.02 x.03 x.04 x.05 x.06 x.07 \  x.08 x.09 x.10 x.11 x.12 x.13 x.14 x.15 \  x.16 x.17 x.18 x.19 x.20 x.21 x.22 x.23 \  x.24 x.25 x.26 x.27 x.28 x.29 x.30 x.31]

...  
```

The first part of the configuration file creates a module named `dsssd1.x` which is a Mesytec MADC 32. This module is configured appropriately in the lines prior to the ellipses.

The `set adcChannels(dsssd1.x) ...` command indicates to SpecTcl that the module named `dsssd1.x` will have as inputs, parameters named `x.00` through `x.31`, which are the individual strip energies.
The SpecTcl initialization file setup.tcl processes this configuration to produce SpecTcl parameters and 1-d spectra for each of these parameters.

3.2.1. A note about the Hytec 2530 ADC.

The Hytec 2530 ADC is an 8 channel ADC. Each event from the ADC includes a timestamp. Timestamps are 48 bits wide and are generated from an onboard 32Mhz clock. This implies that the timestamps will drift from module to module. Furthermore, since modules can only be initialized one-by-one, this implies that the modules will have different 'zero' times.

The timestamp is unpacked by SpecTcl as the first parameter of the event. A typical adcChannels entry for a Hytec 2530 might therefore look like:

```tcl
set adcChannels(hytec1) [list hytec1.timestamp \ hytec1.chan0 \ hytec1.chan1 \ hytec1.chan2 \ hytec1.chan3 \ hytec1.chan4 \ hytec1.chan5 \ hytec1.chan6 \ hytec1.chan7]
```

While SpecTcl’s initialization will create a timestamp spectrum (in the example above, named hytec1.timestamp, this spectrum is typically useless. Normally timestamps are used for rate spectra. To produce a rate spectrum:

1. Be sure you’ve loaded the constparam plugin into SpecTcl in your SpecTclRC.tcl file.
2. Use the constparam plugin (you can invoke it from SpecTclRC.tcl after the config file has been processed) to create a parameter whose value is 1 when the desired parameter(s) are present.
3. Create a strip chart spectrum where the timestamp is an ADC timestamp and the y axis parameter is the constparam you created in the previous step. Be sure that the time range is sufficient to take in the typical run length, and that the resolution is sufficiently compressed to allow you to see the evolution of the rate with time.
4. If you intend to create a sufficiently high resolution strip-chart, you may need to set the DisplayMegabytes in your SpecTclInit.tcl file sufficiently large to accommodate them in the Xamine shared memory.
3.3. Configuring the Scaler Display

Configuring a scaler display is done in two phases.

1. Each offset into the scaler buffer is given a name via the `channel` command.
2. Pages and strip charts are defined using the `page`, `display_single`, `display_ratio`, `stripparam` and `stripratio` configuration commands.

The `~/config/scalerChannels.tcl` script interprets the `daqconfig.tcl` script. It uses methods similar to those used by the SpecTcl configuration software. The only difference from your point of view is that scaler channels are a simple list stored in a global variable named `scalerChannels`. This list is used to generate and execute the appropriate `channel` commands needed to define the scaler channels.

You must then fill in the configuration file `~/config/scalerconfig.tcl` to define the scaler display pages. Let’s see how this works in practice, starting with the code needed in the `daqconfig.tcl` configuration file:

**Example 3-2. Scaler sections of daqconfig.tcl**

```tcl
# # Define scaler modules.
# madcscaler create deadtime -base 0x60000000
...

# Define the scaler channel names:

# Let the scaler display know about the modules and channels.
set scalerChannels [list deadtime totaltime]
...
```

The `madcscaler` command creates a scaler using the deadtime and timer counters in a Mesytec MADC32 adc module. This appears to the scaler readout as a pair of channels. The first counts when the scaler is busy, the second, counts all the time.

The `scalerChannels` variable is given the list of scaler channels, in the order in which they appear in the output buffer. In this case, we are only using the counters from the MADC32 and we name them `deadtime` and `totaltime`. 
Chapter 3. Configuration Files.

The `scalerconfig.tcl` needed to generate a single page scaler display showing these scalers and the dead-time ratio is:

**Example 3-3. Scaler display page configuration**

```tcl
source [file join ~ config scalerChannels.tcl]

page DeadTime {Dead time scalers}

display_ratio DeadTime deadtime totaltime
```

The first line `source` in the `scalerChannels.tcl` script which execute the `daqconfig.tcl` script. This defines the available scaler channels, associating them with offsets in to the scaler buffer.

The `page` command creates a scaler display page. The title of the page will be `Dead time scalers`. The page name will be `DeadTime`.

The `display_ratio` command adds a line to the `DeadTime` scaler page that displays the ratio of the `deadtime` and `totaltime` scalers read out by the madcscaler module.

### 3.4. Script Reference

This section contains reference information about the commands that have been added to Tcl to support configuring VM-USB data taking. In addition a reference to the set of meaningful script variables is supplied.

**adc**

**Name**

adc — Create/configure CAEN V775, V785, V792, V862 modules.

**Synopsis**

adc create *name* *base*

caen965 create *name* *base*

adc config *name* *option* *value* ...

52
Chapter 3. Configuration Files.

caenv956 config name option value ...

adc cget name

caenv956 cget name

DESCRIPTION

This command creates, configures and retrieves the configuration of CAEN V775, V785, V792, and V862, V965 digitizer modules.

Use the create subcommand to create a new adc providing it with a unique name that will be used to identify it in future commands. The base parameter is the base address of the module as set in the module rotary switches.

Use the config subcommand to configure a module named name the option options and legal values are described in the section OPTIONS below.

The cget subcommand returns as its value the configuration of the module name. The configuration is returned as a list of two element sublists where each sublist contains, in order, an option from OPTIONS below, and its value. Note that some values may themselves be lists.

When used with the CAEN V965 or other dual range digitizers via the caenv956 command, each parameter specified in the adcChannels array will result in two SpecTcl channels and corresponding raw spectra. The first will have .h appended to the name and will be the high range conversion while the second will have .l appended to the name and will be the low range value.

OPTIONS

-base value
   Allows you to reconfigure the base address of a module.

-thresholds values
   The value is a list of 32 values that are the module thresholds. Unless -smallthresholds has been configured to be true, these values are multiplied by 16 before being applied as the channel threshold values.

-smallthresholds bool
   The value is a boolean (e.g. on or off). A true boolean means that the threshold is applied as is a false boolean means the threshold value is multiplied by 16 and then applied.
Chapter 3. Configuration Files.

-ipl 0–7
   The interrupt priority level the module should use to request a VME bus interrupt. This defaults to 6 and should be set to zero to disable interrupts. Normally interrupts will be used to trigger an interrupt triggered stack. The default of 6 is historical in nature, in most cases for, the default should be overridden to zero.

-vector 0–255
   The interrupt vector the module should use. This is an integer and defaults to 0x80. The vector value is ignored if the module interrupts are disabled.

-highwater 0–31
   Defines how many events the module should accumulate before it interrupts. A value of zero also disables interrupts. The default value is 24 events. This is best suited for singles applications where allowing the module to accumulate a few events before being read is a good thing. This value will be ignored if the -ipl option has been configured to zero as that also disables module interrupts.

-geo 0–31
   Defines the geographical address that will be set in the module. If the module has a PAUX connector, this must be set to be the module’s physical location in the crate. This option must be correctly programmed in order to help SpecTcl form a mapping between data values and parameter names.

-fastclear value
   Defines the fast clear window for the module.

-supressrange bool
   If true the module will suppress overflows and underthreshold conversions. If not all channels will supply data for an event. In most cases, this should be set to false.

-timescale ns
   Sets the full scale range of the module if it is a V775. If the module is not a V775 TDC, this configuration parameter is silently ignored (the hardware allows the software to determine the module type). ns is the range of the TDC in nanoseconds and must be between 140 and 1200.

-iped value
   Sets the Iped register. See section 4.34 of e.g. the CAEN V965 manual. This register controls the amount of charge initially injected into the conversion circuit and is used to compensate for leakage current that may lower the conversion of a signal during the gate. The default value is 180. Valid values are 0 – 255, however see the description of the Iped register and pedestal injections in the manual.
EXAMPLES

Example 3-1. Sample ADC commands

```
adc create adc1 0x04000000
adc config adc1 -geo 12 -supressrange off -ipl 0 -vector 0
```

Defines a module with base address 0x04000000 to be in geographical address 12. Range suppression and interrupts are disabled.

NOTES

This command can actually initialize/configure V775 TDCs and V792, V862 QDCs in addition to the V785 ADC.

caenchain

Name

caenchain — Aggregate adc modules into CBLT readout chains.

Synopsis

```
caenchain create name
caenchain config name option value...
caenchain cget name
```

DESCRIPTION

The CAEN family of 32 channel digitizers (V775, V785, V792, V862) can be aggregated into CBLT readout chains. A CBLT readout chain can then be read at high performance using a single block read operation.
Chapter 3. Configuration Files.

A CBLT chain consists of a set of at least two modules in consecutive VME bus slots. From the point of view of defining the crate the left most and right most modules are special. (See OPTIONS below, the -module option).

CBLT chains require an additional base address that is used to read the modules. This address is also used as a multicast address to do a simultaneous clear of all modules in the chain as the readout is initialized (at the beginning of a run).

OPTIONS

-validate integer
  Provides the base address to be used to program the CBLT address of the module. Only the top two hexadecimal digits of an 8 digit hexadecimal address should be non zero. e.g. 0x12000000 is ok, but 0x12340000 is not.

-modules module-names
  Supplies the list of modules that should make up the chain. The first module must be the left most in the chain, the last module the right most. Other than that order is unimportant, however note that CBLT readouts always will go from left to right in VME crate. I therefore suggest that you supply the modules in left to right order. The modules must be a valid TCL list, e.g.: -modules [list adc1 adc2 adc3] is ok, -modules adc1 adc2 adc3 is not.

EXAMPLES

The example below takes three ADC modules and aggregates them into a chain for readout.

**Example 3-1. Using the caenchain command.**

```bash
adc create adc1 0x04000000
adc config adc1 -geo 12 -supressrange off -ipl 0 -vector 0
adc create adc2 0x05000000
adc config adc2 -geo 13 -supressrange off -ipl 0 -vector 0
adc create adc3 0x06000000
adc config adc3 -geo 14 -supressrange off -ipl 0 -vector 0
caenchain create chain
caenchain config chain -base 0x10000000 -modules [list adc1 adc2 adc3]
```
sis3820

**Name**

sis3820 — Create and configure SIS 3820 scaler modules

**Synopsis**

sis3820 create name base

sis3820 config name option value ...

sis3820 cget name

**DESCRIPTION**

 Creates and configures the SIS3820 32 channel scaler for use in a stack. The `create` subcommand creates a new module with a base address of `base`, and a name `name` which will be used to refer to this module in later configuration commands.

The scaler readout will result in an array of 32 channels of scaler data placed in the buffer. The first longword of this data is channel 0, the last, channel 1.

The `config` sub-command configures the options for the scaler `name`. The configuration is expressed as a series of one or more `option value` pairs. Options may have default values and are validity checked to ensure that valid values are supplied. See `OPTIONS` below for more information about the option keywords that are supported and their legal values.

The `cget` sub-command returns the current module configuration. The configuration is returned as a Tcl list of `option value` pairs. See `OPTIONS` below for a description of the options and values that are returned. You should not rely on the list being in any specific order. While the list will have a deterministic order, if additional option keywords are added later on, this order may change.
OPTIONS

-base value

Allows you to override the initial base address of the module, specified when the module was created.

EXAMPLES

The example below configures an SIS 3820 scaler to have a base address of 0x38000000

Example 3-1. Configuring an SIS3820 scaler module

```
sis3820 scaler1 0x38000000
```

v830

Name

v830 — Create and configure CAEN V830 32 channel scalers.

Synopsis

```
v830 create name base
v830 config name option value...
v830 cget name
```

DESCRIPTION

This command creates, configures and queries the configuration of CAEN V830 scaler modules. This is a latching scaler module. At present, not all functionality has been enabled. Specifically, the module code is now tailored to the typical use case of a run-time scaler module in the scaler stack.
The `create` subcommand creates a new module. The `base address` should be the VME base address of the module, as configured in the module’s rotary switches. The `name` parameter is a name that you assign to the module, and will be used to refer to the module in future `config` or `cget` commands.

The `config` subcommand configures the module `name`. The configuration is supplied as a set of `option value` pairs. More than one pair can appear on the same line, any number of `config` commands can be used and those that execute later can override those that execute earlier.

The `cget` subcommand returns the configuration of the module `name` as a list of `option value` pairs. You should not rely on the list being in any specific order. While the list order is deterministic within a version of the program it is possible that later versions will return the list in a different order.

**OPTIONS**

`-base address`

Allows you to override the base address of the module set at creation time with a new base `address`.

`-channels mask`

Provides a mask of enabled channels. The low order bit represents channel zero the high order bit, channel 31. Each channel for which a bit is present is enabled to count. The default value for this mask is `0xffffffff` which enables all channels.

`-dwelltime value`

This is used only if the `-trigger` is set to `periodic` and is the time between triggers in 400ns units. Each trigger will latch the current counter values into the MEB and, if `-autoreset` is true, clear them. If the `-ipl` and `-vector` are set, this can produce a backplane interrupt which, in turn, can trigger execution of a stack. The default value is zero which disables the periodic trigger.

`-header bool`

Enables or disables the inclusion of a header on data read from the MEB. For the format of the header, see figure 3.1 of the CAEN V820/V830 manual. Note that this header will only be present if the MEB is read out. A value of `true` enables the header while a value of `false` disables it. The default value is `false` which is appropriate for the case where this module will be used in a scaler stack.

`-trigger random | periodic | vme`

Determines the source the latch trigger. The values are as follows:

`random`

The external trigger input is used to trigger the latch.
periodic

The module will have a periodic trigger that is governed by the value of the -dwelltime option.

vme (default)

Triggers will be supplied by the stack that reads the module out.

-wb bool

Determines if, when the MEB is read, the scalers will be wide (32 bit counters) or narrow (24 bit counters tagged with the channel number). See figures 3.2 and 3.3 in the CAEN V820/830 manual for the data format in both cases.

A value of true enables wide (32 bit) mode. A value of false requests narrow (24 bit) mode. This only affects data read from the MEB. The counter data are always wide. The default value is true.

-autoreset bool

Configures the scaler to reset its counters after storing the data in the MEB or not. If true the counters are cleared after latching the data. If false not. The default value is true.

-geo slot

Programs the module geographical address. This value is only relevant if -header is true or -wide is false. The value can only be programmed for modules that do not have the PAUX connector. Modules with the PAUX connector read their geographical address from the backplane, and this value, if used in other places, must be set to the physical slot number the scaler is using (not a bad idea in any event).

-setgeo bool

If true the configuration code attempts to set the module’s GEO value from the -geo parameter. If not, the -geo value is simply assumed to document the position of the module to other software, and is ignored. The default value is false which is suitable for modules without a PAUX connector and for the normal use case of a module in the scaler stack where the GEO address is not relevant.

-ipl priority

Provides the interrupt priority level for the module’s VME interrupt. The V830 can interrupt when it has at least -highwatermark events in its MEB. This interrupt can be used to trigger a VM-USB stack.

VME bus interrupts are prioritized with the priority value ranging from 1 through 7. This priority determines the service order of simultaneous interrupts. Larger numbers are higher priorities. A value of 0 disables module interrupts. The default value is 0.
Chapter 3. Configuration Files.

-vector statusid

VME interrupts are identified via a status-id value presented by the interrupting device when the interrupt is acknowledged by the interrupt responder (VM-USB). In the case of the CAEN V830, this is a value from 0 through 255.

For historical reasons, this status-id value is also called the interrupt vector. The -vector option sets the value of the interrupt vector presented by the CAEN V830 when it interrupts. To interrupt, the module must also be configured with a nonzero -ipl and nonzero -highwatermark value.

Module interrupts can be used to trigger VM-USB stack execution.

-highwatermark value

Configures the value of the V830 Almost Full Level register. When non-zero, if -ipl is also non-zero, and -vector is also non-zero, when there is at least value events in the MEB, the module will initiate a VME bus interrupt. VME interrupts can be used to trigger VM-USB stacks. The default value is 1.

EXAMPLES

The following example shows how to set up a CAEN V830 scaler for inclusion in the scaler stack. The base address of the scaler is 0x80000000. The default configuration values are suitable for the scaler stack.

Example 3-1. Configuring a CAEN V830 scaler

v830 create scaler 0x80000000

v977

Name

v977 — Create and configure CAEN V977 Input registers

Synopsis

v977 create name ?option value...?
Chapter 3. Configuration Files.

v977 config name option value ?...?

v977 cget name

DESCRIPTION

This command supports the CAEN V977 I/O register. At this time support is only provided to read the module’s input register. Other than setup modes that echo the inputs to the outputs in some way, output is not supported.

The create subcommand creates a new module. The name parameter supplies a unique name by which that module will be known throughout the rest of the configuration file. The optional option value pairs provide additional configuration options and can be selected from the set of configuration options described in OPTIONS below.

The config subcommand allows you to further configure an existing module. The name parameter is the name you assigned to the module in the create operation. The option value pairs configure the module and can be selecte from the configuration options described in OPTIONS below.

The cget allows you to retrieve the current configuration of the module. The name parameter provides the name of the module as defined by the create subcommand. The result is a well formed Tcl list that consists of option name value pairs.

OPTIONS

Configuration options provide a mechanism to define where a module is located in the VME space as well as how the module should be prepared for data taking. A module is configured via the config subcommand. That command identifies the module via the name assigned to it in the create operation and provides configuration information in the form of a set of name value pairs.

Each name selects what is to be configured and each value provides the new value for that item. The configuration keywords supported and the meanings of their values are described below.

-base base-address

Defines the base address of the module. base-address must match the base address set in the module’s rotary switches. This address is used to determine how to access the module in VME address space.
- **inputmask mask**
  
  Provides the value of the module’s input mask register. The value of `mask` is programmed into the module’s input mask register at initialization time. Each bit set in the register prevents the corresponding front panel input from being seen by the module.

- **readmode mode**
  
  Together with the `-readandclear`, determines which register is actually read by the stack. Legal values are `singlehit` and `multihit`. If `singlehit` is selected (the default) is provided the module will read either the Single hit read register or the Singlehit read-clear register depending on the value of the `-readandclear`. If `multihit` is selected, the module’s multihit read or multihit read-clear register will be read.

  See the table at the end of this section for a complete listing of the combinations of read modes and read and clear settings and their implications for how the module is read.

- **outputmask mask**
  
  Provides a value for the output mask register. This defaults to zero.

- **interruptmask mask**
  
  Provides a value that will be programmed into the output mask register. The module can produce an interrupt if bits that are not masked off in this register are set in the pattern gated into the module. See also the `-ipl` and `-vetor` if you intend to use the module with interrupts.

- **readandclear true|false**
  
  Determines whether the module will be atomically cleared as it is read. This option together with the `-readmode` determines which module register read is added to the readout stack. For more information, see the table at the end of this section.

- **ipl interrupt-level**
  
  Determines the interrupt priority level used by the module when creating interrupt requests on the VME dataway. If `interrupt-level` is 0 (the default), interrupts will not be used.

- **vector status-id**
  
  When the module creates a VME dataway interrupt it provides the `status-id` as the interrupt vector when requested to by the interrupt handler module (VM-USB normally). If the `status-id` is 0 (default), interrupts are not generated on the dataway.

- **pattern true|false**
  
  If this is true, the pattern bit is set in the control register, and the module operates in pattern mode. If not, the module operates in I/O register mode.
sis3804

Name
sis3804 — Create and configure SIS 3804 scalers

Synopsis
sis3804 create name ?options...?

sis3804 config name options...

sis3804 cget name option

DESCRIPTION
This command creates and manipulates SIS3804 objects. The SIS3804 is an 8 channel latching scaler manufactured by Struck. The SIS3804 objects can configure and add readout instructions for this hardware to VM-USB stacks.

OPTIONS

-base base-address
Sets the base address of the module. This must match the value selected by the rotary switches on the module. The default for this option is 0, which is typically not what you want.

-refpulser boolean
Enables or disables the reference pulser. When enabled, the reference pulser disables the input to channel 1 and supplies a 25Mhz pulse frequency to that channel.

A value of true enables the pulser while false (the default) disables the reference pulser.

-disables mask
Allows you to disable specific channels from counting. The bottom 8 bits if set disable the corresponding channel of the pulse. For example if mask & 1 is nonzero, channel 1 (numbered from 1) is disabled, while if mask & 0x10 is nonzero, channel 8 is disabled.

The default value is zero which enables all channels.
Chapter 3. Configuration Files.

-autoclear boolean

If this option is true (the default), scalers are cleared atomically after being latched for readout. The NSCL data acquisition system scaler display program expects the scaler counters to be incremental values with the sums computed in software (so that overflows can be explicitly managed). If you have a special application for this scaler, e.g. to provide timestamps in an event, for which you do not want this behavior, you can set this option to false.

EXAMPLES

The example below creates a scaler at the manufacturer’s setting for the base address and configures it to have the 25Mhz reference pulser in channel 1:

Example 3-1. Configuring the SIS 3804 scaler

```
sis3804 create scaler1 -base 0x38383800
sis3804 config scaler1 -refpulser true
```

hytec

Name

hytec — Support the Hytec NADC 2530 adc module.

Synopsis

```
hytec create name ?options?
hytec config name ?options?
hytec cget name
```
Chapter 3. Configuration Files.

DESCRIPTION

This command provides support for the Hytec NADC2530. The NADC2530 is an 8 channel multi-event peak sensing adc. While the module is capable of autonomously creating histograms, this software does not support that capability as it’s much easier to treat that module differently if used in that way.

Beginning with the VM-USB readout program version 3.2-001, the software tags the ADC data with a user defined virtual slot number, and the module provides a 48 bit timestamp with each event if its firmware revision is 2530V305 or later.

As for all VM-USB module support commands, hytec is a command ensemble with the subcommands

create
Which creates an object for an NADC2530 adc which can be referred to by the name name.
Additional options can provide device configuration.

config
Which configures the existing device object name

cget
Which returns the module name’s configuration as an item list of name value pairs.

Configuration options are described in the OPTIONS section below.

OPTIONS

-csr address
This option must appear somewhere in the module configuration. It establishes the base address of the module’s register space. The NADC 2530 has two address spaces, register space, used to configure and control the module, and memory space where events get stored.

The address is an A24 address. this means it must be in the range 0x000000 through 0xffffffff. Each module must have a unique base address, and the address configured in the software must match the address configured in the module’s address jumpers.

-memory address
This option must appear somewhere in the module configuration. It establishes the base address of the module’s event buffer memory. Each module’s event memory must carve out a unique slice of A32 address space (values between 0x00000000 and 0xffffffff).
The buffer memory address space is software programmable. See the NADC2530 manual for the size of this address space as any overlap can cause corrupted data to be read from the modules.

-ipl \textit{irqlevel}

If you are using the module’s interrupt to trigger a stack execution, \textit{irqlevel} must be a nonzero value between 1 and 7, and will be the interrupt priority level on which the module will generate an interrupt request.

The value of this parameter defaults to 0 which disables module interrupts. See also the -vector option below.

-vector \textit{statusid}

If the module is being used to trigger a stack, \textit{statusid} should be the value the module will use to reply to the interrupt acknowledge cycle’s request for a status/id from the module.

The NADC2530 has a status/id 16 bits wide (between 1 and 65535 where zero disables the interrupt). The VM-USB, however only triggers on the bottom 8 bits of the status id field. Therefore you should use values between 1 and 255 unless you are directing the interrupt at other modules in the VME crate.

-lld \textit{millivolts}

Provides the low level threshold value in \textit{millivolts}. Due to the resolution of the threshold DAQ, the actual threshold value programmed may be slightly different than the one requested. See the manual for the relationship between \textit{millivolts} and threshold DAC values.

-hld \textit{volts}

Provides the high level discriminator value in floating point \textit{volts}. Due to the resolution of the threshold DAQ, the actual threshold value programmed may be slightly different than the one requested. See the manual for the relationship between \textit{volts} and threshold DAC values.

-events \textit{count}

Indicates that \textit{count} events must be in the event buffer before an interrupt will be signalled on the VME backplane. This item also configures how many events are required for the module to indicate that it has data. Therefore, if not being used with interrupts, this value should be programmed to its default value of 1

-id \textit{value}

Provides a 16 bit id (virtual slot number) that will be used to tag the data. The NADC2530 is not capable of providing a hardware virtual slot number as of firmware 2530V305, therefore the \textit{value} is inserted as a marker word prior to the data from the ADC.
Chapter 3. Configuration Files.

-zerosuppress on|off

If the value of this parameter is a true boolean, the channels that are outside the high and low level discriminators are not present in the data from the device. If the value is a boolean true, all 8 channels are present in the data.

EXAMPLES

The example below configures an NADC 2530 with a register base address of 0x400000, and a buffer address of 0x10000000. It leaves the thresholds wide open and does not enable module interrupts. A marker word 0xadcl is inserted prior to the data, and, if the discriminator levels are later set, data outside of them will be suppressed from the data stream:

Example 3-1. Sample Hytec 2530 configuration

```bash
hytec create adc
hytec config adc -csr 0x400000 -memory 0x10000000
hytec config adc -lld 0.0 -hld 8.191 -events 1 -id 0xadcl -zerosuppress on
```

DATA FORMAT

Readout prepends the data from the ADC with three additional data words. This section describes what to expect in the event file for this module.

The first word of data from this module will be the value of the -id optin. This defaults to zero if not supplied. This is followed by a mask word, and then a count word. These two words allow you to determine the number of longwords of ADC data that follow.

Suppose the mask word is mask and the count word is count. mask & count computes the number of longwords of ADC data that follow in the data packet from this module.

Following these three words, the data from the ADC as shown in section 4.3.1 of the manual is inserted in the event. Note that:

1. The ADC may convert serveral times for a single channel if multiple signal peaks are detected within the gate time. SpecTcl will only pay attention to the largest of these conversions for a specific channel.
2. When the system starts up, or if the busy lock out logic is not perfect, and allows gates to go to the ADC when the system is busy, you may see additional events or event fragments. SpecTcl will only pay attention to the data from the first event (it will stop processing channels after the first) trailer word or after the longword count described above is exhausted.
3. When the system initially starts, there may be a large number of gates prior to the VM-USB completing initialization... depending on when it actually asserts busy. Once more SpecTcl will only histogram the first of these events from the ADC. As subsequent data from the ADC should be synchronized to the IN1 trigger, this should at most affect the first event.

4. When setting up the SpecTcl parameters for this module, remember that the module provides 9 parameters. The first of these is a 48 bit timestamp. The remaining 8 are the adc parameters. SpecTcl is not able to guess how you want to set up the timestamp spectrum. In general this is not a problem as usually you will be using the timestamp to generate rate stripchart spectra rather than looking at the timestamp itself.

5. When looking at SpecTcl timestamp based spectra you may see significant gaps in the time online. This happens because of the way data are sampled to SpecTcl from the online system. SpecTcl may miss complete buffers of data online if it is not able to keep up with the data rate. If you process the event file for that run offline, these gaps disappear.

madc

**Name**

*madc* — Acquire events from Mesytec MADC32 ADC.

**Synopsis**

*madc create* name *?options?*

*madc config* name *?options?*

*madc cget* name

**DESCRIPTION**

The *madc* command provides scripted support for the Mesytec 32 channel peak sensing adc module. For scaler support for this module’s dead time counters, see the *madcaler* command.

As with all VM-USB module support commands, *madc* is a command ensemble with subcommands that *create* and *configure* modules as well as *cget* which introspects a module configuration.
**create** creates an object with the specified *name*. Additional options are treated like configuration options. **config** configures an existing module, and **cget** returns a list of configuration name/value pairs that describe the configuration of the module.

It is important to note that the module configuration does not actually get loaded until the run is initialized. The order in which configuration parameter are supplied is therefore unimportant. Think of the configuration options as being accumulated and then applied as the run starts. Only modules that are in stack are configured.

**OPTIONS**

**-base address**

*address* must be the module base address as configured in its rotary switches. This base address is used to access the module’s register and event memory.

Each module must be programmed and hardware configured with a different base address. The address used will be an A32 VME address.

**-id vsn**

*vsn* will be used as the module’s identifier or virtual slot number. The *vsn* will be encoded into the event data that is returned by the module. This, in turn is normally used by event decoders to determine which parameters the channels of the module should be unpacked into.

Each module should be given a unique *vsn*.

**-ipl irqlevel**

If the module will be used to trigger an interrupt driven stack, the *irqlevel* parameter must be programmed to a valid non zero interrupt priority level (1 through 7). This must match the interrupt priority level used to trigger the stack.

The default value of 0 disables module interrupts.

**-vector statusId**

If the module will be used to trigger an interrupt driven stack, the *statusId* must be programmed to a non zero 8 bit status id, or *vector* (between 1 and 255).

The value used must match the value of the **-vector** configuration parameter used to trigger the stack.
Chapter 3. Configuration Files.

-timestamp onoff

This option controls whether or not the module tags each event with a trigger number or with a timestamp (see also the -timingsource and -timingdivisor options).

The onoff is a boolean value. If true, the module tags events with a timestamp. If false, with a trigger number.

-gatemode mode

The MADC32 has a pair of gate inputs. The inputs may be used either as separate gates, where each gate controls 16 of the 32 channels, or as common where either input will gate all 32 channels.

The value mode should be either separate or common.

-gategenerator onoff

The module can either use the gates as provided or can insert a gate and delay generator between the gate inputs and the actual gates seen by the ADCs. Since in most cases, gate must be stretched and timed to match the ADC inputs, this feature can reduce the external electronics needed to properly gate the adc.

The onoff is a boolean that if true enables this gate generator, if false, disables it. See also the -holddelays and -holdwidths configuration parameters that control the gate and delay parameters for each of these resources.

-holddelays delayList

If the gate generators are enabled (see -gategenerator above), the delayList is a Tcl list consisting of the two delay parameters, one for each of the gate and delay generators. See the MADC32 manual for a description of the meaning of these values, which are just the values programmed into the module registers.

-holdwidths widthList

If the gate generators are enabled (see -gategenerator above), the widthList is a Tcl list consisting of the two gate and delay generator width parameters. See the MADC32 manual for a description of the meaning of these values, which are just the values programmed into the module registers.

-inputrange rangeSelector

Programs the input range for the module. The rangeSelector must 4v, 8v, or 10v. Where the selector represents the input range in volts.
Chapter 3. Configuration Files.

-ecltermination onoff
This parameter when true enables the ECL input termination. If disabled, the termination is off. If you are bussing the ECL inputs, only the final module in the bus should have termination enabled, all other modules, should have termination turned off.

-ecltiming onoff
This parameter, when true enables the gate1 ECL input to be a clock source for the timestamp if true. If false, the ECL G1 input is an ECL gate1.

-nimtiming onoff
If true, enables the NIM Gate1 input to be a clock source for the timestamp. If not, the NIM Gate1 input is an adc gate.

-timingsource sourceName
Specifies the source of the clock for timestamps. If external, whichever of the NIM or ECL GATE1 inputs are enabled is the clock source. If vme the VME 16Mhz backplane clock is the clockk.

-timingdivisor log2
Specifies a scale-down value for the timestamp clock. At the time I’m typing this, this value is log base 2 of the scale down, that is the final scale down is $1 << \text{log2}$. By the time we get installed, I am supposed to have some firmware that will allow this to be a 16 bit direct scaledown (e.g. the scaledown would be between 1 and 65535).

-thresholds valueList
Supplies the per channel thresholds for the adc. Channels which convert below their threshold are suppressed from the data stream reducing both data volume and dead-time. The valueList is a 32 element Tcl list of the integer thresholds.

Note that at the time I’m typing this, channel thresholds have not yet been implemented in the firmware. The firmware I bring with me at installation time will hopefully implement this feature.

-nimbusy busyselect
This option selects which signal is presented at the NIM busy output lemo connector. By default, this will be the module busy. The busyselect can be any of the following strings:

busy
The module busy is output. This is the default.

gate0
The Gate0 signal is output. If the internal gate and delay generator is enabled for Gate0, the output will be the output of the gate and delay generator. This provides a mechanism to check the gate timing on a scope if you are using the internal gate and delay generators.
Chapter 3. Configuration Files.

gate1

The Gate1 signal is output. If the internal gate and delay generator is enabled for Gate1, the output will be the output of the gate and delay generator. This provides a mechanism to check the gate timing on a scope if you are using the internal gate and delay generators.

It is not clear to me what happens if you are using module common gates.

cbus

The CBUS output is reflected here.

-multievent boolean

Allows the module to be used in multi-event mode. This is normally done in conjunction with the madcchain configuration command. It also usually requires a custom SpecTcl version to handle the data from this device.

The default value for this parameter is false which runs the module in single event mode.

-irqthreshold integer

Sets the interrupt threshold. When a number of complete events have put at least this number of longwords in the fifo, if interrupts are enabled, the module will interrupt.

-resolution 2k|4k|4k hires|8k|8k hires

Sets the resolution of the module. This has an impact on the conversion time.

EXAMPLES

Example 3-1. Sample use of madc command

```
set madcTimeDivisor 14
madc create adc -base 0x40000000 -id 5 -ipl 0
madc config adc -gatemode common -gategenerator disabled
madc config adc -inputrange 8v
madc config adc -timestamp on -timingsource vme -timingdivisor $madcTimeDivisor
for {set i 0} {$i < 32} {incr i} {
    lappend thresholds 0
}
madc config adc -thresholds $thresholds
```

This command creates an object to manage an MADC 32 whose base address is 0x40000000. The module will be referred to by the symbolic name: adc

This line illustrates substitution of a Tcl variable for a parameter value. Tcl variable substitution is textual, so you can also use variables to hold option names, though that may be a bit odd.

This highlights the fact that the configuration file is really a configuration program. The loop creates a variable named thresholds that contains a list of 32 zeroes. This list will be used to program the adc thresholds. Normally these values will neither be zero nor uniform from channel to channel. It may be best to read them from some external file.

This command uses the thresholds variable and programs the channel thresholds of the ADC.

madccchain

Name

madccchain — Support CBLT chains of MADC32 modules.

Synopsis

madccchain create name

madccchain config name ?options

madccchain cget name

DESCRIPTION

This module creates and configures chains of MADC32 modules. If these modules are, in turn, run in multi-event mode, a specialized SpecTcl will be needed to unpack the data. It is possible, however to use single event mode with CBLT readout and use ’normal’ SpecTcl unpacking.

The create creates a new MADC32 chain. name will be used to refer to this chain during configuration. At any time the cget returns the configuration of the chain as a list of parameter-name parameter-value pairs.

The config configures an MADC32 chain. A set of option name option value argument pairs should follow the chain name on the commandline. OPTIONS below describes the options and their legal values.
Chapter 3. Configuration Files.

OPTIONS

This section describes the configuration options supported by the `cmadcchain` command.

- **-cbltaddress** `base-address`
  
  Defines the base address to which the CBLT reads will be directed. When the modules in the chain are initialized, this address will be programmed as the CBLT base address. Note that only the top 8 bits of this value are used.

- **-mcastaddress** `base-address`
  
  Defines the base address for the chain’s multicast address. The multicast address is used to perform synchronous initialization and time-stamp clears. This address will be programmed as the MCAST base address for all modules in the chain. Only the top 8 bits of the `base-address` have any meaning.

- **-maxwordspermodule** `longword-count`
  
  Defines the maximum words that can be read from each module. This should be a number between 1 and 1024. This value should usually be larger than the `-irqthreshold` option programmed into the MADC32 modules in the chain. For each block read, the module will return no more data than the complete event that causes the number of longwords read from the module to exceed this value.

- **-modules** `module-name-list`
  
  Defines the list of MADC32 modules that make up the chain. The value for this option should be a well formulated Tcl list containing names of `madc` modules. There is a restriction on the order of these names. The first name in the list must be the leftmost module in the crate and the last name must be the right most module in the chain.

  For practical purposes, to limit confusion, I generally enumerate the modules from left (low slot number) to right (higher slot number).

madcscaler

**Name**

madcscaler — Support dead-time counters in MADC32 as scalers.

**Synopsis**

```bash
madcscaler create name ?options?
```
madcscaler config name ?options?

madcscaler cget name

DESCRIPTION

The Mesytec MADC 32 module provides two counters. These counters count time and ADC busy time (time between the adc gate and readout completion). Reading these two values as periodic scalers in a scaler stack, allows for the computation of dead-time ratios without the use of an additional external scaler module.

The madcscaler command supports configuring MADC32 modules for use as dead-time scalers.

The madcscaler command provides the usual ensemble of subcommands; create to create a named module object, config to configure a previously existing object by name, and cget to obtain the configuration of an existing named object.

OPTIONS below describes the configuration options.

OPTIONS

-base baseAddress

Provides the base address of the module as configured in its rotary switches.

mase

Name

mase — Support for XLM with MASE firmware.

Synopsis

mase create name ?options?

mase config name ?options?
Chapter 3. Configuration Files.

mase cget name

DESCRIPTION

The mase command allows you to create and configure modules that are XLM-VV FGPA modules with firmware for the Indiana MASE data acquisition subsystem. Please note that this module is a block mode device, in the sense that each VM-USB trigger may result in multiple events being present in the XLM memory by the time the VM-USB reads the data. When used with other modules, you must arrange for event building at some point in the system to ensure that data from the MASE module are coherently assembled with data from other devices.

The module object is created via the create subcommand. The name on this defines a unique name which will be used to refer to this module in future commands directed at it. The optional options are name value pairs used to configure the module. See OPTIONS below for a list of the supported options.

The config subcommand can be used to set or modify configuration parameters after the module is created. The name is the name of the module as defined by the create subcommand. The options are once more name value pairs that are described in the OPTIONS section below.

The cget subcommand is can be used to ask a module about its current configuration. The module configuration is returned as a Tcl list of name value pairs where the name is the name of a configuration option described in the OPTIONS section and the value is the value of that configuration parameter.

OPTIONS

-base base-address

Supplies the base address of the XLM module. The XLM base address is determined by the slot it occupies in a VME backplane with V430 extensions (V430 supplies additional voltages and the slot location on a middle ‘JAUX’ connector). The base address must be an unsigned integer. There is no default value for this configuration option. The value supplied must match the actual board’s base address.

The XLM-VV must be used in a V430 backplane as it has no other mechanism for the board to set the base address. For information about the JAUX connector see e.g. http://cdsweb.cern.ch/record/1201456/files/Blanchetti_001.pdf (http://cdsweb.cern.ch/record/1201456/files/Blanchetti_001.pdf) Specifically page 10 and page 25 and beyond.
-firmware **firmware-path**

*firmware-path* provides the path to the XLM firmware file. In the MASE module, the firmware is loaded prior to the start of a run. The file path can be relative or absolute, however environment variables and tilde expansions are not performed by the module driver. The Tcl script however may use the **file normalize** and the **envname** to perform these substitutions prior to providing the final filename to this option.

### tdcv1x90

**Name**

*tdcv1x90* — Provide support for the CAEN V1x90 TDC family.

**Synopsis**

*tdcv1x90* create name ?options...?

*tdcv1x90* config name ?options?

*tdcv1x90* cget name

**DESCRIPTION**

Creates, configures and introspects CAEN V1x90 TDCs. These modules are only supported in trigger matching mode. Note that the trigger time is only good to the FPGA clock, to get precise trigger relative timing you will need to also digitize the trigger itself. SpecTcl supports doing a digital subtraction of the trigger channel from all other channels to get precise trigger relative timing.

The *create* subcommand creates a new TDC named *name* the *options...* optional parameter are configuration option value pairs as described in OPTIONS below. The *config* subcommand locates the TDC named *name* and further configures it via the options described in OPTIONS below.

The *cget* subcommand returns as a value the module configuration. The configuration is returned as a list. Each element of the list is a two element sublist consisting of the configuration option name and its current value.
OPTIONS

-**base** base-address

  provides the module base address as set by the rotary switches on the board.

-**vsn** geo

  Provides the virtual slot number. This value will appear in the data from the module in the GEO field. The \textit{geo} value must be between 0 and 31 inclusive.

-**ipl** \textit{n}

  Sets the interrupt priority level when using the device interrupts. If this is 0 interrupts are disabled. Legal value are between 0 and 7 inclusive. See the -**vector** switch as well. This defaults to 0.

-**vector** \textit{nnn}

  Sets the interrupt status id value for the module if used in interrupt mode. If this is 0, interrupts are disabled. See -**ipl** as well. This defaults to 0. The values must be integers in the range 0 - 255 inclusive.

-**termination** none|switch|on

  Sets the module termination. The value can be any of the following:

  \textbf{none}

  No termination will be supplied by the module.

  \textbf{switch}

  Termination will be controlled by switches in the module

  \textbf{on}

  Termination is enabled.

  The default is \textbf{on}

-**tagtime** on|off

  Controls whether or not the trigger time will be included in the data. Note that the trigger time is only accurate to within one tick of the 80Mhz FPGA clock.

-**highwatermark** \textit{n}

  Determines the number of events that must be buffered by the TDC to generate an interrupt. The value must be an integer in the range 0..65535 inclusive. The default value 1, means an interrupt is generated whenever there is at least one event in the TDC and the TDC interrupts are enabled.

-**ecloutput** ready|full|almostfull|error

  Determines which signal is presented at the programmable ECL output pins. See the CAEN V1190/1290 manuals for information about the possible values. The default is \textit{ready}
Chapter 3. Configuration Files.

-`window n`

The width of the trigger matching window in 25ns units. This is sets the effective range of the TDC when simulating a common stop or common start TDC. See also `-offset`, `-extramargin` and `-rejectmargin` and see the section of the TDC manual that describes trigger matching mode. The default value is 40 which corresponds to a 1usec matching window.

-`offset n`

Determines when the trigger matching window starts relative to the gate. A positive offset starts the match window after the gate while a negative offset starts the match window prior to the gate. The values are integers and are in 25ns units. See section 2.4.1 for additional constraints. Note that the actual window start time will jitter by +/-25ns, and therefore you should use a reference channel to get good gate relative timing. Values must be in the range -2048 to 40 inclusive... Defaults to -40 or 1usec prior to the gate.

-`extramargin n`

The extra search margin for hits. This is the additional time during after the matching window during which the module will search for hits that are within the window before declaring the event. This is needed because hits are searched for in the module’s L1 buffer. Contention may prevent matching hits from being written to the L1 buffer for some time after they have actually occurred. See 2.4.1 in the manual. The units of this value are also 25ns. Defaults to 8 which is 200ns.

-`rejectmargin n`

The reject margin. This is also in 25ns units. While the module is in continuous storage, it maintains a reject counter that flushes hits from the buffer when the trigger window is not active. The module will only retain hits for the width of the trigger window + offset + reject margin before throwing them away if there is no trigger. This ensures the TDC buffer does not overflow and that the search for matching hits on the trigger is rapid. Defaults to 4 which is 100ns.

-`triggerrelative enabled|disabled`

If enabled, the trigger time is subtracted from all the hits. Note again that the trigger time is only precise to 25ns. Precise timing relative to the trigger can only be done by subtracting a digitized trigger time from the hits. Defaults to `enabled`.

-`edgedetect pair | leading | trailing | both`

Sets the module edge detect mode. Figure 2.2 provides trailing, both} documentation about what this means; ‘pair’ provides the width of a pulse in a channel, ‘leading’ provides a hit time at the leading edge of a pulse, ‘trailing’ provides a hit time at the trailing edge of an input pulse, and ‘both’ provides the time of both the leading and trailing edges of a pulse. Defaults to `leading`.

-`edgeresolution 800ps | 200ps | 100sp | 25ps`

Selects the resolution for the leading/trailing resolution. It is an error to use 25ps if the module is not a V1290. Defaults to 100ps.

-`leresolution 100ps | 200ps | 400ps | 800ps | 1.6ns | 3.12ns | 6.25ns | 12.5ns`

In leading, trailing and both mode, sets the resolution with which the leading edge is detected. Defaults to 100ps.
Chapter 3. Configuration Files.

- **-widthresolution** 100ps | 200ps | 400ps | 800ps | 1.6ns | 3.2ns | 6.25ns | 12.5ns | 400ns | 800ns

Sets the resolution with which a pulse width is measured in pair mode. Defaults to 100ps.

- **-deadtime** 5ns | 10ns | 30ns | Sets 100ns

the double hit resolution. Defaults to 5ns.

- **-encapsulatechip** true | false

If true, the data from a chip is encapsulated as shown in figures 6.2/6.4 by a TDC Chip header/trailer. Default enabled.

- **-maxhits** 0 | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | infinite

Specifies the maximum number of hits for each TDC chip in an event. Note that a TDC chip has 32 channels.

- **-errormark** true | false

If true, when an error is detected, an error mark item is placed in the output buffer. Figure 6.5 describes the format of this item. Defaults to true.

- **-errorbypass** on | off

If enabled, a TDC that reports an error will not be read out for that event. Default: on.

- **-globaloffset** \((n \ m)\)

Provides the global offset/vernier offset. Default: \((0 \ 0)\).

- **-channeladjusts**

Arbitrary number of elements that can adjust the value of individual channels by adding a positive offset to them. The value of this is a list of two element lists where each element consist of a channel number and its offset. e.g. \({{10 \ 6} \ {32 \ 5}}\) sets the channel offsets for channel 10 to 6 and for channel 32 to 5.

- **-refchannel** \(n\)

Sets the reference channel. The value of this channel is subtracted from all other channels to produce high precision times. Defaults to 0. This option is only used by SpecTcl and does not influence the way the TDC is initialized or read.

- **-depth** \(n\)

Sets the number of hits to retain for each channel for the purposes of histogramming. Defaults to 16. This option is only processed by SpecTcl. It setting does not influence the setup or readout of the TDC. Note however that setting this value larger than the value of \(-\text{maxhits}\) is probably not very useful.

- **-channelcount** \(n\)

Sets the number of channels the model of the TDC being used has. This is only used to setup SpecTcl’s histogramming. The value should be one of 16, 32, 64, or 128, however this is not checked by the processing code for this option.
METHODS

create name base

Creates a new instance of a CAEN multihit TDC from the V1x90 family of digitizers. name is used to refer to this module from now on. The base sets the base address of the module.

config name options

Configures an existing TDC name options are option name value pairs described above.

cget name

Returns the module name’s configuration as a list of name value pairs.

EXAMPLES

v1729a

Name

v1729a — CAEN V1729A waveform digitizer.

Synopsis

v1729a create name base

v1729a config name options

v1729a cget name

DESCRIPTION

This command ensemble provides access to the CAEN V1729A waveform digitizer. The three subcommands allow creation configuration and introspection. See METHODS below for more information about these subcommands and their formats.
The options described in OPTIONS below allow data taking and analysis to be configured according the needs of the application.

**OPTIONS**

**-base base**

Provides the module base address. The base must match the base address set in the rotary switches of the module.

**-threshold n**

Threshold for internal trigger. The default value is 4095

**-pretrigger n**

Pretrigger time in samples. Defaults to 10240

**-posttrigger n**

Post trigger time in samples. Defaults to 64

**-triggersource external | internal | both**

Specifies trigger source: his can also be ’internal’, or ’both’. Defaults to external

**-triggeredge rising | falling**

Specifies which edge of the trigger is used. Default is rising

**-triggermask on | off**

Specifies whether the trigger is masked via the EXT_EN_TRIGGER if this is ’on’. EXT_EN_TRIGGER is required to enable triggers to fire. Defaults to off.

**-triggerchannels n**

Only used if internal triggers are allowed. Set one bit for each channel from which triggers are allowed.

**-poststoplatency n**

Sets the post trigger latency register in samples. Defaults to 4

**-postlatencypretrig n**

Sets the value of the post latency pretrigger register. Defaults to 1

**-samplingfreq 2ghz | 1gzh | 500mhz**

Sampling frequency. Default value is 2ghz

**-delay n**

uSec to delay prior to executing the readout. this can be set to zero if the modle interrupt is used to trigger the data acquisition else it should be set to on the order of 700usec depending on triggering
parameters. 650usec from end of the signal are required to transfer data. the remaining time depends on the time of the computer trigger relative to the time of the signal end. When setting this recall that the module requires 650usec to digitize and transfer data from the analog memory to the readout memory.

METHODS

Subcommands recognized by this module are:

```
create name base

Creates a new module that can be referred to by name. base is the module base address. This can be overridden by the -base configuration parameter.
```

```
config name options

Configures the module name. The options are configuration option value pairs described in the OPTIONS section above.
```

```
cget name

Retrieves the module configuration as a list of name value pairs.
```

stack

**Name**

`stack` — Compose and configure VM-USB readout stacks.

**Synopsis**

```
stack create name

stack config name option value...

stack cget name
```
**DESCRIPTION**

The VM-USB supports eight readout *stacks*. A stack can be thought of as a set of VME operations the VM-USB should perform in response to some trigger condition. Reads performed by a stack are placed in an event buffer. Writes occur but result in no data. At appropriate times, the VM-USB transmits buffers of read events to the host computer over its USB interface. Stacks provide avoid the high latency of the USB bus by pushing the readout intelligence for events into the VM-USB, and out of the host.

The VM-USB supports 8 stacks, and three kinds of triggers for stacks. While all stacks can be interrupt triggered, we simplify the VM-USB usage by only allowing stacks 2-7 to accept interrupt triggers, defining stack 0 to be only triggerable on the NIM1 input, and stack 1 to only be triggered with some time periodicity.

The stack command allows you to compose stacks and specify their trigger conditions. When data taking is enabled, all defined stacks are loaded into the VM-USB and their triggers set up. Stacks are composed by specifying the set of modules they should read via the \(-\text{modules}\) configuration option. All modules in a stack are initialized at stack load time and read when the stack is triggered, in the order in which they were specified.

The create subcommand creates a stack named \(\text{name}\). \(\text{name}\) will be used by you in future config and cget subcommands to refer to this stack.

The config subcommand allows you to configure the stack contents and trigger conditions of the stack. \(\text{name}\) determines which stack will be configured. The configuration is specified via a set of \(\text{option value}\) pairs. These are described in OPTIONS below.

The cget command returns the configuration of the stack as a well formed Tcl list of \(\text{option value}\) pairs.

**OPTIONS**

\(-\text{trigger} \ \text{nim1} \mid \text{scaler} \mid \text{interrupt}\)

Defines the trigger source for the stack. When the designated trigger is present, the stack will execute. \(\text{nim1}\) triggers the stack on a nim logic true pulse to the IN1 input of the VM-USB. This forces the stack to be VM-USB stack number zero as that is the only stack that can be triggered by the IN1.

\(\text{scaler}\) triggers the stack on the periodic scaler. This forces the stack to be stack number 1 as that is the only periodically triggerable stack. We restrict periodic triggers to time based periodicity rather than event division periodicity. The \(-\text{period}\) option defined below must be configured as well to define the periodicity of the scaler stack.
Chapter 3. Configuration Files.

interrupt indicates the stack will be interrupt triggered. This can be used for stacks 2 through 7 as defined by the -stack configuration parameter. The actual VME interrupt that will trigger the stack is defined by both its interrupt priority level -ipl and the status id the interrupt source places on the backplane in response to the interrupt acknowledge cycle, and specified with the -vector configuration parameter.

-period seconds

Defines the number of seconds (integer) between scaler stack triggers. This is ignored for all stacks that are not scaler -trigger-ed. The VM-USB manual defines the range of legal values for this option.

-stack seconds

Defines the stack number for interrupt -trigger-ed stacks. This is an integer value in the range 2-7 inclusive. This will also select the interrupt register used to define the trigger for the stack.

-vector status-id

For interrupt -trigger-ed stacks, this defines the status Id that must be presented by the interrupt source in response to an interrupt acknowledge to trigger the list. This taken together with the value of the -ipl configuration option defines the trigger condition for these stacks. The status-id must be a value between 0 and 255.

-ipl priority

Defines the interrupt priority level that of the interrupt that will trigger the stack. An interrupter places a 3 bit interrupt priority level (IPL) on the VME bus when it requests an interrupt. The IPL must be nonzero and is intended to reflect the priority of the interrupt (7 highest, 1 lowest), although, in fact, the VME standard allows interrupt responders to treat this value in any way they want.

The VM-USB uses the specified priority as part of the trigger condition for an interrupt -trigger-ed stack. When an acknowledged interrupt matches both the -ipl, and the -vector configuration parameters of a stack the stack is triggered.

-delay microseconds

The VM-USB allows you to define a delay between the stack trigger condition and the actual start of stack execution. This is normally intended to be used with stacks that are triggered on nim1, as there may be a significant time between the generation of an external trigger and the conversion of the digitizers associated with the trigger. As I read the VM-USB manual, however this delay applies to all stacks.

The microseconds is the number of microseconds to delay (0-255) between triggers and stack executions. Because of my understanding of the intent of this delay, the -delay option is ignored for all stacks that are not nim1 -trigger-ed.
In the very unlikely event that you need to apply a trigger delay to interrupt triggered stacks (most hardware interrupts when data are ready not on a trigger), and you are not using nim1 triggered stacks, simply build a non-empty nim1 triggered stack, set it’s -delay option to the required delay and never trigger that stack (don’t cable anything to the IN1 input).

-modules module-list

Defines the set of modules that will be read out. This can be a valid Tcl list of any set of modules that have been defined so far. The modules are specified in the order in which readout commands for them will be added to the stack. If you have built a caenchain add that to the stack rather than adding the individual modules.

Lists of modules must be valid Tcl lists thus:

```
stack config astack -modules adc1 adc2 adc3 ;  # Incorrect
stack config astack -modules [list adc1 adc2 adc3];  # Correct.
```

**EXAMPLES**

The example below is a rather complete demonstration of how to define a set of modules, adc’s and scalers, aggregate the three adc’s into a caenchain, and read that chain as a stack triggered on NIM1, and read a pair of scalers triggered every two seconds.

**Example 3-1. Building Stacks**

```
adc create adc1 0x04000000
adc config adc1 -geo 12 -supressrange off -ipl 0 -vector 0
adc create adc2 0x05000000
adc config adc2 -geo 13 -supressrange off -ipl 0 -vector 0
adc create adc3 0x06000000
adc config adc3 -geo 14 -supressrange off -ipl 0 -vector 0
caenchain create chain
caenchain config chain -base 0x10000000 -modules [list adc1 adc2 adc3]
sis3820 create scaler1 0x35000000
v830  create scaler2 0x80000000
sis3804 create scaler3 -base 0x38383800
```
variables

**Name**
variables — Description of Tcl variables expected by other scripts.

**Synopsis**
- bufferMultipliers;
- detectors(*detectorname*);
- detectorChannels(*detectorname*);
- adcChannels(*modulename*);
- scalerChannels;
- timeCalibration;

**DESCRIPTION**

The daqconfig.tcl script is directly interpreted by the readout framework, and used to construct stacks that are used for data taking. It also serves as a description of the experiment that, with a bit of help can be used to configure SpecTcl, the scaler display program, and the Event file to xml conversion software.

Each of these applications requires, as a starting point, the same information needed to configure the readout software. Each application, will require additional information not needed to configure the readout program. This information must be stored in a set of global variables and global arrays.

For example, SpecTcl must decode the data produced by the readout software event by event into a set of named parameters. Therefore, in addition to the order of modules in a stack and their module types,
Chapter 3. Configuration Files.

SpecTcl must know the names of the parameters associated with each input for each electronics module. This information must be stored in the global array `adcChannels`.

The intent of this somewhat complex scheme is to encapsulate all knowledge of the experiment needed by the software in a single file. This minimizes the potential for inconsistencies that might develop if each application used a separate configuration file.

An important point to note is that Tcl arrays have textual indices rather than numeric indices. This allows them to serve as lookup tables by e.g. module name.

VARIABLES

`bufferMultiplier`

`bufferMultiplier` controls the aggregation of multiple VM-USB buffers into a single NSCLDAQ output buffer. The value is an integer that specifies how many VM-USB buffers can be aggregated into a single NSCL output buffer.

This is intended to be used if you are taking data from devices that may return a large amount of data for each event (such as the CAEN V1729a waveform digitizer). In that case, mapping a single VM-USB buffer to a single NSCLDAQ buffer may either not work or be inefficient.

If omitted the value is defaulted to 1. The range of suitable values is 1 through 4 inclusive. Values larger than 4 run afoul of the fact that NSCL buffer word counts are 16 bits wide.

`detectors`

The `detectors` variable is an array that is indexed by the name of a detector. The value of each array element is a two element list. The first element of the list is the type of detector. The second element of the list is the identification of the detector. The Tcl `list` command can be used to create this list. List elements that contain spaces or characters that have special meaning to Tcl, should be quoted with curly braces e.g.: `{I have spaces}`.

This information is used by the event file to xml converter to produce the metadata tags and their contents.

Example 3-1. Defining an element of the `detectors` array

```
set detectors(dssd1) [list dssd X-103/2]
```
Chapter 3. Configuration Files.

detectorChannels

This is a Tcl array indexed by detector name. The value of each element is a list of the SpecTcl parameters that are provided by that detector. This information is used by the event file to xml converter to produce the detector metadata. This metadata is used to relate detector outputs to electronics inputs.

Example 3-2. Defining detector outputs

```tcl
set hpge hpge.e
...
set detectorChannels(hpge) $hpge
```

The example above uses a Tcl variable to set the value of `detectorChannels(hpge)`. This ensures that the same names will be used when filling in the `adcChannels` array.

adcChannels

This is a Tcl array indexed by the name of an electronics module. The elements of the array are the SpecTcl parameter names that are assigned to the inputs of that electronics module. This should be a properly formatted Tcl list.

Continuing our example from `detectorChannels` above:

Example 3-3. Naming digitizer inputs

```tcl
set hpge hpge.e
...
madc create hpge -base 0x60000000 -id 6 -ipl 0
...
set adcChannels(hpge) $hpge
```

Taken in conjunction with the `detectorChannels` example, we have described a detector named `hpge` that contributes a signal named `hpge.e`. This signal in turn is plugged into input number 0 of the Mesytec ADC named `hpge` which has a virtual slot number (id) of `id`. SpecTcl will unpack that module so that input number 0 will be unpacked to the parameter named `hpge.e` on which histograms can be defined, and gates created.

scalerChannels

This is a scalar variable. It contains the channel names to be given to the scaler channels in the order in which they are read into the scaler buffer. An additional configuration file must define display pages from these channels. This information is also used to label scaler values in by the event file to XML converter.
Example 3-4. Defining scaler channels with the madc32scaler

```
madcscaler create deadtime -base 0x60000000
stack create scaler
stack config scaler -trigger scaler -period 2 -modules deadtime
set scalerChannels [list deadtime totaltime]
```

The MADC32 module with base address 0x60000000 named deadtime is used as a scaler and is the only scaler in the scaler stack. The scaler contributes two value to the scaler data, these will have the names deadtime and totaltime.

timeCalibration

This variable holds the calibration which multiplies a raw timestamp value to get milliseconds. The sample daqconfig.tcl distributed when a new data acquisition account is created automatically computes that in terms of the actualTimeDivisor. That in turn at present is computed from the value programmed into the MADC32 via the variable madcTimeDivisor. This computation will need to be revised when the MADC32’s scaledown becomes a simple value rather than a divisor shift count.
Chapter 4. SpecTcl

SpecTcl is the histogramming component of the NSCL data acquisition system. It uses an application called Xamine to display spectra that have been created. The Getting Started chapter provides an overview that describes how to use SpecTcl. This chapter provides a bit more background on the program and describes SpecTcl and its interaction with the configuration file.

SpecTcl is actually a library not a program. To make it work with a specific data set, the experimenter, in general must supply software that takes as input raw events, and produces as output parameters from which SpecTcl can increment the appropriate histograms.

For general applications, this user supplied software is organized as a logical pipeline of Event Processors. Event processors are run sequentially, each event processor has access to the raw event and to the parameters that have been computed by event processors that executed prior to it. Normally the first set of event processors operate on the raw event to produce raw parameters. Once this has been done, additional event processors can operate on the raw parameters to produce additional parameters without needing to know the form of the raw event.

An example of an event processor that might operate on decoded parameters would be a calibrator. A calibrator would take a set of raw parameters, and apply a calibration function to each of them to produce calibrated parameters (e.g. taking raw ADC values and producing energies).

The SpecTcl provided with this software includes a first stage event processor that knows how to unpack the raw events from the VM-USB for any set of stacks that can be defined by the daqconfig.tcl file we have described into a set of SpecTcl parameters.

Several plugins are also supplied with SpecTcl. A SpecTcl plugin is a shared library that can be dynamically incorporated into SpecTcl at run-time. Plugins provide additional packages of functionality that may not be required of all users. Plugins can provide additional command, event processors, or any other arbitrary mix of C/C++ and Tcl/Tk code.

This chapter describes how SpecTcl’s initialization scripts use the daqconfig.tcl definitions and variables to create parameter definitions and an initial set of spectra. It also describes and documents the plugins that have been distributed with this version of SpecTcl.

4.1. Using daqconfig.tcl to drive event decoding

SpecTcl needs to know only two thing for each module:

- The geographical address of the module
- The names to be given to parameters from each channel of a module.
The assumption is that each module is a 32 channel module. If you are using a 16 channel module (e.g. a 785N), the module will deliver the even numbered channels (0,2,..30). You must therefore specify the names of parameter for the even numbered channels.

The reference section of the Configuration Files chapter describes the adcChannels array. Recall that Tcl arrays are indexed by strings not numbers. Each element the adcChannels array should be a Tcl list that describes the names of the 32 channels of a module. The index of each element is the name of an adc module created with the adc create command in the daqconfig.tcl file.

The list command is a Tcl command that will build lists with proper quoting and bracketing. Surround each list element that has spaces or { } brackets with quotes. Surround each list element that has [] brackets or $'s with {}'s. For example:

**Example 4-1. Using the Tcl list command**

```tcl
set adcChannels(someAdc) [list simple "has spaces or {}" {has [tcl special] $characters}]
```

The config directory that is installed when you create a new account for data taking includes two additional scripts: configFile.tcl and spectclSetup.tcl. SpecTcl can be told to use the initialization scripts to process daqconfig.tcl when it starts.

**configFile.tcl** contains Tcl procs that mimic the command extensions of the Readout program. Once configFile.tcl has been sourced in, one can also source daqconfig.tcl. This results in the following global variables:

- `adcConfiguration`
  - Many types of digitizers encode a module identifier in their data stream. This provides a simple method to double check the data being decoded is what the decoder expects to find. The module identifier can also serve as a lookup index to locate the set of parameters provided by a module.
  - `adcConfiguration` is a Tcl array that is indexed by module name. Each element of the array is the module identifier or -1 if the module is not capable of providing an identifier.

- `readoutDeviceType`
  - The version of SpecTcl distributed to STUK, can actually decode data from events that consist of data from an arbitrary mix of three digitizer module types (CAEN Vxxx 32 channel digitizers, HYTEC NADC2530, and Mesytec MADC32). The `readoutDeviceType` variable is a Tcl array indexed by module name. Each element of the array contains a code that identifies the module type. This is used by SpecTcl to select module unpackers appropriate to each segment of the event.
  - Module types are currently: 0 for CAEN Vxxx modules, 1 for Hytec NADC2530 modules and 2 for Mesytec MADC32 modules. One acquainted with the history of VM-USB SpecTcl, would note that these values represent the order in which module type support were added to the software.
stackOrder

The VM-USB organizes its readout in stacks. The configuration file software supports stacks by supplying a pseudo module type of stack. Stack modules are created just like any other module. Stack modules, however, implement a -modules option. The value of that option is a list of real modules whose readout code is composes the stack loaded into the VM-USB.

stackOrder is a global Tcl array. The array is indexed by the names of the defined stacks. Each element of the array consists the value of that stack's -modules list.

stackNumber

VM-USB stacks have a numeric id. When a stack executes, the data it produces includes a header that contains the id of the stack. This allows the decoding software to know which set of decoders to run for a given event.

The configuration file allows you to name stacks. The configuration software assigns a stack id to each stack you define. The stackNumber variable is a Tcl array indexed by stack name. Its elements are the stack number assigned to that stack.

The spectclSetup.tcl script glues all of this together it:

1. Sources the configFile.tcl script.
2. Sources the daqconfig.tcl script.
3. Processes the values of the arrays defined by daqconfig.tcl and configFile.tcl to produce SpecTcl parameter names for each parameter defined.
4. Uses the paramMap command to describe the mapping between Adc geographical address/channel number pairs and those parameter for the event decoding event processor.
5. Creates 1-d raw spectra for each of the parameters.

Let's conclude this section with an example of some code in the SpecTclRc.tcl initialization file that makes use of all this:

Example 4-2. Incorporating automatic parameter/spectrum generation into SpecTcl

```tcl
set setupFilename [file join ~ config spectclSetup.tcl]
source $setupFilename
```
4.2. SpecTcl Plugins.

SpecTcl supports plugins via the Tcl load command. A SpecTcl plugin is a shared library and, potentially, supporting Tcl/Tk scripts that provide additional functionality to SpecTcl that is either not universally useful, or depends on optional external software.

Plugins can add commands to SpecTcl, create event processors, provide additional graphical user interfaces and much much more. In this section we will examine some of the plugins that have been included with the STUK system and their functionality. Specifically, we will look at:

1. The firstof plugin, which can be used to create strip position and hit spectra.
2. The calibparam plugin, which can create new parameters by determining and applying linear calibrations to existing parameters.
3. The constparam plugin, which can create a parameter with a constant value if either all of a list of dependent parameters is present, or if any of a list of dependent parameters is present. See the separate reference document for this plugin.

4.2.1. The firstof Plugin

This plugin provides SpecTcl commands that generate and adds additional event processors to the SpecTcl event processing pipeline.

The new event processors locate the first, or largest parameter in a list of parameters that has been assigned a value and produces two output parameters. The first output parameter is the value of the first, or largest defined parameter. The second output parameter is the index of the found parameter in the list.

One sample use of this plugin would be to create hit/position spectra from Si strip detectors.

4.2.1.1. Loading the plugin

The Tcl load command loads plugin libraries. You must specify the path to the plugin library completely. If the plugin has been installed in the SpecTcl installation, you can use the SpecTclHome variable to shorten the path.

The sample below shows how to load the firstof plugin if it was installed in the NSCL SpecTcl directory tree:

Example 4-3. Loading the plugin

```tcl
load $SpecTclHome/TclLibs/libfirstof.so
```
4.2.1.2. The firstof command

Loading the firstof filter adds a new command to SpecTcl: firstof. This command is used to define two new parameters for a list of input parameters. The command installs an event processor at the end of the SpecTcl analysis pipeline to compute these parameters.

By convention, the new parameters are referred to as the value and firsthit parameters. The event processor iterates through the list of the input parameters in the order in which they are specified on the command line. When it locates the first parameter that has been assigned a value, it sets the value output parameter to the value of that parameter. It sets the firsthit parameter to the index into the list of the first defined parameter.

The syntax of the firstof command is:

Example 4-4. Syntax of firstof

\[
\text{firstof } \text{value} \ \text{firsthit} \ \text{input-list}
\]

value
Is the name of the value output parameter. This parameter must not yet be defined in SpecTcl. The command errors out if it is.

firsthit
Is the name of the first hit output parameter. This parameter must not yet be defined in SpecTcl. The command errors out if it is.

input-list
Is a properly formatted list of parameters that are already defined in SpecTcl. These will be the list of input parameters from which the output parameters will be computed.

4.2.1.2.1. The biggestof command

The biggestof command is also added, it has the same syntax as the firstof command but generates an event processor that returns the largest value and index of the largest value parameter in the list of parameter monitored.

An example of this that is identical to that of the firstof example is:

4.2.1.2.2. The firstof plugin and STUK SpecTcl

The firstof plugin is loaded by the STUK SpecTclRC.tcl SpecTcl startup script. In addition, a proc called biggest is defined. biggest is tuned for building the parameters needed to create X/Y position and Energy correlation spectra from double sided silicon strip detectors. It assumes that the front and
Chapter 4. SpecTcl

back strips have names like `somebase.nn` where `somebase` is a common base name and `nn` is a two digit strip number.

In the STUK sample configuration scripts, for example, the dsd x strips are named `dsssd1.x.00` through `dsssd1.x.31`. The basename is `dsssd1.x` while the strip numbers are the `nn`s.

`biggest` executes a `biggestof` command on the strips defined by a basename and produces two new parameters. `basename.e` is the value of the largest parameter (e.g. `dsssd1.x.e` is the largest energy of the hit strips). `basename.hit` is the strip number that has the largest energy.

Experiment specific scripts can invoke this command and subsequently create the 2-d strip hit position spectra and the energy correlation spectrum.

4.2.2. The calibrated parameter plugin

In many applications, a simple linear transform relates raw ADC values to some more meaningful value. For ADC’s this value is often the energy left in a detector by a detected particle or photon. For TDC’s this value is usually a time relative to some other ‘happening’ in the event.

The Calibrated parameter plugin provides a means to compute and apply these linear transforms to any raw parameter value to produce a value with ‘real world’ units, or a calibrated parameter. The plugin supports creating the calibration function as well as providing, and registering an event processor to apply calibration functions to raw parameter to produce calibrated parameters.

The plugin adds two commands to SpecTcl. `calibrationfit` accumulates takes a set of raw parameter/actual parameter pairs, and produces a linear least squares fit that can be used as the calibration function for calibrated parameters. The `calibparam` command associates a raw parameter, with a fit to direct the calibration event processor to produce a new calibrated parameter.

4.2.2.1. Loading and using the plugin

The calibrated parameter plugin is a Tcl loadable package which is a compiled extension.

If you install the plugin in SpecTcl’s installation directory, you will not need to do anything to prepare to load the plugin. If you installed the plugin anywhere else, you will need to add the directory the plugin is installed in to the SpecTcl package load path.

You can do this by using `lappend` to append that directory to `auto_path`. For example:
Example 4-5. Adding the plugin directory to the package load path

    lappend auto_path [file join ~ calibratedparameters]

Once the plugin can be found it can be loaded via the Tcl `package require` command:

Example 4-6. Loading the plugin

    package require Calibrations

4.2.2.1.1. Using the plugin

The sequence for using the plugin, once it is loaded is

1. Create a calibration fit that will provide a calibration function.
2. Add points to the calibration fit. Each point consists of a raw parameter value and the corresponding desired calibrated value. Often fit points are gotten by identifying features in spectra of the raw parameter (e.g. known energy peaks in gamma spectra).
3. Ask the calibration fit to perform itself. Performing a calibration fit does a linear least squares fit to the fit points computing the slope and offset of the best fit line through the points you provided.
4. Create a calibrated parameter by applying your calibration fit to a raw parameter to produce a new parameter. Once this calibrated parameter has been created it looks to SpecTcl like any other parameter; Spectra can be created, pseudo parameters can be scripted using it and so on (creating a calibrated sum parameter from several calibrated parameters e.g.).

The Example below shows this process for calibrating a parameter named `adc1.00` to produce a parameter named `adc1.e.00`. Don’t take the actual calibration points too seriously.

Example 4-7. Calibrating a parameter

    calibrationfit linear testfit
    calibrationfit -add testfit {0.0 0.0} {1.0 2.0} {100.0 202.0}
    calibrationfit -perform testfit
    calibparam -create adc1.e.00 1234 adc1.00 testfit KeV

The first three lines of the example above create a fit named `testfit` add three points to the fit that describe the relationship between channel values and energies. The relationship is a linear with intercept very nearly 0.0 and slope very nearly 2.0. The `calibparam` command creates the new calibrated `adc1.e.00` parameter as parameter number 1234 computing it by applying the fit `testfit` to the raw parameter `adc1.00` for each event. The `KeV` indicates that the units for this new parameter will be `KeV`. 
The fit is a snapshot of the fit. If you later delete the fit, the calibrated parameter will continue to function on the original fit data. If you later re-create the fit with a different set of values, the parameter will continue to compute on the old fit. To change the fit you would need to delete and recreate the calibrated parameter. This is a compromise that is mandated by performance considerations (it would be expensive to have to locate the correct fit for each event, rather than storing a copy of the fit itself).

### 4.2.2.1.2. Calibrated parameter plugin command reference

**calibrationfit**

**Name**

`calibrationfit` — Manipulate fits used to compute calibrated parameters

**Synopsis**

```
calibrationfit ?-create? type name
```

```
calibrationfit -list ?pattern?
```

```
calibrationfit -delete name
```

```
calibrationfit -perform name
```

```
calibrationfit -add name point1 ?point2...?
```

```
calibrationfit -evaluate name xValue
```

**DESCRIPTION**

The `calibrationfit` command manipulates the set of fits that are used to compute calibrated parameters. The fitting software is extensible to any sort of calibration fit that can be computed. At present, however only linear fits are supported.

The general form of the `calibrationfit` command is the command, followed by an option that describes the action the command will take, followed by the parameters that action needs.

The options, their actions and the parameters they need are documented in the OPTIONS section below.
Fits have state associated with them. The state determines what actions you are allowed to perform. The states are: accepting and performed.

OPTIONS

?-create? type name

Creates new fit. The option -create is optional. if not provided, this is the default action. The fit subsystem is extensible to support any type of fit. Fit types have names. The type parameter is the fit type. At present this can only be linear. The name parameter is the name to assign to this fit. The fit name must be unique amongst all fits and will be used to refer to this fit in future calibrationfit and calibparam commands.

The initial state of a fit is accepting indicating that it is capable of accepting fit data points.

-list ?pattern?

Lists the known fits. If pattern is supplied, it is a pattern with glob wild cards that restricts the listing to those fits whose names match the pattern. glob pattern wild-cards are essentially those supported by filesystem matching wild cards in the Linux command shells.

The command returns a (possibly empty) list of fit descriptions. Each fit description is itself a sublist containing in order the fit-name, fit-type, fit-state, A list of the points used to compute the fit; each point is itself a two element raw/real coordinate pair, if the fit state is accepting, the fit description contains a final empty list element. If the state is performed, the final list element contains a list that is a set of name, value pairs that define the computed fit parameters and the Chisquare. The name for the linear fit are: slope, offset, and chisquare.

-delete name

Deletes the fit named name. Note that calibrated parameters are created with a copy of the fit. To stop a calibrated parameter from computing, you will need to delete it.

-perform name

Performs the fit. If there are sufficient points, the fit parameters are computed, and the state of the fit becomes performed. The command returns the list of fit parameter name/value pairs. For the linear fit type these are: slope, offset, chisquare.

-add name point ?...?

Adds calibration points to a fit that is in the accepting state. Any number of points can be added to the fit. At present, there is no way to edit the set of points, other than to delete and re-create the fit. Each point is a Tcl list of two elements, raw value and corresponding calibrated value.
-evaluate name raw-value

This action is only available for fits that are in the performed state. The raw-value is passed to the fitted function computed for name. The resulting calibrated value is returned as the command’s value.

**calibparam**

**Name**

*calibparam* — Create, list and manipulate calibrated parameters.

**Synopsis**

*calibparam* ?-create? name number raw fitname ?units?

calibparam -list ?pattern?

calibparam -delete name

calibparam -refresh ?pattern?

**DESCRIPTION**

The *calibparam* command creates and manipulates calibrated parameters. A calibrated parameter is a computed parameter that depends on a base, or raw parameter and a calibration function or calibrationfit. Calibrated parameters are implemented as event processors and run at compiled speeds.

The base parameter for a calibrated parameter can be any parameter known to SpecTcl at the time the calibrated parameter is created with the exception of scripted pseudo parameters, which are always last in SpecTcl’s evaluation chain in this implementation of SpecTcl.

The first word of the *calibparam* command following the command keyword is an option switch that defines the action the command will be performing. These options and their actions are described in OPTIONS below.

101
OPTIONS

-create name number raw fitname ?units?  
Creates a new calibrated parameter. The -create option is not required. Creation is the default action if no initial option is present. name is the name of the new calibrated parameter and number is the SpecTcl parameter number associated with that name. number must be unique as must name.

raw is the name of the raw parameter to which the fit named fitname will be applied to compute values of the name calibrated parameter. The optional units parameter allows you to associate units of measure with the new parameter name. These units of measure will be displayed on axes of spectra that depend on this parameter.

-list ?pattern?
Lists the calibrated parameters that are now active. If the optional ?pattern? is present, only parameters whose names match the pattern will be listed. The pattern can contain all the wildcards recognized by the unix shells when expanding/finding filenames. If pattern is missing it defaults to * which lists all parameters.

The list is returned as the command result and therefore can be manipulated by other commands. The list is a well formed, possibly empty Tcl list. Each element of the list is a sublist that describes one parameter. The sublists have five elements, the name of the parameter, the number of the parameter, the name of the raw parameter, the name of the fit used to compute the parameter, and the units or an empty string if no units were defined.

-delete name
Deletes the calibrated parameter name

-refresh ?pattern?
Refreshes the calibrated parameters whose names match the pattern. pattern can contain any of the unix file matching wild-cards. If omitted, the pattern defaults to *, refreshing all parameters.

Refreshing a parameter updates any changes to the fit that computes the parameter.
Chapter 5. Scaler Display

The VM-USB scaler stack is triggered periodically. Data it reads are put in scaler buffers. These buffers have an array of scaler data that can be displayed by the NSCL scaler display program. The scaler display program is configured by a Tcl script configuration file.

Scaler display configuration files normally have two segments. The first segment provides names for each of the scalers read out, and a correspondence between each name and its index in the array of scalers in the data buffer. The second segment uses those names to create scaler display pages and, optionally stripcharts of scaler data.

The first part of the configuration can be done automatically by scripts that decode the daconfig.tcl configuration file. The ~/config/scalerChannels.tcl script creates the channel definitions from a list of scaler modules you provide and an array of channel names for each scaler module.

When you write the daqconfig.tcl configuration script, you will need to set the variable scalerChannels to the list of scaler channel names that are read, in the order in which they are read. Scaler modules are read in the order in which they are specified in the scaler stack. Scaler channels are read in order from each module. When an MADC32 is used as a scaler its deadtime register is read first, followed by its elapsed time register.

You will have to write a scaler configuration file that

1. Sources the scalerChannels.tcl file
2. Defines scaler pages, their contents.
3. Optionally defines strip chart parameters.

In order for the application startup files to locate your configuration file, create it by modifying the skeleton placed in ~/config/scalerconfig.tcl.

A completed configuration file might look like this:

Example 5-1. A complete scaler display configuration file

source [file join ~ config scalerChannels.tcl]

# Make the raw counts page, put the channels scaler1.00 - scaler1.03 on it
# as singles:

page Raw "Raw counts page example"
display_single Raw scaler1.00
display_single Raw scaler1.01
display_single Raw scaler1.02
display_single Raw scaler1.03
Chapter 5. Scaler Display

#
# Make a ratios page which will contain the ratios of
# scalern/scalern+1 where n = 4 -> 8

page Ratios "Sample page with ratios."
display_ratio Ratios scaler1.04 scaler1.05
display_ratio Ratios scaler1.06 scaler1.07
display_ratio Ratios scaler1.08 scaler1.09

# Enable the strip chart and have it display:
# scaler1.10 and the ratio of scaler1.11 to scaler1.12:
#
stripparam scaler1.10
stripratio scaler1.11 scaler1.12
Chapter 6. Converting event data to XML files

This chapter describes the evttoxml application. evttoxml translates event files and ancillary data to XML files containing various metadata tags as well as the event data.

This chapter is divided into the following sections:

**Background**

Provides background that describes how the NSCL DAQ organizes data associated with a run, and what data the evttoxml application expects.

**Usage**

Describes how to use the evttoxml application.

**Output description**

Describes the tag that appear in the output file, their attributes and bodies.

### 6.1. Background

In order for evttoxml to produce proper metadata sections it must have access to the daqconfig.tcl configuration file, and any additional metadata file as they were when the run was taken. The NSCL DAQ system supports this by a scheme that associates arbitrary ancillary data with the event file.

NSCL online documentation (http://docs.nscl.msu.edu/daq/bluebook/html/x3386.html) describes the directory structure maintained by the NSCL DAQ software. For this discussion, it is sufficient to know that the directory tree ~/experiment/current is copied and associated with event data for a run at the time the run is ended. If the directory structure contains symbolic links, the contents of the links are copied, creating a snapshot of any linked files and directories.

The event file segments, and ancillary data in the ~/experiment/current directory tree are bundled together in the directory ~/experiment/runxxx, where xxx is the number of the run. That directory will contain a snapshot of the ancillary data and the event file segments.

When a new data taking account is created, scripts that extend the account creation process create a symbolic link ~/experiment/current/config that points to the ~/config directory. As described above, this will associate each run a snapshot of the contents of the ~/config directory.

The evttoxml application uses the daqconfig.tcl, metadata.tcl along with a library script that interprets these to:

- Create metadata sections of the XML output file
• Drive the decode of the event data and subsequent translation into XML in a manner analogous to what SpecTcl does
• Associate names with scaler channels, which then are used in the tags for scaler records, in a manner analogous to the way in which the Scaler Display program works.

On important feature of evttoxml is the capability to incorporate arbitrary user metadata. This should be placed in the script ~/config/metadata.tcl. Metadata is created by using the metadata command.

Here is reference information for that command:

metadata

**Name**
metadata — Support arbitrary metadata

**Synopsis**

```tcl
metadata tagname ?options?
```

**DESCRIPTION**

Creates a piece of user specific metadata that will appear in the <userMetaData> section of the output XML files created by evttoxml.

The `tagname` parameter will be the name of the tag generated.

**OPTIONS**

```tcl
-attribute name value
```

Adds an attribute to the tag. The next command line parameter is a two element Tcl list. The first element is the attribute name. The second its value.

Any number of `-attribute` options can be supplied in the command. All are inserted into the opening tag.
-body body-fragment

Provides a chunk of #CDATA, body-fragment that will appear in the tag body. Any number of -body options can appear. The body-fragments supplied are appended in order to form the body of the tag.

EXAMPLES

Example 6-1. An empty tag

metadata empty

Produces <empty />.

Example 6-2. Tag that only has attributes

metadata experimenter -attribute {name "Ron Fox"} -attribute {institution NSCL} -attribute {email "fox@nscl.msu.edu"}

Produces <experimenter name="Ron Fox" institution="NSCL" email="fox@nscl.msu.edu" />.

Example 6-3. Attribute and body

metadata experimenter -attribute {run 1234} -body {Ron Fox}

Produces <experimenter run="1234">Ron Fox</experimenter>.

6.2. Usage

Example 6-4. Using evttoxml

/usr/opt/evttoxml/evttoxml run

run is the number of the run to translate. The output is sent to stdout. You can either redirect that to a file or pipe it into the input of a database load program.

Shell scripts, tcl scripts or Tk Gui’s or other programs can be used to drive mass translation of several runs.
6.3. Output description

This section describes the output of evttoxml. We give the file contents in a tag by tag description that also describes the order of the file. Note that at the time this is being written, there is no known DTD, therefore, the dtd file path will need to be adjusted in the program itself.

The XML output opens with a header that identifies the file as an XML UTF-8 encoded file and describes the document type, providing a path to the document type description file:

Example 6-5. XML Header

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE events PUBLIC "-//STUK//EVENTS" "file://insertpathtodtdhere.dtd"/>
```

Note that at the time this is being written, there is no known DTD, therefore, the dtd file path will need to be adjusted in the program itself.

The <eml> tag is the root tag of the document. It includes a single mandatory attribute directory whose value is the directory that contains the event file and metadata. The body of this tag, by definition, the remainder of the tags described in this section:

Example 6-6. <eml>

```xml
<eml directory="/home/fox/experiment/run234">
```

The example above would be what would appear for the user fox translating run 234.

The <metadata> tag encloses the entire metadata section of the document. It has no attributes. It’s body is all of the metadata, which consists of three sections:

1. <detectors> which provides metadata about the detectors used in the run.
2. <electronics> which provides metadata about the electronics settings under which the run was taken.
3. <userMetaData> which provides for a section of arbitrary metadata taken from the ~/config/metadata.tcl script.

The <detectors> tag introduces the metadata that describes the detectors used in the run. The tag has no defined attributes. The contents of the tag are detector descriptions.

The <detector> tag contains several detector descriptions. Each detector description begins with a <detector> tag. This tag has the following attributes:

```
type
```

The value of this attributes is the type of detector (e.g. hpge or dsssd
Chapter 6. Converting event data to XML files

code

The value of this attribute is the inventory code for the detector

The body of the <detector> tag is a sequence of <parameter> tags that name the parameter this detector supplies to the data stream. This can be joined to similar electronics data to determine which digitizers service which detectors.

<parameter>. When used in the body of the <detector> tag, this provides a name for a parameter produced by the detector. This tag has no attributes. Its body is the name of a parameter.

Example 6-7. Sample detector metadata

  <detector type="hpge" code="HPGE-3205">
    <parameter>hpge.e</parameter>
  </detector>

<electronics>. This tag encloses the section that describes the electronics and its settings. Two aspects of the electronics are described. For each electronics module, the names of the parameters unpacked from that module appear in <channel> tags. For each parameter that differs from the default value, a <property> tag is generated that describes that property.

<module>. The <module> tag introduces the description of a single VME electronics module. This tag has the following attributes:

type

The value of this tag is the electronics model type (e.g. mesytec-madc32).

name

Each module is given a name in the daqconfig.tcl configuration script. This name is the value of the name attribute

Example 6-8. Sample module tag for the HPGe adc

  <module type="mesytec-madc32" name="hpge”>

<channel>. Each module may have one or more channels. The <channel> tag describes the SpecTcl parameters that are unpacked from a module.

Example 6-9. Sample channel tag from the hpge module

  <channel input="0">hpge.e</channel>

<property> tags. The potential set of properties each electronics module may have depends strongly on the type of the module. While the STUK system as delivered only uses Mesytec MADC 32 modules, other modules are possible. In order to be able to represent the settings of each module type without
expanding the set of tags as new modules are added to the system, module settings are described using property lists. A property list is a sequence of \texttt{<property>} tags. Each tag has a \texttt{name} attribute that describes the name of the property (in general the configuration option with the - removed). The body of the tag is the value of that property.

**Example 6-10. Base address property**

\begin{verbatim}
<property name="base">0x60000000</property>
\end{verbatim}

\texttt{<userMetadata>}. The user metadata follows the electronics section in the XML file. This section is the body of the \texttt{<userMetaData>} tag. The body consists of the metadata tags, attributes and bodies defined by the \texttt{metadata} commands in the \texttt{metadata.tcl} file.

This concludes the description of the metadata tags. Immediately following the metadata tags, is the list mode event data translated to XML. The list mode data is the body of a \texttt{<listdata>} tag, and can consist of the following tag sets:

- \texttt{<beginrun>}. The begin run tag introduces a begin run record. It contains the \texttt{time} attribute, which provides an absolute time stamp at which the run began, and the \texttt{run} attribute which provides the run number.
- \texttt{<title>}. The title tag is contained in the body of the begin run tag. Its body is the title of the run.

**Example 6-11. Sample begin run record in XML**

\begin{verbatim}
<beginrun time="20080511T051154" run="234">
  <title>
    This is a test title for XML conversion
  </title>
</beginrun>
\end{verbatim}

- \texttt{<event>}. The system responds to triggers by reading an event. Each event is enclosed in an event tag. The event tag contains \texttt{<parameter>} tags that describe the parameters ready by the event.
- \texttt{<parameter>}. Parameter tags describe the data in an event. The name attribute is a parameter name that appeared in a channel tag for the electronics. The body of the parameter tag is the value of that parameter for this event.

**Example 6-12. Sample event with only HPGe data**

\begin{verbatim}
<event>
  <parameter name="hpge.e">572</parameter>
</event>
\end{verbatim}

- \texttt{<scalers>}. Periodically a scaler event is emitted. The scaler event is the body of a scalers tag. The scaler tag has the attribute \texttt{timeoffset} which is the number of seconds into the run at which the scaler record was emitted.
< scaler>. Each scaler value is provided as a scaler tag. The tag has the name attributes which provides the channel name of the scaler. The body is the value of the scaler.

**Example 6-13. A scaler record**

```
<scalers timeoffset="5">
  <scaler name="deadtime">16</scaler>
  <scaler name="totaltime">5000000</scaler>
</scalers>
```

< endrun.> The end of the run is signalled with an endrun tag. This provides the time attribute, whose value is the time at which the run ended, and the timeoffset which provides the number of seconds the run was active.

**Example 6-14. End run xml records**

```
<endrun time="20080511T051202" timeoffset="132">
```

### 6.4. Fragments from an an XML event file

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE events PUBLIC "//STUK//EVENTS"
"file://insertpathtodtdhere.dtd">
<eml directory="/user/fox/stagearea/experiment/run234">
<meta>
<detectors>
<detector type="hpge" code="HPGE-3205">
<parameter>hpge.e</parameter>
</detector>
...
</detectors>
<electronics>
<module type="mesytec-madc32" name="dsssd1.x">
<channel input="0">dsssd1.x.00</channel>
<channel input="1">dsssd1.x.01</channel>
<channel input="2">dsssd1.x.02</channel>
<channel input="3">dsssd1.x.03</channel>
<channel input="4">dsssd1.x.04</channel>
<channel input="5">dsssd1.x.05</channel>
<channel input="6">dsssd1.x.06</channel>
<channel input="7">dsssd1.x.07</channel>
<channel input="8">dsssd1.x.08</channel>
<channel input="9">dsssd1.x.09</channel>
<channel input="10">dsssd1.x.10</channel>
...
<channel input="31">dsssd1.x.31</channel>
<channel input="32">timestamp</channel>
<property name="base">0x4000000</property>
<property name="gategenerator">disabled</property>
```
Chapter 6. Converting event data to XML files

<property name="gatemode">common</property>
<property name="id">4</property>
<property name="inputrange">8v</property>
<property name="ipl">0</property>
<property name="thresholds" index="0">0</property>
<property name="thresholds" index="1">0</property>
<property name="thresholds" index="2">0</property>
<property name="thresholds" index="3">0</property>
<property name="thresholds" index="4">0</property>
<property name="thresholds" index="5">0</property>
<property name="thresholds" index="6">0</property>
...
<property name="timingdivisor">14</property>
<property name="timingsource">vme</property>
</module>
...
<userMetaData>
<empty />
<attonly name="Ron Fox" />
<hasbody name="Ron Fox" >
This is the tag body
</hasbody>
</userMetaData>
</metadata>
</listdata>
<beginrun time="20080511T051154" run="234"><title>This is a test title for XML conversion</title>
</beginrun>
<event><parameter name="dsssd1.y.00">23</parameter>
<parameter name="dsssd1.y.01">30</parameter>
<parameter name="dsssd1.y.02">35</parameter>
<parameter name="dsssd1.y.03">24</parameter>
<parameter name="dsssd1.y.04">39</parameter>
<parameter name="dsssd1.y.05">29</parameter>
<parameter name="dsssd1.y.06">33</parameter>
<parameter name="dsssd1.y.07">31</parameter>
<parameter name="dsssd1.y.08">30</parameter>
<parameter name="dsssd1.y.09">30</parameter>
<parameter name="dsssd1.y.10">28</parameter>
<parameter name="dsssd1.y.11">23</parameter>
<parameter name="dsssd1.y.12">37</parameter>
<parameter name="dsssd1.y.13">56</parameter>
<parameter name="dsssd1.y.14">23</parameter>
<parameter name="dsssd1.y.15">35</parameter>
<parameter name="dsssd1.y.16">38</parameter>
<parameter name="dsssd1.y.17">34</parameter>
...
<parameter name="dsssd1.y.31">41</parameter>
<parameter name="timestamp">0</parameter>
</event>
...
<scalers timeoffset="5"><scaler name="deadtime">0</scaler>
<scaler name="totaltime">0</scaler>

112
</scalers>
...
<endrun time="20080511T051202" timeoffset="8" />
</listdata>
</ecl>
Chapter 7. Slow control server

The VMUSB readout skeleton provides support for slow control devices that are installed in the VME crate controlled by the VM-USB. This is done via a server component that accepts connections and performs VME operations on behalf of the client on the other end of the connection. This chapter:

- Describes the application protocol that clients use to control devices.
- Describes the control configuration file `~/config/daqconfig.tcl` that installs slow control devices in the system making them available to clients.
- Describes the clients available for the supported slow control devices.

### 7.1. Application protocol

An application protocol is a description of the messages sent from client to server and the replies that are sent back to satisfy the requests implicit in a client message.

The protocol assumes that there are several instances of device drivers installed in the server and that each instance has a unique name. Each device driver, in turn may have several named parameters. Messages sent to the server are dispatched to the appropriate driver for processing (based on the device name), and the driver returns a reply that is relayed to the client.

The messages exchanged between the client and server are purely textual and must end in a newline character (`C \n`). Each client request looks like a Tcl command. Replies for successful requests begin with the text string `OK` and may contain additional text that is request and driver dependent. Replies for unsuccessful requests begin with the text string `ERROR` followed by additional text that describes the error. This text depends on the error and who caught it.

The following requests are currently supported:

#### Set request

**Name**

Set — Requests a parameter be set to a value.

**Synopsis**

Set `device-name parameter-name value`
DESCRIPTION

The **Set** request attempts to set a device parameter. `device-name` is the name of a device defined in the configuration file. `parameter-name` identifies what is to be set and is device driver specific. `value` is new value of the parameter.

Get request

**Name**

Get request — Retrieves the value of a parameter.

**Synopsis**

Get `device-name parameter-name`

**DESCRIPTION**

Retrieves the value of a parameter from a device. `device-name` is the name of the device in the configuration file from which data will be retrieved. `parameter-name` is the name of the parameter to retrieve.

Normally, on success, the result is the text `current-value`. This is an exception to the statement that successful replies must start with `OK` and is intended to make decoding the value by the client a bit simpler.

Update request

**Name**

Update request — Update from shadow registers.
Chapter 7. Slow control server

Synopsis

Update device-name

DESCRIPTION

Some devices have a write-only state. The Update is intended to refresh the device state from internal shadow state. Normally, device drivers have a mechanism to load initial state from some file this command can be used to ensure that initial state has actually propagated to the hardware.

device-name is the name of the device being refreshed.

7.2. Controls Configuration file

The slow controls configuration file describes the set of slow control device drivers that are activated. This is a Tcl script that is processed when the Readout program is started.

The slow control device drivers that are currently supported are:

jtecgdg
   The Jtec 8 channel Gate and delay logic unit.
caenv812
   The CAEN V812 CFD discriminator module.
caenv895
   The CAEN V895 leading edge discriminator.
vmusb
   Provides the ability to execute arbitrary VM-USB lists formatted by a client.

The control configuration file is named: ~/config/controlconfig.tcl You must have a file by this name even if it is empty. You may use the full set of tcl commands. In addition the Module is used to create, configure and otherwise manipulate instances of slow control device drivers.

The reference section below describes the Module command. Following that reference, I will describe the specific parameters that each device supports.
Module

Name
Module — Manipulate slow control drivers in controlconfig.tcl

Synopsis
Module create device-type device-name

Module config device-name ?option-value-pairs?

Module cget device-name

DESCRIPTION

The Module command is responsible for creating, configuring and querying the configuration of instances of a slow controls device driver. This command is a command ensemble who’s sub-commands determine exactly with action is performed.

The create subcommand creates an instance of a slow control driver. The device-type selects the specific type of device driver to be instantiated. The device-name provides a unique name that will be used to refer to this device both in future Module commands, and in interactions by remote device control client applications.

The config subcommand provides configuration options to a instance of a slow controls device driver. The device-name parameter is the name of a device you created with a previous Module create command. The option-value-pairs are pairs of command line parameter. The first parameter of each pair identifies a configuration optino, the second parameter it’s desired value. The specific option names depend completely on the specific device driver type and will be described in reference sections devoted to each device driver.

The cget subcommand returns configuration about a device driver instance. The device-name parameter is the name of a device driver instance you have already created via the Module create command.
OPTIONS

As indicated above, all options are specific to a device driver. A subsequent set of reference sections will describe the device driver options recognized by each device driver.

gdg - Jtec/Wiener Gate and delay generator

Name

GDG — Device driver for the Jtec/Wiener 8 channel gate and delay generator

Synopsis

The gdg device driver manages instances of the 8 channel Jtec/Wiener gate and delay generator/logic module.

DESCRIPTION

The Jtec/Wiener MDGG-8 gate and delay generator is a logic module that has many modes of operation. This device driver is currently only able to run this module as 8 channels of gate and delay.

DEVICE TYPE:

The GDG device driver is instantiated by using a device type of jtecgdg in the Module create command.

OPTIONS

-base

The value of this configuration parameter should be the base address set via the module’s a address jumpers.
CONTROL PARAMETERS

The MGDG-8 supports the following control parameters as targets for the Set and Get protocol operations.

delay\{n\}

Sets the delay for channel \{n\}. Use parameter names like delay0 to set the delay for channel number zero. \{n\} can be a value between 0 and 7 inclusive.

width\{n\}

Sets the width for channel \{n\}. Use parameter names like width0 to set the output width for channel number zero. \{n\} can be a value from 0 to 7 inclusive.

caeenv812/canev895

Name
caeenv812/canev895 — Unified device driver for the CAEN V812 CFD and CAEN V895 LED

Synopsis

This driver provides support for the CAEN V812 constant fraction discriminator and the CAEN V895 leading edge discriminator.

DESCRIPTION

This driver provides support for the CAEN V812 constant fraction discriminator and the CAEN V895 leading edge discriminator. The only difference between these modules are the set of parameters that are actually supported. See the CONTROL PARAMETERS section below where the additional parameters supported by the CAENV 812 are described.

This module is instantiated by specifying either the caenv812 or the caenv895 device type in the Module create command. Note that the actual device type is determined by querying the module registers rather than trusting the device type you provide. The actual device type you provide is for documentation purposes.
OPTIONS

-base
The value of the -base option is used to determine the VME address space occupied by the module. This must match the base address set in the module rotary switches.

-file
The value of the -file option is the full, absolute path to a file that contains the saved module configuration. This file is normally maintained by the graphical user interface for the discriminator.

For the most part the file is a Tcl script that contains a bunch of Tcl set commands that describe the settings for the module.

Detailed information about the format of this file can be found by typing man -M/usr/opt/daq/current/share/man caen812configfile at a shell prompt when logged into a data acquisition system.

CONTROL PARAMETERS

The CAEN V812 and CAEN V895 share a base set of parameters. The CAEN V812 requires a slightly larger set of parameters however. These are indicated in the descriptions below.

Each module is divided into two banks of 8 channels for a total of 16 channels. Some control parameters affect individual channels. The index of the channel affected is shown by the text (chan). The (chan) index can be replaced by a value from 0 through 15 inclusive.

Other control parameters affect banks of channels. The bank index is shown by the text (bank). (bank) can be replaced by 0 or 1. Bank zero controls channels 0–7 while bank 1 controls channels 8–15.

thresholds(chan)
Sets the value of the threshold for channel (chan). The value will be written or read without interpretation to a module threshold register.

widths(bank)
Sets the output width of a bank of outputs.

deadtimes(bank)
\textit{v812 only} The CAEN v812 manual calls this a deadtime, however this actually sets the delay parameter for the constant fraction discrimination for a bank of discriminator channels.
inhibits

Sets a mask of bits that determine which channels are inhibited. The least significant bit of the mask is channel 0 the most significant bit channel 15. Bits are set for channels that should be inhibited.

majority

Sets the majority level for the unit.

---

vmusb

**Name**

vmusb — Provide generic VME access.

**Synopsis**

This device driver provides generic access to the VME by accepting VMUSB lists of VME instructions, executing them immediately and returning the read data as a result.

The device driver name is `vmusb`

**DESCRIPTION**

This device driver provides generic access to the VME by accepting VMUSB lists of VME instructions and executing them immediately.

C programs can use this in conjunction with the `CVMUSBReadoutList` and `CVMUSBRemote` classes to do arbitrary accesses to the VME.

Note that if data taking is active, it is stopped for the duration of the execution of the list.

**OPTIONS**

This driver has no configuration options.
CONTROL PARAMETERS

This module has only one control parameter; `list` can only be set not gotten.

The data that can be ‘Set’ in the `list` control panel must be a well structured Tcl list. The first element of the list is the maximum size of the input buffer to use in accepting data read by the operation. The second element of the list is itself a well structured Tcl list. Each element of the list is a number that represents the value of an element of the VMUSB stack you want executed.

The `OK` return from setting this control parameter is followed in line by a space, a dash and a payload that is a well formatted Tcl list. Each element of that list is a number that represents a byte in the data read. The payload may be an empty list (the stack could consist only of write values) but it is always present. The number of elements in the payload list will never be more than the maximum input buffer size requested, however it may be smaller.

I strongly suggest that software using this driver be based around the `CVMUSBReadoutList` and `CVMUSBRemote` classes that will be described in the section describing Slow Control clients. These classes allow you to format lists of operations, marshall them and submit them for execution to the Tcl server in Readout, and then marshall the results.

v6533

Name
v6533 — CAEN V6533 HV module

Synopsis

Provides support for the CAEN V6533 HV module.

DESCRIPTION

This module type provides support for the CAEN v6533 HV module. It supports only one option: `-base` which supplies the module base address.
The following parameters are supported by this module. (Note that these parameters divide between settings and readings). Settings apply to a single specified channel while readings will fetch a Tcl formatted list of values representing the parameter value for all 6 channels of the controller.

\(v_n\)

Sets the requested voltage (setpoint) for channel \(n\)

\(i\)

Sets the current limit for channel \(n\)

\(on_n\)

Turns channel \(n\) on (if the value is a boolean true), else off if the value is a boolean false.

\(trip_n\)

Sets the trip time for channel \(n\). The trip time is the number of seconds the channel is allowed to be out of tolerance before the controller declares a trip and shuts the channel down.

\(svmax_n\)

Sets the maximum voltage for channel \(n\). If the set point is set above this value, this value will be used as the actual setpoint.

\(rdown_n\)

Sets the ramp down rate for channel \(n\)

\(rup_n\)

Sets the ramp up rate for channel \(n\)

\(pdownmode_n\)

Sets the power down mode of \(n\) to either \(off\) which shuts off power abruptly or to \(ramp\) which ramps the channel down at the rate set by its ramp down rate.

\(globalvmax\)

Returns the global maximum voltage. At power on this is the max voltage for all channels in the device.

\(globalmaxI\)

Returns the global current limit. If not set, this is the currently limit for the individual channels.

\(v\)

Returns all six requested voltages (set points).

\(i\)

Returns all 6 current limits.
on
  Returns the on/off status of all 6 channels.

vact
  Returns the actual voltage for all 6 channels.

iact
  Returns the actual current draw of all 6 channels.

status
  Returns the values of the 6 channel status registers.

ttrip
  Returns the trip times for all 6 channels.

svmax
  Returns the software maximum voltage for all 6 channels.

rdown
  Returns the ramp down rates for all 6 channels.

rup
  Returns the ramp up rates for all 6 channels.

pdownmode
  Get the power down modes for all 6 channels.

polarity
  Get the polarity of all 6 channels.

temp
  Get the temperature in all 6 channels. Note that the temperature is returned in degrees centigrade.

**MONITORED DATA**

This module uses the monitor feature of the slow controls thread. The monitor feature periodically executes a list of VME operations without interrupting online data taking. The resulting data are then routed to the Tcl server thread which provides it to each device driver for unpacking.

The data returned from the `mon` directed at a CV6533 device is a Tcl list that contains the following items:

- The global status register value
Chapter 7. Slow control server

- A Tcl list containing the 6 actual voltages from each channel.
- A Tcl list containing the 6 actual currents from each channel.
- A Tcl list containing the 6 temperatures from each of the channels.

7.3. Slow Control clients

This section describes slow control client software that is available for slow control drivers that are supported by the VM-USB readout software.

- A description is provided of the GUIs for the gdg and caenv812/caenv895 drivers.
- Reference material is provided for the CVMUSBReadoutList and CVMUSBRemote classes that allow you to interface with the vmusb device drivers.

7.3.1. Jtec MDGG-8 control panel

This section describes the GUI control panel for the Jtec MDGG-8 Gate and Delay generator logic module.

7.3.2. CAEN discriminator control panel

This section describes the GUI control panel for the CAEN discriminator modules caev812 and caev895

7.3.3. VMUSB remote access support classes

This section describes the C++ interfaces to the vmusb device driver. To use this driver you must create your own application. In the discussion that follows, we are going to assume that the VMUSB Readout software is installed in /usr/opt/daq/vmusbReadout If your installation differs, you may need to adjust some of the paths described below.

The headers described in this section are locateded in /usr/opt/vmusbreadout/include. Your compilations will therefore need to specify the flag -I/usr/opt/vmusbReadout/include

The remote access library itself is libVMUSBRemote.so The link stage of your appilcation build must therefore specify the flags: -L/usr/opt/vmusbReadout/lib -lVMUSBRemote
Chapter 7. Slow control server

-Wl,"-rpath=/usr/opt/vmusbReadout/lib" The last flag on that line ensures that the dynamic loader will search the /usr/opt/vmusbReadout/lib for shared libraries at run time.

CMVUSBReadoutList

Name

CVMUSBReadoutList — Generate lists of VME operations

Synopsis

#include <CVMUSBReadoutlist.h>

class CVMUSBReadoutList {
    CVMUSBReadoutList();
    CVMUSBReadoutList(std::vector<uint32_t>& list, const CVMUSBReadoutList& rhs);
    void clear();
    void clear();
    const size_t size();
    const std::vector<uint32_t>& get();
    void addRegisterRead(unsigned int address);
    void addRegisterWrite(unsigned int address, uint32_t data);
    void addWrite32(uint32_t address, uint8_t amod, uint32_t datum);
    void addWrite16(uint32_t address, uint8_t amod, uint16_t datum);
    void addWrite8(uint32_t address, uint8_t amod, uint8_t datum);
    void addBlockRead32(uint32_t baseAddress, uint8_t amod, size_t transfers);
    void addFifoRead32(uint32_t baseAddress, uint8_t amod, size_t transfers);
    void addBlockWrite32(uint32_t baseAddress, uint8_t amod, void* data, size_t transfers);
    void addBlockCountRead8(uint32_t address, uint32_t mask, uint8_t amod);
    void addBlockCountRead16(uint32_t address, uint32_t mask, uint8_t amod);
    void addBlockCountRead16(uint32_t address, uint32_t mask, uint8_t amod);
}

DESCRIPTION

METHODS

126
7.3.4. The CAEN V6533 high voltage control panel

The CAEN V6533 control panel allows you to control a single V6533 module via the slow control interface. This section describes:

- How to start the control panel
- How to use the main window to control the channels
- How to get the detailed channel parameters window for each channel, and how to use it.
- How to get the channel status window for each channel

Starting the control panel. To start the HV control panel, first be sure that the readout program is active. The HV control panel is started via the command:

```
/usr/opt/vmusbcontrols/caenv6533panel name
```

where `name` is the name of a CAEN V6533 module in your `~/config/controlconfig.tcl` file.

This will bring up the control panel in the figure below:
Figure 7-1. CAEN V6533 control panel.

<table>
<thead>
<tr>
<th>name</th>
<th>Ch 0</th>
<th>Ch 1</th>
<th>Ch 2</th>
<th>Ch 3</th>
<th>Ch 4</th>
<th>Ch 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>setpoint</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>voltage</td>
<td>1.1</td>
<td>1.3</td>
<td>1.3</td>
<td>1.0</td>
<td>1.5</td>
<td>1.4</td>
</tr>
<tr>
<td>current</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>unit</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>
Appendix A. Reference material for the NSCL Scaler Display Program.

This appendix provides a reference for the NSCL Scaler display program. It is copied verbatim from the NSCL comprehensive documentation.

The scaler display software consists of two components:

**sclclient**

Connects to the NSCL data acquisition system buffer manager and accepts scaler and run state-change buffers. The software connects to a tclserver application and maintains a set of Tcl variables in that server that describe instantaneous and continuous scaler state.

**ScalerDisplay**

A script that starts up a tclserver that runs a script which takes the variables maintained by sclclient, a configuration file and produces a scaler display.

---

### sclclient

**Name**

*sclclient* — Maintain scaler state in a tclserver

**Synopsis**

*sclclient* [options]

**DESCRIPTION**

*sclclient* accepts scaler buffers from a DAQ systems (spdaq system at the NSCL), processes them and sends processed information to a tclserver program. Given an appropriate setup script to describe a visual presentation of the scalers, sclclient and tclserver together create a scaler display subsystem.

Command options (see OPTIONS below), describe how the program starts up. sclclient interacts with tclserver by sending it a set of tcl commands to maintain some global variables (see VARIABLES below). In addition, at key points, procedures are called that are assume to be loaded into the tclserver program by its setup scripts (see PROCEDURES below).
OPTIONS

-h tclserver_host

Defines the system on which the tcl server is running. It is possible for sclclient to run in a system remote from tclserver. By default, however the host connected to is localhost.

-p port_number

Defines the number of the TCP/IP port on which tclserver is listening. By default and convention, scaler display tclservers listen on port 2700.

-s spectrodaq_url

Defines the url of the spectrodaq buffer manager from which data will be acquired. This is of the form tap://hostname:2602/. By default this is tcp://localhost:2602/ causing data to be taken from the system sclclient is running on.

VARIABLES

sclclient maintains several variables and arrays in the TCL server to which it is connected. Scaler displays are therefore constructed by displaying the values of these variables and arrays as well as by providing procedures called by sclclient (see PROCEDURES below).

The variables sclclient maintains are:

Scaler_Totals

This is a TCL array indexed by scaler channel number (channels start at 0. Each element of the array is the total number of counts in that channel either since the scaler program started up or the run began, whichever happened latest.

Scaler_Increments

This is a TCL array indexed by channel number. The value of each element is the number of counts in that channel since the latest of the beginning of the run, starting sclclient, and the previous scaler buffer.

ScalerDeltaTime

This variable maintains the number of seconds in the most recently received set of scaler increments in seconds. If no scaler increments have been received, this variable is 0. ScalerDeltaTime can be used to calculate scaler rates.

ElapsedRunTime

This variable contains the number of seconds since the start of run, or when the scaler client program started, whichever is latest.
Appendix A. Reference material for the NSCL Scaler Display Program.

RunNumber
This variable contains the number of the current run. If not yet known, it has the value "Unknown" instead.

RunState
This variable has the known run state. It is any of HALTED, ACTIVE, or PAUSED, or lastly "Unknown" if the run state is not yet known.

RunTitle
Contains the title of the current run if known or the text "Unknown" if not.

PROCEDURES

In addition to maintaining the global variables described in VARIABLES above, sclclient calls procedures at well defined points in time. These procedures must be defined in the tclserver, even if only as empty procedures.

Procedures are not passed any parameters. The procedures sclclient requires are:

Update
Called whenever variables have been updated. The tclserver code here will usually refresh the display picture.

BeginRun
Called when a begin run is detected.

EndRun
Called when an end run is detected.

PauseRun
Called when a pause run is detected.

ResumeRun
Called when a resume run is detected.

RunInProgress
Called when the first data to come in is not a begin run. This indicates that sclclient started while a run is in progress.
Appendix A. Reference material for the NSCL Scaler Display Program.

EXAMPLES

The sample below starts out sclclient taking data from spdaq22 and feeding it to a tclserver on u6pc2 at port 2700:

Example A-1. Starting sclclient

/usr/opt/daq/bin/sclclient -s tcp://spdaq22:2602/ -h u6pc2 -s 2700

SEE ALSO

ScalerDisplay(1tcl), tclserver(1)

ScalerDisplay

Name

ScalerDisplay — Live Scaler Displays

Synopsis

export DAQHOST=datasourcecomputer

ScalerDisplay configfile

DESCRIPTION

This script provides a configurable scaler display for the NSCL Data Acquisition system. The script requires that:

• An environment variable named DAQHOST be defined to be the name of the computer that is taking data. At the NSCL this will usually be a system named spdaqnn where nn is a two digit number.

• A single command parameter provides the name of a TCL script that is used to configure the display. The full range of TCL functionality may be used by this configuration script. The display script, in addition defines several commands that are used to configure the display (See CONFIGURATION COMMANDS below).
Appendix A. Reference material for the NSCL Scaler Display Program.

In addition to configuring the display itself, ScalerDisplay supports the invocation of user written code at well defined points of operations. For more information about that see CALLOUTS below.

CONFIGURATION COMMANDS

The ScalerDisplay program understands the following object types:

channels
A channel is a scaler channel. It has a name and an index. The name is used to refer to and label the channel. The index is the offset into the set of scalers (numbered from 0) that contains that channel.

pages
A page is a set of scalers grouped together on one display page. Pages have a title, which is an arbitrary text string that is displayed at the top of the page when the page is active, and a Tabname which is used to select the page from the tabbed notebook widget that displays them.

lines
A line is a single scaler or a pair of scalers or blank displayed on a line of the scaler display.

stripcharts
A strip chart is a plot of the rate of one or more scaler channels or their ratios with respect to time. The strip chart part of the user interface is only visible if configured or at least one channel has been added to the chart.

Channels are defined using the channel command. The format of this command is:

```
channel [?-lowalarm value?] [?-hialarm value?] name index
```

The -lowalarm and -hialarm allow the user to set lower and upper limits on the 'healthy count rates'. If the actual count rates go outside those limits, the scaler channel will be in the alarm state.

Channels in the low alarm state, and their counts are displayed in the low alarm color which defaults to green (see CONFIGURATION below). The channel is considered to be in the high alarm state. Channels in the high alarm state and their counts are displayed in the high alarm color which defaults to red. Note that in the case of a ratio where one is in high alarm state and the other is in low alarm state, the colors of the names reflects the individual channel alarm states while the rate values are shown using the 'both alarm color' which defaults to orange.

Pages are defined via the page command. The format of this command is:
Appendix A. Reference material for the NSCL Scaler Display Program.

\begin{verbatim}
page Tabname "Some meaningful title string"

Note that the title string must be enclosed in quotes if it contains whitespace or other TCL word-separators. The Tabname text is used to label the tab of the page in the tabbed notebook widget that is used to display the scalers. If a page that is not currently being displayed has alarms, its tab will be displayed using either the low, high or both alarm color as appropriate to the alarm state of the channels within that page.

Single scaler lines are defined via the display_single command. The format of this command is:

\texttt{display_single Tabname channelname}

Where \texttt{Tabname} is the Tab name of a scaler page and \texttt{channelname} is the name of a scaler channel.

Ratio lines are defined via the display_ratio command:

\texttt{display_ratio Tabname numerator_channel denominator_channel}

Blank lines are defined via the blank command:

\texttt{blank Tabname}

Strip charts are defined using the stripparam stripratio and stripconfig commands. The format of the stripparam command is:

\texttt{stripparam channel}

The channel is the name of a channel defined by the channel command. The count rates of this scaler are added to the set of scaler rates on the strip chart using the next free line color and style (see CONFIGURATION below).
\end{verbatim}
Appendix A. Reference material for the NSCL Scaler Display Program.

`stripratio numerator denominator`

The ratio of the rates in the two channels numerator and denominator are added to the set of rates on the strip chart using the next free line color and style. The data set created will be named numerator_over_denominator.

`stripconfig [?-log 0|1?] [?-timeaxis seconds?]`

Configures either or both of the Y axis scale type and the length the time axis of the strip chart. If the parameter of the -log option is 1, the Y axis will initially be a logarithmically scaled axis. If 0, the Y axis will be linear. The seconds parameter to the -timeaxis option determines the number of seconds of scaler data that will be displayed on the time axis. The default for these options is to use a linear Y scale and a time axis that is 3600 seconds (1 hour) long. For additional strip chart configuration options, see the CONFIGURATION section.

**CONFIGURATION**

This section describes some advanced configuration techniques. The key to understanding the advanced this section is to realize that the scaler display program is just a Tcl/Tk script that is sourced in to a TclServer interpreter, and that your configuration file is also just a Tcl script that is sourced in after the display program script. As such, any defaults established by the scaler display program can be overridden by your configuration script.

**Alarm Colors**

Three global variables control the three alarm colors. lowColor contains the color to use when displaying channels in the low alarm state. hiColor contains the color to use when displaying channels in the high alarm state, and bothColor contains the color used when it is necessary to indicate that both alarm states are present. You may modify these colors within your script. Colors may be specified by name in many cases or by hexadecimal values. On linux systems, see the file: `/usr/X11R6/lib/X11/rgb.txt` for the list of known color names. Hexdecimal color values are given in any of the following forms: `#RGB` `#RRGGBB` `#RRRGGGBBB` or `#RRRGGGGBBBBB` where R,G,B are hexadecimal digits which, when taken together, form the Red, Green and Blue intensities of the color respectively. The two lines below both set the low alarm color to cyan (an equal mixture of Green and Blue):

```
set lowColor cyan
set lowColor #0ff
```
Appendix A. Reference material for the NSCL Scaler Display Program.

Tear off pages

The BLT tabset widget in which the scaler pages are displayed supports tear-off pages. When enabled, this feature allows you to tear off any page of the notebook into a new top level window. When the top level window is deleted, it is returned to the notebook. This feature and other BLT tabset configuration options can be configured by using the fact that the notebook widget path is stored in the global variable Notebook. Thus to enable the tear-off functionality the following line can be added to the configuration file:

```
$Notebook configure -tearoff 1
```

Strip chart line styles and colors

Channels on the strip chart widget are assigned line color and style by iterating over a list of colors and line styles. The procedure selectElementStyle does this and is expected to return a two element list. The first element of this list is the color of the line used to draw the element, and the second the argument to the -dashes configuration option for the element. You can modify the way in which colors and line styles are selected either by modifying the values in the color and linestyle list or by just overriding the definition of the selectElementStyle procedure.

The default implementation of selectElementStyle iterates through a list of colors stored in the global variable stripColors, selecting linestyles from the dash specifications in the global variable stripStyles. When colors are exhausted, the procedure steps to the next line style, resetting the index into the color list to zero. The two lines below add the color yellow to the set of colors that can be used to chart rates (yellow is low contrast relative to the white chart background so it was left off the default list), and a new linestyle where every other pixel is lit with the selected color or is background:

```c
lappend stripColors yellow
lappend stripStyles [list 1 1]
```

Strip chart configuration

The strip chart widget path is stored in the variable stripchartWidget. The Widget itself is only created when the first of stripparam, stripratio or stripconfig command is seen. You may therefore only configure the strip chart widget directly after one of these commands has executed in your configuration file. You can then use the stripchartWidget variable to configure the strip chart widget arbitrarily. The example below enables the display of gridlines on the plot surface, and moves the legend to the left side of the plot area:

```c
stripconfig -timeaxis 3600;  # Trick to get the widget defined....
$stripchartWidget grid configure -hide 0
```
Appendix A. Reference material for the NSCL Scaler Display Program.

For more information about how you can configure the stripchart at its elements, see the BLT stripchart widget documentation.

CALLOUTS

The scaler script will invoke user written procedures defined in the configuration script (or scripts sourced by it) at well defined points of the run. These callouts can be used to provide functionality not originally foreseen by the program.

UserUpdate
UserUpdate, if defined, is called by the script whenever it has updated the displays. No parameters are passed in to the procedure but several global variables are useful (see GLOBAL VARIABLES below).

UserBeginRun
UserBeginRun is called at the beginning of a run, if it has been defined. No parameters are passed.

UserEndRun
UserEndRun, if defined, is called at the end of a run. No parameters are passed.

GLOBAL VARIABLES

The following global variables are useful within user callouts.

RunNumber
The number of the current run.

RunTitle
A string containing the title of the current run.

Scaler_Totals
An array indexed by scaler channel number containing the total number of counts in each channel.

Scaler_Increments
An array indexed by scaler channel number containing the number of counts in the last time increment (see also ScalerDeltaTime)
ScalerDeltaTime

The number of seconds of counts represented by the Scaler_Increments array elements.

ScalerMap

An array indexed by scaler names. Each element of this array is the index of the corresponding scaler. For example, if you have defined a channel named george, ScalerMap(george) will be the scaler channel index associated with george.

scalerWin

This global is the name of the widget into which the scaler display will be drawn, or "" if the display is drawn into "."

If you are adding more elements to the GUI you can use this to know where to manage these new elements. For example:

```tcl```
checkbutton $scalerWin.silence -text {Silence Alarms} -command [silence]
```
creates a checkbutton that is a child of the scaler display page and can be packed on that page.

If you are using the scaler display program from within SpecTcl, you can set this widget to allow the scaler display program to pop up in a separate top level. For example:

```tcl```
set scalerWin [toplevel .scaler]
source /usr/opt/daq/current/Scripts/scaler.tcl
```

Creates the scaler display in a new top level widget called .scaler

EXAMPLE(S)

```tcl```
# Define the scaler channels:
# These can be in any order, IÂ’m just copying the order from the original file. my preference in fact would be to go in channel order.
# This is a TCL script with commands Â“channelÂ” - to define a channel name/buffer position correspondence
# Â“pageÂ” - To define a scaler page.
# Â“display_singleÂ” - To define a single scaler line in a page.
#
channel gas.PIN.cfd 0
channel gas.qA.cfd 16
channel gas.qB.cfd 17
```

138
Appendix A. Reference material for the NSCL Scaler Display Program.

channel gas.gc.cfd 18; # is this a typo in the original file?
channel gas.qD.cfd 19
channel gas.de.cfd 1
channel gas.Ge.cfd 2
channel gas.Ge.cfd_B-OFF 12
channel gas.PS.cfd 5
channel gas.PS.cfd_B-OFF 13
channel I2.SCI.N 3
channel I2.SCI.S 4
channel TA.BaF2 6
channel master.gated 11
channel master.free 10
channel cpu.lam.TO 7
channel clock.gated 9
channel clock.free 8
channel beam.cycle.on 14
channel beam.cycle.off 15

# Next define the pages, their long titles and the tab name
# and their contents.
# I've defined the page ALL to be the original page
# as well as some additional pages so that you can get the
# idea of how you can use this to organize the display if you want to.
# If you don't, you can rip out the extra pages.
#
#
page ALL "Gas Cell DAQ All Scalers"
display_single ALL gas.PIN.cfd
display_ratio ALL gas.qA.cfd gas.qB.cfd
display_ratio ALL gas.gc.cfd gas.qD.cfd
display_single ALL gas.Ge.cfd
display_ratio ALL gas.Ge.cfd gas.Ge.cfd_B-OFF
display_ratio ALL gas.PS.cfd gas.PS.cfd_B-OFF
display_ratio ALL I2.SCI.N I2.SCI.S
display_single ALL TA.BaF2
display_ratio ALL master.gated master.free
display_ratio ALL cpu.lam.TO master.gated
display_ratio ALL clock.gated clock.free
display_ratio ALL gasN4.de.cfd gasN4.PIN.cfd
display_ratio ALL beam.cycle.on beam.cycle.off

# If you only want the first page, then remove all lines
#---------------------------- cut below here -------------------

# A second page:
# Just showing the livetime information mostly.

page Livetime "Live time information"
display_ratio Livetime master.gated master.free
display_ratio Livetime clock.gated clock.free
display_ratio Livetime cop.lam.TO master.gated
Appendix A. Reference material for the NSCL Scaler Display Program.

# A third page showing only the gas cell:

page GasCell "Gas cell scalers"

display_single GasCell gas.PIN.cfd
display_ratio GasCell gas.qA.cfd gas.qB.cfd
display_ratio GasCell gas.qC.cfd gas.qD.cfd
display_single GasCell gas.qE.cfd
display_ratio GasCell gas.qE.cfd gas.qE.cfd_B-OFF
display_ratio GasCell gas.PS.cfd gas.PS.cfd_B-OFF

# Do a strip chart of the live master rates and the
# Livetime computed by clock.gated/clock.free:

stripparam master.gated
stripratio clock.gated clock.free

BUGS AND RESTRICTIONS

- This software only available with release 8.0 and later of nscldaq.
- The startup script for this software requires the TCP/IP server port manager that made its debut with release 8.0 of the software.
- The BLT stripchart widget used to display rate strip charts requires that channels displayed on it have names that consist only of letter, digits, underscores an periods. There are no restrictions on channel names that are not displayed on the strip chart, however users are encouraged to maintain the BLT restrictions.

SEE ALSO

tclserver(1), sclclient(1)
Appendix B. Validity checks on configuration parameter values

This Appendix is sort of a bridge between a user guide and a chunk of internals. In this section I will describe the configuration parameter validation system. The purpose of this system is to provide an extensible scheme for checking the validity of parameters that will be stored in a device’s configuration data.

From the point of the configuration system API, a device’s configuration is just a collection of string pairs. The first string of each pair is the name of the configuration parameter, the second string, the value. Interpretation of a configuration parameter as anything other than a string is up to the device.

Since the configuration is defined in a script and then later applied to the device, it is useful to have some scheme for validating parameters prior to application. This is where the validity check system described in this appendix comes in.

Associated with each device is a configuration. The configuration contains parameters. Associated with each parameter is a value and a validator. Prior to allowing a parameter’s value to be set, the validator for the parameter is invoked with the new proposed value as a parameter. If the value is valid, the validator returns true otherwise false. If the validator returns true, the configuration parameter value is set to the new proposed value. If false the value is unchanged and the configuration command throws an error at Tcl level. The Tcl error is caught by the code that is running the interpreter and turned into an error message, and the begin run operation in progress is aborted.

A validator is just an ordinary function, however the CConfigurableObject class provides several static member functions that can be used to perform common validations. The remainder of this appendix describes:

1. The definition of a validation function
2. The canned validators provided by the CConfigurableObject class

**The Validation function.** The validation function has the following prototype in CConfigurableObject.h:

```cpp
typedef bool (*typeChecker)(std::string name, std::string value, void* arg);
```

Parameters are as follows:

**Type:** std::string  
**Name:** name
Appendix B. Validity checks on configuration parameter values

Description: Name of the configuration parameter being validated.

Type: std::string
Name: value
Description: The new proposed value for name

Type: void*
Name: arg
Description: A parameter that is passed without interpretation to the validator. This parameter is passed to the CConfigurableObject object when the configuration parameter is defined. For the canned validators, this is used to further refine the validation. For example, the integer validator uses it to specify an optional restriction on the range of the parameter value.

Canned validators provided by CConfigurableObject. Several common validators are provided as static member functions of the CConfigurableObject class. These are used throughout the configurations for the devices that are supported by the VM-USB readout framework. This section will describe the validators, and the meaning of the arg parameter they each take.

isInteger
Returns true (validates), if the new proposed value can be interpreted as an integer. The additional args parameter is pointer to a CConfigurableObject::Limits structure. This struct has two fields, s_atLeast and s_atMost which define lower and upper limits on the value and are themselves CConfigurableObject::limit structs which have fields: s_checkMe, a bool which is true if the limit should be checked. and s_value which is the value of the associated limit.

The arg pointer can be NULL which is exactly the same as a pointer to a struct both of whose members have s_checkMe false, or put more simply, no range checking will be performed.

isBool
Returns true if the proposed value is a bool. The arg parameter is not used. Valid true values are true, yes, 1 on and enabled. Valid false values are the natural complements to the true values: false, no, 0 off and disabled.

isEnum
Returns true only if the proposed value is one of a specific set of textual values. The arg parameter is a pointer to a std::set<string> containing the acceptable strings.

isList
Returns true only if the proposed value is a well formed Tcl list. The arg parameter allows additional constraints to be placed on the list including constraints on the number of elements of the list and validations on the list elements. Note that this is the most general list checker, and there are additional more specific list validity checkers that are much simpler to use.
Appendix B. Validity checks on configuration parameter values

`arg` is a pointer to a CConfigurableObject::isListParameter struct. This has the following fields:
- `s_allowedSize` which can place limits on the size of the list and
- `s_checker` which allows you to apply a validator to each element of the list.

`s_allowedSize` in turn is a CConfigurableObject::ListSizeConstraint struct that has the elements
- `s_atLeast` and `s_atMost` which are both of type CConfigurableObject::limit and define optional limits on the fewest and largest number of items that can be in the list.

`s_checker` is of type CConfigurableObject::TypeCheckInfo which is an std::pair<CConfigurableObject::typeChecker, void*>. The elements of the pair are a validation function and its argument respectively. This checker will be applied to all elements of the list. All elements of the list must pass validation for the list to pass validation.

isBoolList
Validates a list ensuring that all elements are bool. The `arg` parameter points to a CConfigurableObject::ListSizeConstraint that constrains the size of the list. If NULL no size constraint is applied. Note that in this case, an empty list is perfectly legal.

isIntList
Validates a list ensuring that all elements are int. No validation on the range of the list elements is performed, use `isList` directly to obtain that.

The `arg` is a pointer to a CConfigurableObject::ListSizeConstraint struct which, if supplied places constraints on the number of elements in the list. If NULL the length of the list is unconstrained.

isStringList
Validates a list based only on the size of the list and any constraint on the list size. The `arg` is a pointer to a CConfigurableObject::ListSizeConstraint struct. If not NULL this struct places lower and upper limits on the list size.
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

SpecTcl and the VM-USB readout collaborate to use the `daqconfig.tcl` configuration file to provide zero programming setup that both configures the readout of the experiment and the production of raw parameter spectra. This appendix describes how to add event processors that produce parameters that are computed from the raw parameters and made available for histogramming.

The appendix is organized as follows:

1. The SpecTcl event processing pipeline and event processors are introduced, and a short guide that describes how to write and add an event processor is provided.
2. The treeparameter package is introduced, and its relevance to the task of writing pseudo parameter event processors described.
3. An example is worked through to show how to do all of this.

C.1. SpecTcl’s event processing pipeline

This section describes SpecTcl’s event processing pipeline and shows how to add an event processor to that pipeline.

C.1.1. The Event Processing Pipeline

SpecTcl is an application framework. Application frameworks turn the idea of writing programs upside down and inside out. We are used to writing programs by writing some main program and then maybe using some canned libraries to fill in commonly re-used components.

An application framework provides the main program. The application framework takes care of the normal flow of program operation. You provide a library of application specific code that is called by the framework in response to well defined conditions and events.

For a histogramming framework like SpecTcl, the most important event is the availability of an event. SpecTcl users are responsible for ensuring that they provide code that maps the raw event into a set of SpecTcl parameters. SpecTcl is then able to histogram the data from those parameters in accordance to parameter, spectrum, gate and gate definitions.

SpecTcl’s framework design recognizes that you may want to create your SpecTcl parameters in several stages. For example, an early stage may unpack the raw event into parameters, later stages may refer to...
these unpacked parameters producing calibrated parameters. Still later stages may compute physically meaningful parameters and so on.

SpecTcl’s mechanism for supporting this sort of processing is to allow you to write several event processors. These event processors form a logical event processing pipeline.

Each event processor you write has access to the raw event as well as any parameters that have been assigned values by previous step in the event processing pipeline. Event processors take the form of a C++ class. Event processor classes must be instantiated into objects which are then registered with SpecTcl. When events are available, event processor objects are called in the order in which they are registered.

If you are not familiar with C++ programming, you may want to look at a C++ tutorial. The slides for a C++ course that has been given at the NSCL are available at: http://www.nscl.msu.edu/~fox/talks.htm

The remainder of this section will

1. Show how to set up a development directory for creating an event processor to be used with VM-USB SpecTcl
2. Show a template for defining an event processor to be used with the VM-USB SpecTcl
3. Show how to register an event processor with SpecTcl
4. Show how to run your SpecTcl online with the VM-USB data acquisition system.

C.1.1. Setting up a development directory

When you write event processors for the VM-USB data taking system and SpecTcl, you will be extending the VM-USB SpecTcl skeleton rather than the base SpecTcl skeleton. This is done by copying the VMUSBSpecTcl sources into an empty directory and cleaning out any object and executable files for example:

```
cd /some/directory/i/can/write/to
mkdir myspectcl
cp /usr/opt/vmusbSpecTcl myspectcl
cd myspectcl
make clean
```

Once you have done this, you can edit the files you have copied, add new files and create an extended VM-USB SpecTcl program.
C.1.1.2. A template for event processors

Event processors are C++ classes. This section describes the code for the header and implementation files for an event processor that does nothing.

You can use this code as a starting point for your own event processors.

Event processors are classes that derive (inherit) from the CEventProcessor base class. The header for this file is EventProcessor.h in the include directory of the SpecTcl base installation.

The minimal event processor header for a custom event processor is shown below:

Example C-1. Minimal Event Processor header file - MyEp.h

```cpp
#ifndef __MYEP_H
#define __MYEP_H

#include <EventProcessor.h>

class MyEp : public CEventProcessor
{
public:
  virtual Bool_t OnAttach(CAnalyzer& rAnalyzer);  
  virtual Bool_t OnBegin(CAnalyzer& rAnalyzer, 
                         CBufferDecoder& rDecoder);  
  virtual Bool_t OnEnd(CAnalyzer& rAnalyzer, 
                       CBufferDecoder& rBuffer);  
  virtual Bool_t OnPause(CAnalyzer& rAnalyzer, 
                        CBufferDecoder& rDecoder);  
  virtual Bool_t OnResume(CAnalyzer& rAnalyzer, 
                         CBufferDecoder& rDecoder);  
  virtual Bool_t OnOther(UInt_t nType, 
                        CAnalyzer& rAnalyzer, 
                        CBufferDecoder& rDecoder);  
  virtual Bool_t OnEventSourceOpen(std::string name);  
  virtual Bool_t OnEventSourceEOF();  
  virtual Bool_t operator()(const Address_t pEvent, 
                            CEvent& rEvent, 
                            CAnalyzer& rAnalyzer, 
                            CBufferDecoder& rDecoder);  

};
#endif
```

146
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

1. This standard construction along with the trailing `#endif` at the end of the file is called an **include guard**. It is good practice to put something like this in all your header files. It ensures that multiple includes of the same header don’t result in compiler errors.

2. Member data and member functions have specific **visibility**. This visibility determines who can call or see the function of data. Public visibility allows functions from outside of the class to call or see the members. Since SpecTcl’s analyzer will be invoking member functions of our class, it is important to specify them as public. In C++ all members specified after a visibility specification like `public:` are public until subsequent visibility specification is made.

3. Since the class `MyEp` will be derived from the `CEventProcessor` base class, the compiler will need to see the definition of `CEventProcessor`. Therefore we include its header in this file.

4. This line starts the class definition. The class definition continues until the closing `};`. The `public CEventProcessor` is what the C++ compiler needs to see to tell it that `MyEp` is derived (inherits) from `CEventProcessor`.

Within the curly brackets you place definitions of class (static) and instance data and methods (functions). In this example, of a minimal event processor, we will only place the methods that are defined in the base class that often must be overridden in a real event processor. Note that if you don’t actually need to provide code for one of these methods you don’t need to define it or implement an empty function. The base class does that for you.

Unless otherwise mentioned, each method is expected to return `kFTRUE` on success and `kFALSE` on failure. If `kFALSE` is returned for a method that’s called in event pipeline processing, the remainder of the event pipeline is aborted.

5. When the event processor is registered with SpecTcl, SpecTcl will call the `OnAttach` method. This method is supposed to do any one-time initialization and setup required that cannot be done by an object constructor. The event processor should return `kFTRUE` on success and `kFALSE` if it fails. At present, no action is taken on the return value but future versions of SpecTcl may remove an event processor whose `OnAttach` returns `kFALSE`.

The parameter `rAnalyzer` is a reference to an object in SpecTcl called the `Analyzer`. The analyzer is responsible for controlling the flow of analysis of events. With the introduction of the SpecTcl API class, most event processors don’t need to access this parameter.

To see the services offered by the Analyzer see the SpecTcl include file `Analyzer.h`.

6. Called for each registered event processor in the event processing pipeline when SpecTcl sees a begin run buffer/event. The parameters are as follows:

   **Type:** `CAnalyzer&`
   **Name:** `rAnalyzer`
   **Description:** A reference to the analyzer. See the description of `OnAttach` for more information about this parameter.
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

**Type:** CBufferDecoder&  
**Name:** rDecoder  
**Description:** A reference to the SpecTcl object that is supposed to know the overall buffer structure. This object supplies several services that may be of use to `OnBegin` for example, you can use this object to retrieve the run number, and use that run number to load calibration constants that depend on the run being analyzed. See the `BufferDecoder.h` header for the set of services provided by this object.

- Called when SpecTcl encounters an end of run buffer/event. The parameters are the same as for `OnBegin`. One application of this function is to trigger an automated output of some analysis products (e.g. write spectra), when you are analyzing a batch of runs sequentially.

- Called when SpecTcl encounters a pause run buffer/event. Note that the VM-USB DAQ system does not support pausing runs. The parameters to this function are the same as for `OnBegin`.

- Called when SpecTcl encounters a resume run buffer/event. Note that the VM-USB DAQ system does not support pause/resume. The parameters to this function are the same as for `OnBegin`.

(10) Called when SpecTcl encounters a buffer/event of some type other than the ones specifically handled by event processors (for example a scaler buffer). The parameter `type` is the type of the buffer. Buffer type codes are defined symbolically in the `buftypes.h` header file.

(11) This method is called when SpecTcl opens a new event source `name` is the 'connection identifier' for the event source In general the connection identifier is a keyword that specifies the event source type followed by a string that is meaningful in the context of that type.

If the type is `File` SpecTcl has just been attached to an event file and the string following the colon is the path to the file.

If the type is `Pipe from:` SpecTcl has just been attached to a pipe data source. The string following the colon is the command that is on the other end of the pipe. Pipe event sources are most frequently used when analyzing data taken online.

(12) Called when SpecTcl encounters an end of file condition on the current event source.

(13) The function call operator is called when SpecTcl needs an event analyzed into parameters. The parameters are as follows:

**Type:** const Address_t  
**Name:** pEvent  
**Description:** Pointer to the raw event from the data source. The `Address_t` is an alias for void *. See the sample implementation, however for more information about how to access an event in a portable manner.

**Type:** CEvent&  
**Name:** rEvent  
**Description:** A reference to the event’s parameters object. The `rEvent` is an object that acts very much like an array. The indices if the array are parameter numbers the values are parameters unpacked from or computed from the raw event or other parameter. The job of the event unpacker is to provide values to the set of `rEvent` elements that represent the data in the event.
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

**Type:** CAnalyzer&  
**Name:** rAnalyzer  
**Description:** A reference to the analyzer. See the description of OnAttach for more information about this parameter.

**Type:** CBufferDecoder&  
**Name:** rDecoder  
**Description:** A reference to the SpecTcl object that is supposed to know the overall buffer structure. This object supplies several services that may be of use to OnBegin for example, you can use this object to retrieve the run number, and use that run number to load calibration constants that depend on the run being analyzed. See the BufferDecoder.h header for the set of services provided by this object.

A template for the implementation of an event processor is shown in the example below. This event processor also does nothing. It is assumed to not be the first event processor in the event processing pipeline as that event processor is the one responsible for the automated unpacking of VM-USB data into parameter values.

**Example C-2. Template Event Processor**

```cpp
#include <config.h>  
#include "MyEp.h"  
#include <Analyzer.h>  
#include <BufferDecoder.h>  
#include <Event.h>  

using namespace std;

Bool_t MyEp::OnAttach(CAnalyzer& rAnalyzer)  
{  
  return kfTRUE;
}

Bool_t MyEp::OnBegin(CAnalyzer& rAnalyzer,  
    CBufferDecoder& rDecoder)  
{  
  return kfTRUE;
}

Bool_t MyEp::OnEnd(CAnalyzer& rAnalyzer,  
    CBufferDecoder& rBuffer)  
{  
  return kfTRUE;
}

Bool_t MyEp::OnPause(CAnalyzer& rAnalyzer,  
    CBufferDecoder& rDecoder)
```

149
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

```c++
{  
  return kfTRUE;
}
Bool_t          
MyEp::OnResume(CAnalyzer& rAnalyzer,  
                CBufferDecoder& rDecoder)  
{  
  return kfTRUE;
}
Bool_t          
MyEp::OnOther(UInt_t nType,  
              CAnalyzer& rAnalyzer,  
              CBufferDecoder& rDecoder)  
{  
  return kfTRUE;
}
Bool_t          
MyEp::OnEventSourceOpen(std::string name)  
{  
  return kfTRUE;
}
Bool_t          
MyEp::OnEventSourceEOF()  
{  
  return kfTRUE;
}
Bool_t          
MyEp::operator()(const Address_t pEvent,  
                 CEvent& rEvent,  
                 CAnalyzer& rAnalyzer,  
                 CBufferDecoder& rDecoder)  
{  
  return kfTRUE;
}

1 Includes the SpecTcl configuration header. When SpecTcl is built the build software creates this file which has some definitions that vary from system to system. This include must appear first in implementation (.cpp) files.

2 Includes the header that defines the class. Each implementation file must include its own header. It is a best practice to include that immediately after including the configuration file so that you know that the header can stand on its own rather than being dependent on other headers.

3 These includes include definitions for classes that are used as parameters to methods (functions) defined as member functions for the MyEp class.

If a type is used in a header in a way that the compiler does not need to know the exact shape of the type (e.g. as a pointer or as a reference only), the include can be omitted. Typically, however
implementations will need to know the shape of the type, and should therefore include the type’s header.

Deferring the includes to the implementation file wherever possible is a best practice that avoids creating dependency loops in headers on other headers.

This imports the names from the C++ std namespace into the global namespace so that you can omit the text std:: in many parts of your implementation file.

This shows how to implement a typical class method. class methods look like any ordinary C function however they can access object data, and should have classname:: prior to the function name.

All of the event processor methods should return kfTRUE on success. Since none of these members actually do anything they all are assumed to succeed.

In a later section we will work through a sample event processor filling in actual code that turns this class into a useful event processor.

C.1.1.3. Registering an event processor MyEp.cpp

In the previous section we wrote an event processor class that does nothing. In order for an event processor to be called by SpecTcl:

• An instance of the class must be created.
• The instance (an object), must be registered with the SpecTcl event processing pipeline.

All of these are done by modifying a file named MySpeTclApp.cpp in your development directory

Before describing how to do this, I want to make the distinction between a class, and an object, or instance of the class. A class defines a data type and a set of operations on that data type. An instance or object (the two words are used interchangeably), defines a variable of that type, on which those operations can be performed.

Defining an instance of the MyEp class requires including the header for that class in MySpecTclApp.cpp and defining a variable just like any other C variable, however the type will be MyEp.

Start by editing MySpecTclApp.cpp and locating the block of header includes and adding an include for MyEp.h. When you’ve done this you should have a block of code that looks like:

```
#include <config.h>
#include "MySpecTclApp.h"
#include "EventProcessor.h"
#include "TCLAnalyzer.h"
#include <Event.h>
```
#include <TreeParameter.h>
#include "MyEp.h"

Next, locate the declarations of myApp and add a definition for an instance of MyEp called myEventProcessor. This should result in code like:

```cpp
CMySpecTclApp myApp;
CTclGrammarApp& app(myApp); // Create an instance of me.
CTCLApplication* gpTCLApplication=&app; // Findable by the Tcl/tk framework.
static MyEp myEventProcessor;
```

Finally, locate the method CMySpecTclApp::CreateAnalysisPipeline(CAnalyzer& rAnalyzer) and register your event processor at the end of the pipeline by adding it after the existing element(s):

```cpp
void CMySpecTclApp::CreateAnalysisPipeline(CAnalyzer& rAnalyzer)
{
    RegisterEventProcessor(* (new CStackUnpacker), "adc-data");
    // RegisterEventProcessor(* (new CRateEventProcessor), "rate-stripchart");
    RegisterEventProcessor(myEventProcessor, "stupid-processor");
}
```

This registers myEventProcessor at the end of the event processing pipeline, associates the name stupid-processor with the processor and calls the OnAttach method associated with the object myEventProcessor.

Finally, to use your event processor, you must incorporate it into the Makefile for your SpecTcl development directory and build a new tailored SpecTcl. To do this, you must edit the Makefile. Locate the definition of OBJECTS and add your file to it:

```
OBJECTS=MySpecTclApp.o MyEp.o
```

Once this is done you can build your SpecTcl:

```
make
```
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

C.1.1.4. Using your extended SpecTcl with the online system

Once you have tested your SpecTcl with offline data you may want to incorporate it into an online system. There are three ways to do this:

1. Point the SpecTcl startup script for the experiment account at your SpecTcl rather than the one in
/usr/opt/vmusbSpecTcl
2. Install your SpecTcl to replace the centralized SpecTcl.
3. Centrally install your SpecTcl somewhere else, but point the scripts that start it up to this alternate location (in both the experimental account and in the skeleton directories).

Which of these choices you select depends on how generic the event processors are. If they are generic enough to span the range of experiments planned, you may wish to centrally install this version using either the second or third methods.

Adjusting the start script to point to your SpecTcl. SpecTcl is started by the script in
~/.bin/startSpecTcl simply edit this script to start the SpecTcl you built. It is important to ensure that the current working directory at the time you start SpecTcl is still the ~/.spectcl directory to ensure the correct startup scripts are run.

Replacing the standard SpecTcl. The standard VMUSB SpecTcl is installed in
/usr/opt/vmusbSpecTcl. To replace it with your version you should, as root:

cd /usr/opt
mv vmusbSpecTcl vmusbSpecTcl.saved
mv /path/to/your/spectcl myvmusbSpecTcl
chmod -R a+rx myvmusbSpecTcl
ln -s myvmusbSpecTcl vmusbSpecTcl

Once you have done this the normal startup scripts will start your tailored SpecTcl rather than the originally installed one. You can go back to using the original by:

cd /usr/opt
rm vmusbSpecTcl
ln -s vmusbSpecTcl.saved vmusbSpecTcl

Centrally installing your SpecTcl and adjusting the scripts. To do this:

cd /usr/opt
mv vmusbSpecTcl vmusbSpecTcl.saved
mv /path/to/your/spectcl myvmusbSpecTcl
chmod -R a+rx myvmusbSpecTcl

Having done this, adjust your scripts as in the first option. If you want to make this SpecTcl standard for all new accounts, locate the startup script in /usr/opt/skel or /etc/skel and edit it as well.
C.1.2. Writing event processors

Now let’s look at the stuff you may have to put into the implementation of an event processor. We are going to implement an event processor that does the simple charge division calculation for parameters named left and right to produce a new parameter named x.position.

To do this we will see:

- How to define our new parameter x.position
- How to locate the parameters our calculation depends on (left and right).
- How to compute the position on an event by event basis, this includes recognizing when the parameters we need are actually present in an event.

Our starting point for this adventure will be the template event processor we created and added to SpecTcl in the previous section.

C.1.2.1. Creating a new SpecTcl parameter in an event processor

SpecTcl’s application programming interface (API) is defined in what is called a singleton class. A singleton class is one that ensures that there can be only one object of that class in the entire application. We will need to get and interact with that object to add our parameter.

The API class is called SpecTcl and it is defined in the SpecTcl.h header. We will need three methods from that class:

```
static SpecTcl* getInstance()
```

The static qualifier means that the function is a class method rather than an instance method.

Instance methods require an object if they are to be called. Class methods can be called by using the notation ClassName::methodname(parameters) e.g.

```
SpecTcl* api = SpecTcl::getInstance();
```

```
UInt_t AssignParameterId()
```

This is an instance method that assigns the lowest unused parameter number. Since it is an instance method an object is required to call it for example:

```
UInt_t parameterNumber = api->AssignParameterId();
```
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

CParameter* AddParameter (std::string name, UInt_t Id, std::string Units);

Creates a new SpecTcl parameter. The arguments to this function are as follows:

**Parameter:** : name

**Type:** : std::string

**Description:** : The name of the parameter you are creating. This must be a parameter that does not yet exist in SpecTcl’s parameter dictionary.

**Parameter:** : Id

**Type:** : Uint_t

**Description:** : The parameter number of the new parameter. This will be the index of the parameter in the rEvent array. Normally you use the AssignParameterId to choose an id as these also must be unique.

**Parameter:** : units

**Type:** : std::string

**Description:** : Units of measure to be associated with the parameter. If, for example we computed our parameter calibrated to centimeters we might put "cm" here. If the units are not known you can use an empty string or "channels" or whatever else you’d like to see on the axis labels associated with this parameter.

So let’s get to work. The first thing we will need to do is to store the parameter Id once we’ve gotten it so that we have it available when creating our parameter for each event. The best way to do this is to create member data for our class MyEp.

Member data is data that is associated with each instance of a class. You can treat this member data as local data within the member functions of that class. The lifetime of member data is the lifetime of the object.

Edit the file MyEp.h we created during the last section and add member data definition for the parameter id of the position:

**Example C-3. Adding member data to an event processor**

```cpp
... class MyEp : public CEventProcessor {
private:
   UInt_t m_positionId;

public:
   virtual Bool_t OnAttach(CAnalyzer& rAnalyzer); // Called on registration.
...```
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

1. The `private:` visibility specification means that only class members can see this data. A best practice for data in classes is that of minimal visibility. This means make data only as visible as necessary for the functionality of the class. Typically only private and protected data should be declared.

2. This declares a member variable `m_positionId` that is an unsigned integer and will hold the parameter id of the position parameter. The use of the prefix `m_` for member data is part of a naming standard I follow that makes it easy to see at a glance which data referenced by a function are member data.

Next we need to actually create our parameter and store its id in `m_positionId`. Since this is done only once. We can do this in the `OnAttach` member data. Edit `MyEp.cpp` so that the body of that function looks like this:

Example C-4. OnAttach creating the x.position parameter.

```cpp
#include <SpecTcl.h>  
...
Bool_t
MyEp::OnAttach(CAnalyzer& rAnalyzer)
{
    SpecTcl* api = SpecTcl::getInstance();  
    m_positionId = api->AssignParameterId();  
    try {
        api->AddParameter("x.position",
                         m_positionId,
                         "channels");
    }
    catch (...) {
        m_positionId = 0xffffffff;  
        return kfFALSE;
    }
    return kfTRUE;
}
```

1. In order to make use of the SpecTcl API class we need to include its header.

2. The API object is a singleton. We must use its `getInstance` static member to get a pointer to the API object.

3. This requests SpecTcl to allocate a new, unused parameter ‘slot’. The function returns the parameter ID which we store in the member data we created in our previous step of this process.

4. SpecTcl API functions often report errors by throwing exceptions. An exception is an error which must be explicitly handled, else the program will exit. The way that exceptions are handled in C++ is to use a `try/catch` block. The code within the `try` block is evaluated. If an exception occurs the `catch` blocks are searched for matches and the matching one is executed. If there is no matching `catch` block, the stack is unwound and `catch` blocks in the caller are searched. If there are no `catch` blocks and the stack is completely unwound, the program aborts.
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

➎ Creates the parameter x.position using the parameter Id we allocated and the units channels
➏ If the parameter could not be created, the position id is set to something easily recognized and 
kfFALSE is returned indicating that OnAttach failed to properly initialize.

Once you’ve made these changes, compile and run SpecTcl. You should now have a new parameter
named x.position.

C.1.3. Locating dependent parameters

The next thing we will need to do, is to be sure that our event processor knows where the parameters it
depends on, left and right. If you have followed along so far, this section should present very few
surprises.

We will need to provide some data members to keep track of the locations in the rEvent array in which
we should expect to see those parameters. We will also need to add code to locate those elements. Since
in general, the set of parameters is not yet defined when OnAttach is called, we’ll do the lookup in
OnBegin as they are sure to be defined by the time we start analyzing data.

In order to make knowing if we know enough to compute the x.position parameter easy on the event
processor, we will also create a flag that indicates if we have successfully figured out all of the
parameters.

To add the appropriate data members edit MyEp.h so that its data members look like this:

Example C-5. Data members for input parameters

    ...
    class MyEp : public CEventProcessor
    {
    private:
        UInt_t m_positionId;
        UInt_t m_leftId;
        UInt_t m_rightId;
        bool m_canCompute;
    ...

Once more if you have followed along so far the only possible surprise is the fact that C++ has a bool
data type which C does not have. m_leftId and m_rightId will hold the parameter ids of the left and
right parameters of the charge division respectively while m_canCompute will be true if all of the
parameters have been looked up and the output parameter has been created.
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

Before looking up the parameters we need to know one more API function and something about what it returns:

```cpp
CParameter* FindParameter(std::string name);
```

`SpecTcl::FindParameter` searches for a parameter definition by name. It returns a pointer to the parameter definition object which is of type `CParameter`. If no matching parameter is found a null pointer is returned instead.

You can look at the full definition of `CParameter` and all of its members by examining the header `Parameter.h`. We only need to get the parameter id. It’s not immediately obvious how to do that by looking at `Parameter.h`. Note, however, that a `CParameter` inherits from `CNamedItem` which is a standard base class that SpecTcl uses for objects that have names and ids (parameters, spectra and gates to name three). Looking in `NamedItem.h` reveals the that the `getNumber` member function gives us the number associated with a named item. For parameters, this is the parameter id.

Therefore, we add code to `OnBegin` that looks like this:

**Example C-6. Locating dependent parameters in `OnBegin`**

```cpp
Bool_t MyEp::OnBegin(CAnalyzer& rAnalyzer, CBufferDecoder& rDecoder)
{
    SpecTcl* api = SpecTcl::getInstance();
    CParameter* pLeft = api->FindParameter("left"); ➊
    CParameter* pRight = api->FindParameter("right");

    if (pLeft && pRight) { ➋
        m_leftId = pLeft->getNumber(); ➌
        m_rightId = pRight->getNumber(); ➍
        m_canCompute = m_positionId != 0xffffffff; ➍
    } else {
        m_canCompute = false; ➎
    }

    return kfTRUE; ➏
}
```

➊ These three calls locate the parameter objects we want. Note that `pLeft` and/or `pRight` might be null.
Since one or both of the parameters we want may not be defined, we ensure that they both are before attempting to call any member functions belonging to the parameter objects. Calling a member function on a null object will, of course abort the program.

These two lines get and store the parameter ids associated with our parameters.

If in addition to both of our dependent parameters being defined, we were able, in OnAttach, to create our result parameter, we note that we can compute our output parameter.

If either of the parameters we depend on does not exist, we mark that our output parameter cannot be computed.

It might seem that since it is not possible to compute our parameter that we should return kFALSE. That is an option, however that would abort the processing of the remaining OnBegin members of any other event processors that come after us in the event processing pipeline. In general, the error we have is not serious enough to do that. We can run SpecTcl, but we won’t get our position parameter. The only other thing that we might have done to make it clear why this is the case would be to output an error message. This is left as an exercise to the reader.

C.1.4. Computing the parameter

The final step in our event processor is computing the output parameter on an event by event basis. We can only compute an output parameter if

• m_canCompute is true
• The rEvent array is long enough to contain our input parameter ids and the parameters have been assigned values earlier in the event processing pipeline.

Earlier we spoke of rEvent as if it can be treated as an array of double precision values. In fact this is not exactly true. rEvent is an object of type CEvent. See the Event.h header for a definition of this class. The important thing to know about CEvent is that it expands as needed, that it exports a member function size that tells you how big it is at any given time and that each element of this array like object is also an object of the type CValidValue which is a double with the added property that it knows when it has been set and can report that fact through its isValid member function.

Armed with this knowledge we are ready to write the operator() member function. Recall that this function is called each time an event is to be analyzed.

Example C-7. operator() for charge division

```cpp
Bool_t
MyEp::operator()(const Address_t pEvent,
                 CEvent& rEvent,
                 CAnalyzer& rAnalyzer,
                 CBufferDecoder& rDecoder)
{
  if (m_canCompute) {
    UInt_t eventSize = rEvent.size();
```
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

if((eventSize < m_leftId) && (eventSize < m_rightId)) {
  if (rEvent[m_leftId].isValid() && rEvent[m_rightId].isValid()) {
    double sum = rEvent[m_leftId] + rEvent[m_rightId];
    double dif = rEvent[m_leftId] - rEvent[m_rightId];
    if (sum != 0.0 ) {
      rEvent[m_positionId] = dif/sum;
    }
  }
  return kfTRUE;
}

1. The size member function of the CEvent returns the number of elements currently in rEvent. There is no point in continuing the computation if the event is not big enough to have the two parameters we depend on. If necessary, rEvent will automatically expand to hold our result parameter.

2. Each element of the rEvent object is a CValidValue. The isValid member function returns true if the element has been assigned a value this event. We can only compute the charge division if that is the case.

3. Computes the position parameter.

4. Regardless, if the event processor does not return kfTRUE, the event processing pipeline aborts and the event will not be histogrammed.

C.2. TreeParameter

The treeparameter package provides assistance to those writing event processors. It allows you to easily bind unpacked parameters to objects that can be treated like variables during the computations performed in an event processor. This has the effect of insulating you from the exact choice of parameter number used by the other event processing pipeline elements. The tree parameter package was originally conceived of and written by Daniel Bazin of the NSCL.

The treeparameter package also supports a similar binding between variable like objects and Tcl variables. This binding makes it easy to 'steer' the computation of your event processors by adjusting the values of Tcl variables.

To understand this section you need to know that C++ allows one to define classes (and hence objects) that, for all intents and purposes are not distinguishable from variables. These objects implement
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

assignment as well as all of the normal arithmetic operations. The magic of the treeparameter package is
accomplished by defining such variable-like like objects.

The remainder of this section describes tree parameter objects, how to create them and how to use them.
This is followed by a discussion of treevariable objects and how to use them. The next full section of this
appendix pulls everything together through a worked example.

C.2.1. TreeParameter objects

Previously, we gave an example that showed how to calculate a position using charge division. The
example required a lot of interaction with the SpecTcl API to locate parameters and to create the new
output parameter.

Variables of the type CTreeParameter act as ordinary variables, but serve as proxies for specific
elements of the rEvent array. They are constructed in a way that connects them with the desired
parameter given a parameter name. If the desired parameter does not exist, the CTreeParameter will
create it. Thus tree parameters can be either inputs or output of computations.

The full interface to the CTreeParameter class is in the file CTreeParameter.h. Note that usually
this file is included via the convenience header TreeParameter.h.

A related class, CTreeParameterArray allows you to create arrays of parameters. Arrays of
parameters have are given a basename and index. A name for a tree parameter array element might, for
example be crdc.pads.12 where the basename is crdc.pads and the index is 12.

The code fragments below show an event processor class that uses tree parameters to access its input and
output parameters.

Example C-8. Using tree parameters in an event processor -- header.

```cpp
#include <TreeParameter.h>  
#include <string> 

class MyEp : public CEventProcessor 
{ 
private:
    CTreeParameter m_left; 
    CTreeParameter m_right; 
    CTreeParameter m_position; 

public:
    MyEp(std::string left,
```
std::string right,
std::string position); ➊

Bool_t operator() (const Address_t pEvent,
  CEvent& rEvent,
  CAnalyzer& rEvent,
  CBufferDecoder& rDecoder); ➋

➊ Includes the tree parameter header file. This includes all of the other headers that are used by the tree parameter package.

➋ The C++ Standard library defines a \texttt{string} that is used to hold character strings. This includes the header for that file. Note that the \texttt{string} class lives in the \texttt{std namespace}. More about that later.

➌ These member data are of type \texttt{CTreeParameter} they will be used to access the input and output parameters of our computation.

➍ This illustrates a special function called a \texttt{constructor}. Many classes require that objects be initialized in a specific way. The constructor function is the way this is done. A constructor is called when an object is created. In this case we will use our constructor to ensure that the tree parameter member data in turn is correctly constructed.

The thing that makes a tree parameter so useful is that it acts almost exactly like a double precision floating point number, but it is \textit{linked} to the SpecTcl parameter (rEvent element) that has the same name as the tree parameter. Just like elements of rEvent, tree parameters have an isValid member function that allows you to determine if event processors in prior stages of the analysis pipeline have assigned a value to the parameter.

Next let’s look at the implementation file for this event processor (the .cpp file). For the sake of brevity, we will only look at the segments of this file that implement the constructor and the function call operator.

\textbf{Example C-9. Tree parameter event processor constructor}

MyEp::MyEp(string left, string right, string position) :
  m_left(left, 0.0, 4095.0, "channels"),
  m_right(right, 0.0, 4095.0, "channels"),
  m_position(position, 0.0, 10.0, "mm")
{
}

➊ The colon on the end of this declaration is allowed only for constructors and indicates to the compiler that the declaration will be followed by a list of \textit{initializers}. An initializer is used to construct member variables that, themselves have non-trivial constructors. Each element of the
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

A comma separated list that follows causes the associated tree parameter to be initialized via the CTreeParameter constructor that matches this sort of parameter list.

Each of the elements of this comma separated list initializes (constructs) one of the Tree Parameter member variables. The syntax of an initializer is the name of the member variable used as a function whose actual arguments are arguments of that member’s class constructor.

The parameters for each constructor used here are, in order:

**Name:** : name
**Type:** : std::string
**Description:**: Name of the SpecTcl parameter to which this CTreeParameter will be bound. If, at the time the parameters are bound, a SpecTcl parameter by that name does not yet exist, one is created. This binding happens after all of the initialization scripts are sourced by SpecTcl. So an additional power of tree parameters is that in addition to referring to existing parameters, they can be used to create a new parameter.

**Name:** : low
**Type:** : double
**Description:**: Low limit of the range of this parameter. The SpecTcl folder GUI uses this information to suggest the lower limits of axes of spectra that depend on this parameter.

**Name:** : high
**Type:** : double
**Description:**: High limit of the range of this parameter. The SpecTcl folder GUI uses this information to suggest an upper limit for axes of spectra that depend on this parameter.

**Name:** : units
**Type:** : std::string
**Description:**: Units of measure of the parameter. This is used as in the axis labels of spectra that depend on this parameter.

Note that the units of measure for the position are listed as **mm**. In the next section, TreeVariable objects we will see how to calibrate the parameter to millimeters.

Now let’s look at the implementation of the function call operator. Note that since tree parameters are aliases for elements of the rEvent array, we don’t actually need to reference any of the parameter of the function call operator.

**Example C-10. Tree parameter version of function call operator.**

```c++
Bool_t
MyEp::operator()(const Address_t pEvent,
        CEvent& rEvent,
```
CAnalyzer& rAnalyzer,
CBufferDecoder& rDecoder)
{
    if (m_left.isValid() && m_right.isValid()) {
        double sum = m_left + m_right;
        double dif = m_left - m_right;

        if (sum != 0) {
            m_position = 10.0 * dif / sum + 5.0;  
➊
        }
    }
    return kfTRUE;
}

The implementation is straightforward. Comparing this function with the non tree version we see that rEvent element references have simply been replaced by tree parameter variables bound to those elements.

There is one lie. The conversion of the position to millimeters is probably not even close to correct. We’ll see how to fix that in the next section, which describes tree variables.

The power of tree variables is that they have hidden all of the interactions with SpecTcl’s API to figure out which element of rEvent held our parameters. Tree parameters have also allowed us to use sensible variable names instead of rEvent indices. Less obvious, is the fact that for complex event structures you can crate structs of tree parameters that allow you to organize the event parameters into a hierarchy that more naturally reflects the structure of the data than rEvent indices do.

C.2.2. TreeVariable objects

Tree variable objects allow you to create objects that act very much like a floating point variables but are, in fact, bound to Tcl variables. The overhead of using a treevariable is quite small as Tcl supports directly linking a C/C++ variable to a Tcl variable, and this is exploited by the CTTreeVariable class.

In the previous section, we implemented a compiled charge division computed parameter using tree parameters. We left the calibration of the position to millimeters as hard coded constants. In reality, we normally want to supply calibration constants that are changeable. Tree variables provide one way to do this that does not require us to recompile SpecTcl each time we need to change the calibration.

We will assume that the calibration is linear for simplicity. We will therefore add two Tree parameters to our event processor, a scale factor and an offset. This makes the header look like this:
Example C-11. Adding scale factor and offset to the tree event processor.

```cpp
...
class MyEp : public CEventProcessor
{
private:
  CTreeParameter m_left;
  CTreeParameter m_right;
  CTreeParameter m_position;
  CTreeVariable m_scaleFactor;  ➊
  CTreeVariable m_offset;

public:
  MyEp(std::string left, std::string right, std::string position,
       std::string scaleName, std::string offsetName);  ➋
...

 ➊ This section declares two tree variables as member data for our class. m_scaleFactor will be the position calibration scale factor while m_offset will be the calibration offset.

 ➋ The constructor will initialize these tree variables by binding them to specific named Tcl variables. The two parameters scaleName and offsetName will provide the names of these parameters.

Naturally this means that the constructor for the event processor in MySpecTclApp.cpp must be modified to pass these parameters. This is not described in this manual, but left as an exercise for the reader.

Let's look at how tree variable member data are initialized in the implementation of the constructor:

Example C-12. Treevariable event processor constructor implementation

```cpp
...
MyEp::MyEp(string left, string right, string position,
            string scaleName, string offsetName) :
  m_left(left, 0.0, 4095.0, "channels"),
  m_right(right, 0.0, 4095.0, "channels"),
  m_position(position, 0.0, 10.0, "mm"),
  m_scaleFactor(scaleName, 4095.0, "mm"),  ➊
  m_offset(offsetName, 2048.0, "mm")
{
}
}```
We have added initializers for the two tree variables. The parameters for these constructors is:

- **Name:** name
  - **Type:** std::string
  - **Description:** Name of the Tcl variable to which this tree parameter will be bound.

- **Name:** value
  - **Type:** double
  - **Description:** The initial value that will be assigned to the Tcl variable at binding time. Initial values assigned match those of the previous section.

- **Name:** units
  - **Type:** std::string
  - **Description:** Units of measure of the variable, if appropriate. For unit-less constants, use an empty string ("""). Since we are scaling a unit-less value, it is appropriate for both of these variables to be in mm. The SpecTcl folder GUI provides a folder in which tree parameters are listed. The current value of each parameter along with its unit of measure is displayed. Double-clicking a variable allows you to modify its value and, if appropriate, its units.

These tree parameters allow us to modify the computation of the position as follows:

**Example C-13. Using tree variables in the position computation**

```cpp
Bool_t
MyEp::operator()(const Address_t pEvent,
     CEvent& rEvent,
     CAnalyzer& rAnalyzer,
     CBufferDecoder& rDecoder)
{
    if (m_left.isValid() && m_right.isValid() ) {
        double sum = m_left + m_right;
        double dif = m_left - m_right;

        if (sum != 0) {
            m_position = m_scaleFactor * dif/sum + m_offset; ➊
        }
    }
    return kfTRUE;
}
```

Note how tree variables can just be used directly in computations as if they were double variables.

The power of tree variables is that they allow us to steer our computations using Tcl variables with very little additional programming overhead. Since tree variables integrate with the folder GUI, there’s no
need to build special GUIs to control these parameters (although you are certainly free to do so if the folder GUI does not provide the interface you’d like).

C.3. Source code for the examples.

This section provides the complete source code for the files described in previous sections of this appendix.

C.4. Non-Tree Parameter code

Example C-14. MySpecTclApp.h

```c++
// Class: CMyspecTclApp //ANSI C++
// File: MySpecTclApp.h
/*
The user creates this subclass and fills in the appropriate overrides for any additions they want to make. The class is a self contained example which registers two event processors. One which unpacks a simple fixed length event and another which produces a pseudo parameter from the sum of the first two parameters in an event.
*/
// Author:
// Ron Fox
// NSCL
// Michigan State University
// East Lansing, MI 48824-1321
// mailto:fox@nscl.msu.edu
//
// Copyright
#ifndef __MYSPECTCLAPP_H //Required for current class
#define __MYSPECTCLAPP_H

// Include files:
// Required for base classes
#ifndef __TCLGRAMMERAPP_H //CTclGrammerApp
#include "TclGrammerApp.h"
#endif

class CMyspecTclApp : public CTclGrammerApp { public:
// Constructors:
CMyspecTclApp(); //Default constructor alternative to compiler provided default constructor
~CMyspecTclApp(); //Destructor - Delete any pointer data members that used new in constructor
//Destructor should be virtual if and only if class contains at least one virtual function
//Objects destroyed in the reverse order of the construction order
```

167
private:
CMySpecTclApp(const CMySpecTclApp& aCMySpecTclApp); // Copy Constructor.

// Operators:
CMySpecTclApp& operator=(const CMySpecTclApp& aCMySpecTclApp);
int operator==(const CMySpecTclApp& aCMySpecTclApp) const;

// Class operations:
public:
virtual void BindTCLVariables(CTCLInterpreter& rInterp);
virtual void SourceLimitScripts(CTCLInterpreter& rInterpreter);
virtual void SetLimits();
virtual void CreateHistogrammer();
virtual void SelectDisplayer(UInt_t nDisplaySize, CHistogrammer& rHistogrammer);
virtual void CreateHistogrammer();
virtual void SourceLimitScripts(CTCLInterpreter& rInterpreter);
virtual void CreateAnalyzer();
virtual void SetupRunControl();
virtual void SourceFunctionalScripts(CTCLInterpreter& rInterp);
virtual int operator()();
};
#endif

Example C-15. MySpecTclApp.cpp

#include <config.h>
#include "MySpecTclApp.h"
#include "EventProcessor.h"
#include "TCLAnalyzer.h"
#include <Event.h>
#include <TreeParameter.h>

#include "MyEp.h"

#ifdef HAVE_STD_NAMESPACE
using namespace std;
#endif

// Local Class definitions:

// This is a sample tree parameter event structure:
// It defines an array of 10 raw parameters that will
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

// be unpacked from the data and a weighted sum
// that will be computed.

typedef
struct {
    CTreeParameterArray& raw;
    CTreeParameter& sum;
} MyEvent;

// Having created the struct we must make an instance
// that constructs the appropriate objects:

MyEvent event = {
    *(new CTreeParameterArray("event.raw", "channels", 10, 0)),
    *(new CTreeParameter("event.sum", "arbitrary"))
};

// Here’s a sample tree variable structure
// that defines the weights for the weighted
// sum so that they can be varied from the command line:
// An array is also declared for testing purposes but not used.

typedef
struct {
    CTreeVariable& w1;
    CTreeVariable& w2;
    CTreeVariableArray& unused;
} MyParameters;

// Similarly, having declared the structure, we must define
// it and construct its elements

MyParameters vars = {
    *(new CTreeVariable("vars.w1", 1.0, "arb/chan")),
    *(new CTreeVariable("vars.w2", 1.0, "arb/chan")),
    *(new CTreeVariableArray("vars.unused", 0.0, "furl/fort", 10, 0))
};

// CFixedEventUnpacker - Unpacks a fixed format event into
// a sequential set of parameters.

class CFixedEventUnpacker : public CEventProcessor
{
public:
    virtual Bool_t operator()(const Address_t pEvent, CEvent& rEvent, CAnalyzer& rAnalyzer, CBufferDecoder& rDecoder);
};

Bool_t
CFixedEventUnpacker::operator()(const Address_t pEvent,
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

```cpp
CEvent& rEvent,
CAnalyzer& rAnalyzer,
CBufferDecoder& rDecoder)
{
    // This sample unpacker unpacks a fixed length event which is
    // preceded by a word count.
    TranslatorPointer<UShort_t> p(* (rDecoder.getBufferTranslator()), pEvent);
    CTclAnalyzer& rAna((CTclAnalyzer&)rAnalyzer);
    UShort_t nWords = *p++;
    Int_t  i    = 1;
    // At least one member of the pipeline must tell the analyzer how
    // many bytes were in the raw event so it knows where to find the
    // next event.
    rAna.SetEventSize(nWords*sizeof(UShort_t)); // Set event size.
    nWords--; // The word count is self inclusive.
    int param = 0; // No more than 10 parameters.
    while(nWords && (param < 10)) { // Put parameters in the event starting at 1.
        event.raw[param] = *p++;
        param++;
    }
    return kfTRUE; // kfFALSE would abort pipeline.
}

// CAddFirst2 - Sample unpacker which adds a pair of unpacked parameters
// together to get a new parameter.

class CAddFirst2 : public CEventProcessor
{
public:
    virtual Bool_t operator()(const Address_t pEvent,
       CEvent& rEvent,
       CAnalyzer& rAnalyzer,
       CBufferDecoder& rDecoder);
};

Bool_t CAddFirst2::operator()(const Address_t pEvent,
       CEvent& rEvent,
       CAnalyzer& rAnalyzer,
       CBufferDecoder& rDecoder)
{
    event.sum = event.raw[0]*vars.w1 + event.raw[1]*vars.w2;
    return kfTRUE;
}
```
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

// Instantiate the unpackers we’ll use.

static CFixedEventUnpacker Stage1;
static CAddFirst2 Stage2;
static MyEp myEventProcessor;

// CFortranUnpacker:
// This sample unpacker is a bridge between the C++ SpecTcl
// and a FORTRAN unpacking/analysis package.
// The raw event is passed as a parameter to the FORTRAN function
// f77unpacker. This function has the signature:
// LOGICAL F77UNPACKER(EVENT)
// INTEGER*2 EVENT(*)
// As with all event processors, this function is supposed to
// return .TRUE. if processing continues or .FALSE. to abort event processing.
// Unpacked events are returned to SpecTcl via a Common block declared
// as follows:
// ! generated by the unpacker:
// COMMON/F77PARAMS/NOFFSET, NUSED, PARAMETERS(F77NPARAMS),
// 1 FPARAMETERS(F77NPARAMS)
// INTEGER*4 PARAMETERS
// LOGICAL FPARAMETERS
// At compile time, define: WITHF77UNPACKER
// and F77NPARAMS to be the maximum number of parameters the Fortran
// unpacker will fill in.
// In the unpacker, set FPARAMETERS(i) if i has been unpacked from
// the event.
// PARAMETERS are copied from 1 -> NUSED into parameters numbered
// NOFFSET -> NUSED+NOFFSET
// NOFFSET starts from zero.
// The fortran program should be compiled:
// cpp -DF77NPARAMS=nnnn yourprog.f > yourprog.for
// f77 -c yourprog.for
// rm yourprog.for
// Assuming you’ve written the file yourprog.f

#ifdef WITHF77UNPACKER

struct {
    int nOffset;
    int nUsed;
    int nParameters[F77NPARAMS]; // Fortran will extend this appropriately.
#endif

171
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

```c
int fParameters[F77NPARAMS];
} f77params_;

extern "C" Bool_t f77unpacker_(const Address_t pEvent);

class CFortranUnpacker : public CEventProcessor
{
 public:
     virtual Bool_t operator()(const Address_t pEvent, CEvent& rEvent,
         CAnalyzer& rAnalyzer, CBufferDecoder& rDecoder);

 Bool_t
 CFortranUnpacker::operator()(const Address_t pEvent,
         CEvent& rEvent,
         CAnalyzer& rAnalyzer,
         CBufferDecoder& rDecoder)
{
    Bool_t result = f77unpacker_(pEvent);
    if(result) {
        int dest = f77params_.nOffset;
        for(int i = 0; i < f77params_.nUsed; i++) {
            if(f77params_.fParameters[i])
                rEvent[dest] = f77params_.nParameters[i];
                dest++;
        }
        USHort_t* pSize = (USHort_t*)pEvent;
        CTclAnalyzer& rAna((CTclAnalyzer&)rAnalyzer);
        rAna.SetEventSize( (*pSize) * sizeof(UShort_t));
    }
    return result;
}

CFortranUnpacker legacyunpacker;
```

Set up an analysis pipeline. This function is required and must be filled in by the SpecTcl user. Pipeline elements are objects which are members of classes derived from the CEventProcessor class. They should be added to the Analyzer’s event processing pipeline by calling RegisterEventProcessor (non virtual base class member).

The sample implementation in this file produces a two step pipeline. The first step decodes a fixed length event into the CEvent array. The first parameter is put into index 1 and so on. The second step produces a compiled pseudo parameter by adding event array
elements 1 and 2 and putting the result into element 0.

*/

void CMySpecTclApp::CreateAnalysisPipeline(CAnalyzer& rAnalyzer)
{
    #ifdef WITHF77UNPACKER
        RegisterEventProcessor(legacyunpacker);
    #endif
        RegisterEventProcessor(Stage1, "Raw");
        RegisterEventProcessor(Stage2, "Computed");
        RegisterEventProcessor(myEventProcessor, "stupid-processor");
    }

    // Constructors, destructors and other replacements for compiler cannonicals:

    CMySpecTclApp::CMySpecTclApp ()
    {
    }
    // Destructor:

    CMySpecTclApp::~CMySpecTclApp ()
    {
    }

    // Functions for class CMySpecTclApp

    // Function:
    //      void BindTCLVariables(CTCLInterpreter& rInterp)
    //   Operation Type:
    //      override
    /*
    Purpose:

    Add code to this function to bind any TCL variable to
    the SpecTcl interpreter. Note that at this time,
    variables have not yet been necessarily created so you
    can do Set but not necessarily Get operations.

    */
    void CMySpecTclApp::BindTCLVariables(CTCLInterpreter& rInterp)
    {

}
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

CTclGrammerApp::BindTCLVariables(rInterp);

// Function:
// void SourceLimitScripts(CTCLInterpreter& rInterpreter)
// Operation Type:
// Override
/*/ Purpose:

Add code here to source additional variable setting scripts. At this time the entire SpecTcl/Tk infrastructure is not yet set up. Scripts run at this stage can only run basic Tcl/Tk commands, and not SpecTcl extensions. Typically, this might be used to set a bunch of initial values for variables which were bound in BindTCLVariables.

*/
void CMySpecTclApp::SourceLimitScripts(CTCLInterpreter& rInterpreter)
{
  CTclGrammerApp::SourceLimitScripts(rInterpreter);
}

// Function:
// void SetLimits()
// Operation Type:
// override
/*/ Purpose:

Called after BindVariables and SourceLimitScripts. This function can be used to fetch values of bound Tcl variables which were modified/set by the limit scripts to update program default values.

*/
void CMySpecTclApp::SetLimits()
{
  CTclGrammerApp::SetLimits();
}

// Function:
// void CreateHistogrammer()
// Operation Type:
// Override
/*/ Purpose:

Creates the histogramming data sink. If you want to override this in general you probably won’t make use of the actual base class function. You might, however extend this by defining a base set of parameters and histograms from within the program.
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

`void CMySpecTclApp::CreateHistogrammer()
{ CTclGrammerApp::CreateHistogrammer(); }

// Function:
// void SelectDisplayer(UInt_t nDisplaySize, CHistogrammer& rHistogrammer)
// Operation Type:
// Override.
/*@ Purpose:
Select a displayer object and link it to the histogrammer. The default code will link Xamine to
the displayer, and set up the Xamine event handler to deal with gate objects accepted by Xamine interaction.
*/
`void CMySpecTclApp::SelectDisplayer(UInt_t nDisplaySize, CHistogrammer& rHistogrammer)
{ CTclGrammerApp::SelectDisplayer(nDisplaySize, rHistogrammer); }

// Function:
// void SetupTestDataSource()
// Operation Type:
// Override
/*@ Purpose:
Allows you to set up a test data source. At present, SpecTcl must have a data source of some sort connected to it... The default test data source produces a fixed length event where all parameters are selected from a gaussian distribution. If you can figure out how to do it, you can setup your own data source... as long as you don’t start analysis, the default one is harmless.
*/
`void CMySpecTclApp::SetupTestDataSource()
{ CTclGrammerApp::SetupTestDataSource(); }

// Function:
// void CreateAnalyzer(CEventSink* pSink)
// Operation Type:
// Override
/*@ Purpose:
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

Creates an analyzer. The Analyzer is connected to the data source which supplies buffers. Connected to the analyzer is a buffer decoder and an event unpacker. The event unpacker is the main experiment dependent chunk of code, not the analyzer. The analyzer constructed by the base class is a CTclAnalyzer instance. This is an analyzer which maintains statistics about itself in Tcl Variables.

```c
*/
void
CMySpecTclApp::CreateAnalyzer(CEventSink* pSink)
{ CTclGrammerApp::CreateAnalyzer(pSink);
}
```

// Function:
// void SelectDecoder(CAnalyzer& rAnalyzer)
// Operation Type:
//   Override
/*
Purpose:

Selects a decoder and attaches it to the analyzer. A decoder is responsible for knowing the overall structure of a buffer produced by a data analysis system. The default code constructs a CNSCLBufferDecoder object which knows the format of NSCL buffers.

```c
*/
void
CMySpecTclApp::SelectDecoder(CAnalyzer& rAnalyzer)
{ CTclGrammerApp::SelectDecoder(rAnalyzer);
}
```

// Function:
// void AddCommands(CTCLInterpreter& rInterp)
// Operation Type:
//   Override
/*
Purpose:

This function adds commands to extend Tcl/Tk/SpecTcl. The base class function registers the standard SpecTcl command packages. Your commands can be registered at this point. Do not remove the sample code or the SpecTcl commands will not get registered.

```c
*/
void
CMySpecTclApp::AddCommands(CTCLInterpreter& rInterp)
{ CTclGrammerApp::AddCommands(rInterp);
}
// Function:
// void SetupRunControl()
// Operation Type:
// Override.
/*
Purpose:
Sets up the Run control object. The run control object is responsible for interacting with the underlying operating system and programming framework to route data from the data source to the SpecTcl analyzer. The base class object instantiates a CTKRunControl object. This object uses fd waiting within the Tcl/TK event processing loop framework to dispatch buffers for processing as they become available.
*/
void CMySpecTclApp::SetupRunControl()
{
    CTclGrammerApp::SetupRunControl();
}

// Function:
// void SourceFunctionalScripts(CTCLInterpreter& rInterp)
// Operation Type:
// Override
/*
Purpose:
This function allows the user to source scripts which have access to the full Tcl/Tk/SpecTcl command set along with whatever extensions have been added by the user in AddCommands.
*/
void CMySpecTclApp::SourceFunctionalScripts(CTCLInterpreter& rInterp)
{
    CTclGrammerApp::SourceFunctionalScripts(rInterp);
}

// Function:
// int operator()()
// Operation Type:
// Override.
/*
Purpose:
Entered at Tcl/Tk initialization time (think of this as the entry point of the SpecTcl program). The base class default implementation calls the member functions of this class in an appropriate order. It’s possible for the user to extend this functionality by adding code to this function.
*/
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

```cpp
int CMySpecTclApp::operator()()
{
    return CTclGrammerApp::operator()();
}

CMySpecTclApp myApp;
CTclGrammerApp& app(myApp); // Create an instance of me.
CTCLApplication* gpTCLApplication=&app; // Findable by the Tcl/tk framework.

Example C-16. MyEp.h

#ifndef __MYEP_H
#define __MYEP_H

#include <EventProcessor.h>

class MyEp : public CEventProcessor
{
private:
    UInt_t m_positionId;
    UInt_t m_leftId;
    UInt_t m_rightId;
    bool m_canCompute;

public:
    virtual Bool_t OnAttach(CAnalyzer& rAnalyzer); // Called on registration.
    virtual Bool_t OnBegin(CAnalyzer& rAnalyzer,
                            CBufferDecoder& rDecoder); // Begin Run.
    virtual Bool_t OnEnd(CAnalyzer& rAnalyzer,
                          CBufferDecoder& rBuffer); // End Run.
    virtual Bool_t OnPause(CAnalyzer& rAnalyzer,
                           CBufferDecoder& rDecoder); // Pause Run.
    virtual Bool_t OnResume(CAnalyzer& rAnalyzer,
                           CBufferDecoder& rDecoder); // Resume Run.
    virtual Bool_t OnOther(UInt_t nType,
                          CAnalyzer& rAnalyzer,
                          CBufferDecoder& rDecoder); // Unrecognized buftype.
    virtual Bool_t OnEventSourceOpen(std::string name);
    virtual Bool_t OnEventSourceEOF();
    virtual Bool_t operator()(const Address_t pEvent,
                               CEvent& rEvent,
                               CAnalyzer& rAnalyzer,
                               CBufferDecoder& rDecoder); // Physics Event.

```
Example C-17. MyEp.cpp

```c++
#include <config.h>
#include "MyEp.h"
#include <Analyzer.h>
#include <BufferDecoder.h>
#include <Event.h>
#include <SpecTcl.h>

using namespace std;

Bool_t
MyEp::OnAttach(CAnalyzer& rAnalyzer)
{
    SpecTcl* api = SpecTcl::getInstance();
    m_positionId = api->AssignParameterId();
    try {
        api->AddParameter("x.position", m_positionId, "channels");
    } catch (...) {
        m_positionId = 0xffffffff;
        return kfFALSE;
    }
    return kfTRUE;
}

Bool_t
MyEp::OnBegin(CAnalyzer& rAnalyzer, 
               CBufferDecoder& rDecoder)
{
    SpecTcl* api = SpecTcl::getInstance();
    CParameter* pLeft = api->FindParameter("left");
    CParameter* pRight = api->FindParameter("right");
    if (pLeft && pRight) {
        m_leftId = pLeft->getNumber();
        m_rightId = pRight->getNumber();
        m_canCompute = m_positionId != 0xffffffff;
    } else {
        m_canCompute = false;
    }
    return kfTRUE;
}
```
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

```c++
Bool_t MyEp::OnEnd(CAnalyzer& rAnalyzer,
                   CBufferDecoder& rBuffer)
{
    return kfTRUE;
}

Bool_t MyEp::OnPause(CAnalyzer& rAnalyzer,
                      CBufferDecoder& rDecoder)
{
    return kfTRUE;
}

Bool_t MyEp::OnResume(CAnalyzer& rAnalyzer,
                       CBufferDecoder& rDecoder)
{
    return kfTRUE;
}

Bool_t MyEp::OnOther(UInt_t nType,
                     CAnalyzer& rAnalyzer,
                     CBufferDecoder& rDecoder)
{
    return kfTRUE;
}

Bool_t MyEp::OnEventSourceOpen(STD(string) name)
{
    return kfTRUE;
}

Bool_t MyEp::OnEventSourceEOF()
{
    return kfTRUE;
}

Bool_t MyEp::operator()(const Address_t pEvent,
                         CEvent& rEvent,
                         CAnalyzer& rAnalyzer,
                         CBufferDecoder& rDecoder)
{

    UInt_t eventSize = rEvent.size();
    if((eventSize < m_leftId) && (eventSize < m_rightId)) {
        if (rEvent[m_leftId].isValid() && rEvent[m_rightId].isValid()) {
            double sum = rEvent[m_leftId] + rEvent[m_rightId];
            double dif = rEvent[m_leftId] - rEvent[m_rightId];

            if (sum != 0.0) {
                rEvent[m_positionId] = dif/sum;
            }
        }
    }
}
```
Example C-18. Makefile

INSTDIR=/usr/opt/spectcl/3.3
# Skeleton makefile for 3.3
include $(INSTDIR)/etc/SpecTcl_Makefile.include

# If you have any switches that need to be added to the default c++ compilation
# rules, add them to the definition below:
USERCXXFLAGS=

# If you have any switches you need to add to the default c compilation rules,
# add them to the definition below:
USERCCFLAGS=$(USERCXXFLAGS)

# If you have any switches you need to add to the link add them below:
USERLDFLAGS=

# Appends your objects to the definitions below:
#
OBJECTS=MySpecTclApp.o MyEp.o

# Finally the makefile targets.
#
SpecTcl: $(OBJECTS)
$(CXXLD) -o SpecTcl $(OBJECTS) $(USERLDFLAGS) \ $(LDFLAGS)

clean:
rm -f $(OBJECTS) SpecTcl
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

depend:
makedepend $(USERCXXFLAGS) *.cpp *.c

help:
echo "make" - Build customized SpecTcl"
echo "make clean" - Remove objects from previous builds"
echo "make depend" - Add dependencies to the Makefile. "
# DO NOT DELETE

MySpecTclApp.o: MySpecTclApp.h MyEp.h

C.5. TreeParameter version

Example C-19. MySpecTclApp.h

// Class: CMySpecTclApp //ANSI C++
// File: MySpecTclApp.h
/*
The user creates this subclass and fills in the appropriate overrides for any
additions they want to make. The class is a self contained example which
registers two event processors. One which unpacks a simple fixed length
event and another which produces a pseudo parameter from the sum of
the first two parameters in an event.
*/
// Author:
// Ron Fox
// NSCL
// Michigan State University
// East Lansing, MI 48824-1321
// mailto:fox@nscl.msu.edu
//
// Copyright
#endif __MYSPECTCLAPP_H //Required for current class
#define __MYSPECTCLAPP_H

// Include files:
// Required for base classes
#ifndef __TCLGRAMMERAPP_H //CTclGrammerApp
#include "TclGrammerApp.h"
#endif

class CMySpecTclApp : public CTclGrammerApp {
public:
   // Constructors:
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

CMYSpecTclApp(); //Default constructor alternative to compiler provided default constructor.
~CMySpecTclApp(); //Destructor - Delete any pointer data members that used new in constructor.
//Destructor should be virtual if and only if class contains at least one virtual function
//Objects destroyed in the reverse order of the construction order
private:
CMySpecTclApp(const CMySpecTclApp& aCMySpecTclApp ); // Copy Constructor.

// Operators:
CMySpecTclApp& operator=(const CMySpecTclApp& aCMySpecTclApp);
int operator==(const CMySpecTclApp& aCMySpecTclApp) const;

// Class operations:
public:
virtual void BindTCLVariables(CTCLInterpreter& rInterp);
virtual void SourceLimitScripts(CTCLInterpreter& rInterpreter);
virtual void SetLimits();
virtual void CreateHistogrammer();
virtual void SelectDisplaySize(UInt_t nDisplaySize, CHistogrammer& rHistogrammer);
virtual void SetupTestDataSource();
virtual void CreateAnalyzer(CEventSink* pSink);
virtual void SelectDecoder(CAnalyzer& rAnalyzer);
virtual void CreateAnalysisPipeline(CAnalyzer& rAnalyzer);
virtual void AddCommands(CTCLInterpreter& rInterp);
virtual void SetupRunControl();
virtual void SourceFunctionalScripts(CTCLInterpreter& rInterp);
virtual int operator()();
};
#endif

Example C-20. MySpecTclApp.cpp

static const char* Copyright = "(C) Copyright Michigan State University 2008, All rights reserved.

// Class: CMySpecTclApp

///////////////////////////////////////////////////////////////////// FILE_NAME.cpp/////////////////////////////////////////////////////////////////////
#include <config.h>
#include "MySpecTclApp.h"
#include "EventProcessor.h"
#include "TCLAnalyzer.h"
#include <Event.h>
#include <TreeParameter.h>

#include "MyEp.h"

#ifdef HAVE_STD_NAMESPACE
using namespace std;
#endif

// Local Class definitions:

// This is a sample tree parameter event structure:
// It defines an array of 10 raw parameters that will
// be unpacked from the data and a weighted sum
// that will be computed.
//
typedef
struct {
    CTreeParameterArray& raw;
    CTreeParameter& sum;
} MyEvent;

// Having created the struct we must make an instance
// that constructs the appropriate objects:

MyEvent event = {
    *(new CTreeParameterArray("event.raw", "channels", 10, 0)),
    *(new CTreeParameter("event.sum", "arbitrary"))
};

// Here’s a sample tree variable structure
// that defines the weights for the weighted
// sum so that they can be varied from the command line:
// An array is also declared for testing purposes but not used.
//
typedef
struct {
    CTreeVariables w1;
    CTreeVariables w2;
    CTreeVariableArray unused;
} MyParameters;

// Similarly, having declared the structure, we must define
// it and construct its elements

MyParameters vars = {
    *(new CTreeVariable("vars.w1", 1.0, "arb/chan")),
    *(new CTreeVariable("vars.w2", 1.0, "arb/chan")),
    *(new CTreeVariableArray("vars.unused", 0.0, "furl/fort", 10, 0))
};

// CFixedEventUnpacker - Unpacks a fixed format event into
// a sequential set of parameters.
//
class CFixedEventUnpacker : public CEventProcessor
{
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

public:
    virtual Bool_t operator()(const Address_t pEvent,
        CEvent& rEvent,
        CAnalyzer& rAnalyzer,
        CBufferDecoder& rDecoder);
};

Bool_t
CFixedEventUnpacker::operator()(const Address_t pEvent,
    CEvent& rEvent,
    CAnalyzer& rAnalyzer,
    CBufferDecoder& rDecoder)
{
    // This sample unpacker unpacks a fixed length event which is
    // preceded by a word count.
    //
    TranslatorPointer<UShort_t> p(* (rDecoder.getBufferTranslator()), pEvent);
    CTclAnalyzer& rAna((CTclAnalyzer&)rAnalyzer);
    UShort_t nWords = *p++;
    Int_t i = 1;

    // At least one member of the pipeline must tell the analyzer how
    // many bytes were in the raw event so it knows where to find the
    // next event.
    rAna.SetEventSize(nWords* sizeof(UShort_t)); // Set event size.
    nWords--; // The word count is self inclusive.
    int param = 0; // No more than 10 parameters.

    while(nWords && (param < 10)) { // Put parameters in the event starting at 1.
        event.raw[param] = *p++;
        nWords--;
        param++;
    }
    return kfTRUE; // kfFALSE would abort pipeline.
}

// CAddFirst2 - Sample unpacker which adds a pair of unpacked parameters
// together to get a new parameter.

class CAddFirst2 : public CEventProcessor
{
public:
    virtual Bool_t operator()(const Address_t pEvent,
        CEvent& rEvent,
        CAnalyzer& rAnalyzer,
        CBufferDecoder& rDecoder);
};

185
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

```
Bool_t
CAddFirst2::operator()(const Address_t pEvent,
               CEvent& rEvent,
               CAnalyzer& rAnalyzer,
               CBufferDecoder& rDecoder)
{
    event.sum = event.raw[0] * vars.w1 + event.raw[1] * vars.w2;
    return kTRUE;
}
```

// Instantiate the unpackers we'll use.

```c
static CFixedEventUnpacker Stage1;
static CAddFirst2 Stage2;
static MyEp myEventProcessor("left", "right", "x.position");
```

// CFortranUnpacker:
// This sample unpacker is a bridge between the C++ SpecTcl
// and a FORTRAN unpacking/analysis package.
// The raw event is passed as a parameter to the FORTRAN function
// f77unpacker. This function has the signature:
//   LOGICAL F77UNPACKER(EVENT)
//   INTEGER*2 EVENT(*)
//   As with all event processors, this function is supposed to
//   return .TRUE. if processing continues or .FALSE. to abort event processing.
//   Unpacked events are returned to SpecTcl via a Common block declared
//   as follows:
//   ! generated by the unpacker:
//   COMMON/F77PARAMS/NOFFSET, NUSED, PARAMETERS(F77NPARAMS),
//       FPARAMETERS(F77NPARAMS)
//   INTEGER*4 PARAMETERS
//   LOGICAL   FPARAMETERS
//   At compile time, define: WITHF77UNPACKER
//   and F77NPARAMS to be the maximum number of parameters the Fortran
//   unpacker will fill in.
//   In the unpacker, set FPARAMETERS(i) if i has been unpacked from
//   the event.
//   PARAMETERS are copied from 1 -> NUSED into parameters numbered
//   NOFFSET -> NUSED+NOFFSET
//   NOFFSET starts from zero.
//   The fortran program should be compiled:
//   cpp -DF77NPARAMS=nnnn yourprog.f > yourprog.for
//   f77 -c yourprog.for
//   rm yourprog.for
```
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

// Assuming you’ve written the file yourprog.f

#ifdefined WITHF77UNPACKER

struct {
    int nOffset;
    int nUsed;
    int nParameters[F77NPARAMS]; // Fortran will extend this appropriately.
    int fParameters[F77NPARAMS];
} f77params_;

extern "C" Bool_t f77unpacker_(const Address_t pEvent);

class CFortranUnpacker : public CEventProcessor
{
public:
    virtual Bool_t operator()(const Address_t pEvent, CEvent& rEvent,
        CAnalyzer& rAnalyzer, CBufferDecoder& rDecoder);
};

Bool_t
CFortranUnpacker::operator()(const Address_t pEvent, CEvent& rEvent,
    CAnalyzer& rAnalyzer, CBufferDecoder& rDecoder)
{
    Bool_t result = f77unpacker_(pEvent);
    if(result) {
        int dest = f77params_.nOffset;
        for(int i = 0; i < f77params_.nUsed; i++) {
            if(f77params_.fParameters[i])
                rEvent[dest] = f77params_.nParameters[i];
            dest++;
        }
        UShort_t* pSize = (UShort_t*)pEvent;
        CTclAnalyzer& rAna((CTclAnalyzer&)rAnalyzer);
        rAna.SetEventSize( (*pSize) * sizeof(UShort_t));
    }
    return result;
}

CFortranUnpacker legacyunpacker;

#endif

// Function:
//   void CreateAnalysisPipeline(CAnalyzer& rAnalyzer)
// Operation Type:
//   Override
// Purpose:
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

Sets up an analysis pipeline. This function is required and must be filled in by the SpecTcl user. Pipeline elements are objects which are members of classes derived from the CEventProcessor class. They should be added to the Analyzer’s event processing pipeline by calling RegisterEventProcessor (non virtual base class member).

The sample implementation in this file produces a two step pipeline. The first step decodes a fixed length event into the CEvent array. The first parameter is put into index 1 and so on. The second step produces a compiled pseudo parameter by adding event array elements 1 and 2 and putting the result into element 0.

```c
void CMySpecTclApp::CreateAnalysisPipeline(CAnalyzer& rAnalyzer)
{
    #ifdef WITHF77UNPACKER
        RegisterEventProcessor(legacyunpacker);
    #endif

    RegisterEventProcessor(Stage1, "Raw");
    RegisterEventProcessor(Stage2, "Computed");
    RegisterEventProcessor(myEventProcessor, "stupid-processor");
}
```

// Constructors, destructors and other replacements for compiler cannonicals:

CMySpecTclApp::CMySpecTclApp ()
{
}

// Destructor:

CMySpecTclApp::~CMySpecTclApp ()
{
}

// Functions for class CMySpecTclApp

// Function:
// void BindTCLVariables(CTCLInterpreter& rInterp)
// Operation Type:
// override
// Purpose:

Add code to this function to bind any TCL variable to the SpecTcl interpreter. Note that at this time,
variables have not yet been necessarily created so you can do Set but not necessarily Get operations.

*/
void
CMysSpecTclApp::BindTCLVariables(CTCLInterpreter& rInterp)
{
  CTclGrammerApp::BindTCLVariables(rInterp);
}

// Function:
// void SourceLimitScripts(CTCLInterpreter& rInterpreter)
// Operation Type:
//    Override
/*
Purpose:
Add code here to source additional variable setting scripts. At this time the entire SpecTcl/Tk infrastructure is not yet set up. Scripts run at this stage can only run basic Tcl/Tk commands, and not SpecTcl extensions. Typically, this might be used to set a bunch of initial values for variables which were bound in BindTCLVariables.

*/
void
CMysSpecTclApp::SourceLimitScripts(CTCLInterpreter& rInterpreter)
{
  CTclGrammerApp::SourceLimitScripts(rInterpreter);
}

// Function:
// void SetLimits()
// Operation Type:
//    override
/*
Purpose:
Called after BindVariables and SourceLimitScripts. This function can be used to fetch values of bound Tcl variables which were modified/set by the limit scripts to update program default values.

*/
void
CMysSpecTclApp::SetLimits()
{
  CTclGrammerApp::SetLimits();
}

// Function:
// void CreateHistogrammer()
// Operation Type:
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

// Override
/
Purpose:

Creates the histogramming data sink. If you want to override this in general you probably won’t make use of the actual base class function. You might, however extend this by defining a base set of parameters and histograms from within the program.

*/
void
CMySpecTclApp::CreateHistogrammer()
{ CTclGrammerApp::CreateHistogrammer();
}

// Function:
// void SelectDisplayer(UInt_t nDisplaySize, CHistogrammer& rHistogrammer)
// Operation Type:
// Override.
/*
Purpose:

Select a displayer object and link it to the histogrammer. The default code will link Xamine to the displayer, and set up the Xamine event handler to deal with gate objects accepted by Xamine interaction.

*/
void
CMySpecTclApp::SelectDisplayer(UInt_t nDisplaySize, C Histogrammer& rHistogrammer)
{ CTclGrammerApp::SelectDisplayer(nDisplaySize, rHistogrammer);
}

// Function:
// void SetupTestDataSource()
// Operation Type:
// Override
/*
Purpose:

Allows you to set up a test data source. At present, SpecTcl must have a data source of some sort connected to it... The default test data source produces a fixed length event where all parameters are selected from a gaussian distribution. If you can figure out how to do it, you can setup your own data source... as long as you don’t start analysis, the default one is harmless.

*/
void
CMySpecTclApp::SetupTestDataSource()
{ CTclGrammerApp::SetupTestDataSource();

190
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

```c++
// Function:
// void CreateAnalyzer(CEventSink* pSink)
// Operation Type:
//     Override
/*
Purpose:

Creates an analyzer. The Analyzer is connected to the data
source which supplies buffers. Connected to the analyzer is a
buffer decoder and an event unpacker. The event unpacker is
the main experiment dependent chunk of code, not the analyzer.
The analyzer constructed by the base class is a CTclAnalyzer instance.
This is an analyzer which maintains statistics about itself in Tcl Variables.
*/
void
CMySpecTclApp::CreateAnalyzer(CEventSink* pSink)
{
    CTclGrammerApp::CreateAnalyzer(pSink);
}

// Function:
// void SelectDecoder(CAnalyzer& rAnalyzer)
// Operation Type:
//     Override
/*
Purpose:

Selects a decoder and attaches it to the analyzer.
A decoder is responsible for knowing the overall structure of
a buffer produced by a data analysis system. The default code
constructs a CNSCLBufferDecoder object which knows the format
of NSCL buffers.
*/
void
CMySpecTclApp::SelectDecoder(CAnalyzer& rAnalyzer)
{
    CTclGrammerApp::SelectDecoder(rAnalyzer);
}

// Function:
// void AddCommands(CTCLInterpreter& rInterp)
// Operation Type:
//     Override
/*
Purpose:

This function adds commands to extend Tcl/Tk/SpecTcl.
The base class function registers the standard SpecTcl command
packages. Your commands can be registered at this point.
*/
```
Do not remove the sample code or the SpecTcl commands will not get registered.

*/
void
CMySpecTclApp::AddCommands(CTCLInterpreter& rInterp)
{ CTclGrammerApp::AddCommands(rInterp);
}

// Function:
// void SetupRunControl()
// Operation Type:
// Override.
/*
Purpose:

Sets up the Run control object. The run control object is responsible for interacting with the underlying operating system and programming framework to route data from the data source to the SpecTcl analyzer. The base class object instantiates a CTKRunControl object. This object uses fd waiting within the Tcl/TK event processing loop framework to dispatch buffers for processing as they become available.

*/
void
CMySpecTclApp::SetupRunControl()
{ CTclGrammerApp::SetupRunControl();
}

// Function:
// void SourceFunctionalScripts(CTCLInterpreter& rInterp)
// Operation Type:
// Override
/*
Purpose:

This function allows the user to source scripts which have access to the full Tcl/Tk/SpecTcl command set along with whatever extensions have been added by the user in AddCommands.

*/
void
CMySpecTclApp::SourceFunctionalScripts(CTCLInterpreter& rInterp)
{ CTclGrammerApp::SourceFunctionalScripts(rInterp);
}
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

Purpose:

Entered at Tcl/Tk initialization time (think of this as the entry point of the SpecTcl program). The base class default implementation calls the member functions of this class in an appropriate order. It’s possible for the user to extend this functionality by adding code to this function.

*/
int CMySpecTclApp::operator()()
{
return CTclGrammerApp::operator()();
}

CMySpecTclApp myApp;
CTclGrammerApp& app(myApp); // Create an instance of me.
CTCLApplication* gpTCLApplication=&app; // Findable by the Tcl/tk framework.

Example C-21. MyEp.h

#ifndef __MYEP_H
#define __MYEP_H

#include <EventProcessor.h>
#include <TreeParameter.h>
#include <string>

class MyEp : public CEventProcessor
{
private:
    CTreeParameter m_left;
    CTreeParameter m_right;
    CTreeParameter m_position;

public:
    MyEp(std::string left, std::string right, std::string position);

    virtual Bool_t OnAttach(CAnalyzer& rAnalyzer); // Called on registration.
    virtual Bool_t OnBegin(CAnalyzer& rAnalyzer,
                              CBufferDecoder& rDecoder); // Begin Run.
    virtual Bool_t OnEnd(CAnalyzer& rAnalyzer,
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

```cpp
#include <config.h>
#include "MyEp.h"
#include <Analyzer.h>
#include <BufferDecoder.h>
#include <Event.h>

using namespace std;

MyEp::MyEp(string left, string right, string position) :
  m_left(left, 0.0, 4095.0, "channels"),
  m_right(right, 0.0, 4095.0, "channels"),
  m_position(position, 0.0, 10.0, "mm")
{
}

Bool_t
MyEp::OnAttach(CAnalyzer& rAnalyzer)
{
  return kfTRUE;
}

Bool_t
```
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

MyEp::OnBegin(CAnalyzer& rAnalyzer, 
               CBufferDecoder& rDecoder)
{
    return kfTRUE;
}

Bool_t
MyEp::OnEnd(CAnalyzer& rAnalyzer, 
             CBufferDecoder& rBuffer)
{
    return kfTRUE;
}

Bool_t
MyEp::OnPause(CAnalyzer& rAnalyzer, 
               CBufferDecoder& rDecoder)
{
    return kfTRUE;
}

Bool_t
MyEp::OnResume(CAnalyzer& rAnalyzer, 
                CBufferDecoder& rDecoder)
{
    return kfTRUE;
}

Bool_t
MyEp::OnOther(UInt_t nType, 
               CAnalyzer& rAnalyzer, 
               CBufferDecoder& rDecoder)
{
    return kfTRUE;
}

Bool_t
MyEp::OnEventSourceOpen(STD(string) name)
{
    return kfTRUE;
}

Bool_t
MyEp::OnEventSourceEOF()
{
    return kfTRUE;
}

Bool_t
MyEp::operator()(const Address_t pEvent, 
                  CEvent& rEvent, 
                  CAnalyzer& rAnalyzer, 
                  CBufferDecoder& rDecoder)
{
    if (m_left.isValid() && m_right.isValid()) {
        double sum = m_left + m_right;
        double dif = m_left - m_right;

        if (sum != 0) {
            
        }
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

```c
m_position = 10.0 * dif / sum + 5.0;
}
}
return kfTRUE;
}
```

**Example C-23. Makefile**

```makefile
INSTDIR=/usr/opt/spectcl/3.3
# Skeleton makefile for 3.3
include $(INSTDIR)/etc/SpecTcl_Makefile.include

# If you have any switches that need to be added to the default c++ compilation
# rules, add them to the definition below:
USERCXXFLAGS=

# If you have any switches you need to add to the default c compilation rules,
# add them to the definition below:
USERCCFLAGS=$(USERCXXFLAGS)

# If you have any switches you need to add to the link add them below:
USERLDFLAGS=

# Append your objects to the definitions below:

OBJECTS=MySpecTclApp.o MyEp.o

# Finally the makefile targets.

SpecTcl: $(OBJECTS)
$(CXXLD) -o SpecTcl $(OBJECTS) $(USERLDFLAGS) \ $(LDFLAGS)

clean:
rm -f $(OBJECTS) SpecTcl

depend:
makedepend $(USERCXXFLAGS) *.cpp *.c
```
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

help:
echo "make" - Build customized SpecTcl
echo "make clean" - Remove objects from previous builds
echo "make depend" - Add dependencies to the Makefile.
# DO NOT DELETE

MySpecTclApp.o: MySpecTclApp.h MyEp.h

C.6. TreeVariable version

Example C-24. MySpecTclApp.h

#ifndef __MYSPECTCLAPP_H //Required for current class
#define __MYSPECTCLAPP_H

// Include files:
// Required for base classes
#ifndef __TCLGRAMMERAPP_H //CTclGrammerApp
#include "TclGrammerApp.h"
#endif

class CMySpecTclApp : public CTclGrammerApp {
public:
    // Constructors:
    CMySpecTclApp(); //Default constructor alternative to compiler provided default constructor.
    ~CMySpecTclApp(); //Destructor - Delete any pointer data members that used new in constructors
    //Destructor should be virtual if and only if class contains at least one virtual function.
    //Objects destroyed in the reverse order of the construction order
private:
    CMySpecTclApp(const CMySpecTclApp& aCMySpecTclApp); // Copy Constructor.

    // Operators:
    CMySpecTclApps& operator=(const CMySpecTclApp& aCMySpecTclApp);
    int operator==(const CMySpecTclApp& aCMySpecTclApp) const;

    // Class operations:
public:
    virtual void BindTCLVariables(CTCLInterpreter& rInterp);
    virtual void SourceLimitScripts(CTCLInterpreter& rInterpreter);
    virtual void SetLimits();
    virtual void CreateHistogrammer();
    virtual void SelectDisplayer(UInt_t nDisplaySize, CHistogrammer& rHistogrammer);
    virtual void SetupTestDataSource();
    virtual void CreateAnalyzer(CEventSink* pSink);
virtual void SelectDecoder(CAnalyzer& rAnalyzer);
virtual void CreateAnalysisPipeline(CAnalyzer& rAnalyzer);
virtual void AddCommands(CTCLInterpreter& rInterp);
virtual void SetupRunControl();
virtual void SourceFunctionalScripts(CTCLInterpreter& rInterp);
virtual int operator()();
}
#endif

Example C-25. MySpecTclApp.cpp

static const char* Copyright = "(C) Copyright Michigan State University 2008, All rights reserved."

// Class: CMySpecTclApp

////////////////////////// FILE_NAME.cpp //////////////////////////////////////////////////
#include <config.h>
#include "MySpecTclApp.h"
#include "EventProcessor.h"
#include "TCLAnalyzer.h"
#include <Event.h>
#include <TreeParameter.h>
#include "MyEp.h"

#ifdef HAVE_STD_NAMESPACE
using namespace std;
#endif

// Local Class definitions:

// This is a sample tree parameter event structure: 
// It defines an array of 10 raw parameters that will 
// be unpacked from the data and a weighted sum 
// that will be computed. 
//
typedef
struct {
    CTreeParameterArray& raw;
    CTreeParameter& sum;
} MyEvent;

// Having created the struct we must make an instance 
// that constructs the appropriate objects:
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

MyEvent event = {
   *(new CTreeParameterArray("event.raw", "channels", 10, 0)),
   *(new CTreeParameter("event.sum", "arbitrary"))
};

// Here's a sample tree variable structure
// that defines the weights for the weighted
// sum so that they can be varied from the command line:
// An array is also declared for testing purposes but not used.
typedef
struct {
   CTreeVariable& w1;
   CTreeVariable& w2;
   CTreeVariableArray& unused;
} MyParameters;

// Similarly, having declared the structure, we must define
// it and construct its elements
MyParameters vars = {
   *(new CTreeVariable("vars.w1", 1.0, "arb/chan")),
   *(new CTreeVariable("vars.w2", 1.0, "arb/chan")),
   *(new CTreeVariableArray("vars.unused", 0.0, "furl/fort", 10, 0))
};

// CFixedEventUnpacker - Unpacks a fixed format event into
// a sequential set of parameters.
//
// class CFixedEventUnpacker : public CEventProcessor
{
public:
   virtual Bool_t operator()(const Address_t pEvent,
                              CEvent& rEvent,
                              CAnalyzer& rAnalyzer,
                              CBufferDecoder& rDecoder);
};

Bool_t
CFixedEventUnpacker::operator()(const Address_t pEvent,
                                  CEvent& rEvent,
                                  CAnalyzer& rAnalyzer,
                                  CBufferDecoder& rDecoder)
{

   // This sample unpacker unpacks a fixed length event which is
   // preceded by a word count.
   //
   TranslatorPointer<UShort_t> p(*rDecoder.getBufferTranslator(), pEvent);
   CTclAnalyzer& rAna((CTclAnalyzer&)rAnalyzer);
   UShort_t nWords = *p++;
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

Int_t i = 1;

// At least one member of the pipeline must tell the analyzer how
// many bytes were in the raw event so it knows where to find the
// next event.

rAna.SetEventSize(nWords*sizeof(UShort_t)); // Set event size.

nWords--; // The word count is self inclusive.
int param = 0; // No more than 10 parameters.

while(nWords && (param < 10)) { // Put parameters in the event starting at 1.
  event.raw[param] = *p++;
  nWords--;
  param++;
}
return kfTRUE; // kfFALSE would abort pipeline.

// CAddFirst2 - Sample unpacker which adds a pair of unpacked parameters
// together to get a new parameter.

class CAddFirst2 : public CEventProcessor
{
public:
  virtual Bool_t operator()(const Address_t pEvent,
   CEvent& rEvent,
   CAnalyzer& rAnalyzer,
   CBufferDecoder& rDecoder);
};

Bool_t CAddFirst2::operator()(const Address_t pEvent,
   CEvent& rEvent,
   CAnalyzer& rAnalyzer,
   CBufferDecoder& rDecoder)
{
  event.sum = event.raw[0]*vars.w1 + event.raw[1]*vars.w2;
  return kfTRUE;
}

// Instantiate the unpackers we’ll use.

static CFixedEventUnpacker Stage1;
static CAddFirst2 Stage2;
static MyEp myEventProcessor("left", "right", "x.position", "xscale", "xoffset");

// CFortranUnpacker:
// This sample unpacker is a bridge between the C++ SpecTcl
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

// and a FORTRAN unpacking/analysis package.
// The raw event is passed as a parameter to the FORTRAN function
// f77unpacker. This function has the signature:
// LOGICAL F77UNPACKER(EVENT)
// INTEGER*2 EVENT(*)
//
// As with all event processors, this function is supposed to
// return .TRUE. if processing continues or .FALSE. to abort event processing.
// Unpacked events are returned to SpecTcl via a Common block declared
// as follows:
// ! generated by the unpacker:
// COMMON/F77PARAMS/NOFFSET, NUSED, PARAMETERS(F77NPARAMS),
// 1 FPARAMETERS(F77NPARAMS)
// INTEGER*4 PARAMETERS
// LOGICAL FPARAMETERS
//
// At compile time, define: WITHF77UNPACKER
// and F77NPARAMS to be the maximum number of parameters the Fortran
// unpacker will fill in.
//
// In the unpacker, set FPARAMETERS(i) if i has been unpacked from
// the event.
//
// PARAMETERS are copied from 1 -> NUSED into parameters numbered
// NOFFSET -> NUSED+NOFFSET
// NOFFSET starts from zero.
//
// The fortran program should be compiled:
// cpp -DF77NPARAMS=nnnn yourprog.f > yourprog.for
// f77 -c yourprog.for
// rm yourprog.for
//
// Assuming you've writtne the file yourprog.f

#ifdef WITHF77UNPACKER

struct {
    int nOffset;
    int nUsed;
    int nParameters[F77NPARAMS]; // Fortran will extend this appropriately.
    int fParameters[F77NPARAMS];
} f77params_;

extern "C" Bool_t f77unpacker_(const Address_t pEvent);

class CFortranUnpacker : public CEventProcessor
{
public:
    virtual Bool_t operator()(const Address_t pEvent, CEvent& rEvent,
                                CAnalyzer& rAnalyzer, CBufferDecoder& rDecoder);
};
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

```c
Bool_t CFortranUnpacker::operator()(const Address_t pEvent,
    CEvent& rEvent,
    CAnalyzer& rAnalyzer,
    CBufferDecoder& rDecoder)
{
    Bool_t result = f77unpacker_(pEvent);
    if(result) {
        int dest = f77params_.nOffset;
        for(int i = 0; i < f77params_.nUsed; i++) {
            if(f77params_.fParameters[i])
                rEvent[dest] = f77params_.nParameters[i];
            dest++;
        }
    }
    UShort_t* pSize = (UShort_t*)pEvent;
    CTclAnalyzer& rAna((CTclAnalyzer&)rAnalyzer);
    rAna.SetEventSize( (*pSize) * sizeof(UShort_t));
    return result;
}

CFortranUnpacker legacyunpacker;

#endif

// Function:
//    void CreateAnalysisPipeline(CAnalyzer& rAnalyzer)
// Operation Type:
//    Override
/*
Purpose:

Sets up an analysis pipeline. This function is required and must be filled in by the SpecTcl user. Pipeline elements are objects which are members of classes derived from the CEventProcessor class. They should be added to the Analyzer’s event processing pipeline by calling RegisterEventProcessor (non virtual base class member).

The sample implementation in this file produces a two step pipeline. The first step decodes a fixed length event into the CEvent array. The first parameter is put into index 1 and so on. The second step produces a compiled pseudo parameter by adding event array elements 1 and 2 and putting the result into element 0.
 */

void CMySpecTclApp::CreateAnalysisPipeline(CAnalyzer& rAnalyzer)
{
    #ifdef WITHF77UNPACKER
        RegisterEventProcessor(legacyunpacker);
    
```
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

// Constructors, destructors and other replacements for compiler cannonicals:

CMySpecTclApp::CMySpecTclApp ()
{
    // Destructor:
}

// Functions for class CMySpecTclApp

// Function:
// void BindTCLVariables(CTCLInterpreter& rInterp)
// Operation Type:
// override
/*/ Purpose:
Add code to this function to bind any TCL variable to the SpecTcl interpreter. Note that at this time, variables have not yet been necessarily created so you can do Set but not necessarily Get operations.

/*/ void
CMySpecTclApp::BindTCLVariables(CTCLInterpreter& rInterp)
{
    CTclGrammerApp::BindTCLVariables(rInterp);
}

// Function:
// void SourceLimitScripts(CTCLInterpreter& rInterpreter)
// Operation Type:
// Override
/*/ Purpose:
Add code here to source additional variable setting
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

scripts. At this time the entire SpecTcl/Tk infrastructure is not yet set up. Scripts run at this stage can only run basic Tcl/Tk commands, and not SpecTcl extensions. Typically, this might be used to set a bunch of initial values for variables which were bound in BindTCLVariables.

```c
/*
void CMySpecTclApp::SourceLimitScripts(CTCLInterpreter& rInterpreter)
{
  CTclGrammerApp::SourceLimitScripts(rInterpreter);
}

// Function:
// void SetLimits()
// Operation Type:
//   override
/**
Purpose:

Called after BindVariables and SourceLimitScripts. This function can be used to fetch values of bound Tcl variables which were modified/set by the limit scripts to update program default values.

*/
void CMySpecTclApp::SetLimits()
{
  CTclGrammerApp::SetLimits();
}

// Function:
// void CreateHistogrammer()
// Operation Type:
//   Override
/*
Purpose:

Creates the histogramming data sink. If you want to override this in general you probably won’t make use of the actual base class function. You might, however extend this by defining a base set of parameters and histograms from within the program.

*/
void CMySpecTclApp::CreateHistogrammer()
{
  CTclGrammerApp::CreateHistogrammer();
}

// Function:
//   void SelectDisplayer(UInt_t nDisplaySize, CHistogrammer& rHistogrammer)
// Operation Type:
//   Override.
```
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

/*
Purpose:

Select a displayer object and link it to the histogrammer. The default code will link Xamine to the displayer, and set up the Xamine event handler to deal with gate objects accepted by Xamine interaction.

*/

void CMySpecTclApp::SelectDisplayer(UInt_t nDisplaySize, CHistogrammer& rHistogrammer)
{
    CTclGrammerApp::SelectDisplayer(nDisplaySize, rHistogrammer);
}

// Function:
// void SetupTestDataSource()
// Operation Type:
// Override
/*
Purpose:

Allows you to set up a test data source. At present, SpecTcl must have a data source of some sort connected to it... The default test data source produces a fixed length event where all parameters are selected from a gaussian distribution. If you can figure out how to do it, you can setup your own data source... as long as you don’t start analysis, the default one is harmless.

*/

void CMySpecTclApp::SetupTestDataSource()
{
    CTclGrammerApp::SetupTestDataSource();
}

// Function:
// void CreateAnalyzer(CEventSink* pSink)
// Operation Type:
// Override
/*
Purpose:

Creates an analyzer. The Analyzer is connected to the data source which supplies buffers. Connected to the analyzer is a buffer decoder and an event unpacker. The event unpacker is the main experiment dependent chunk of code, not the analyzer. The analyzer constructed by the base class is a CTclAnalyzer instance. This is an analyzer which maintains statistics about itself in Tcl Variables.

*/

void CMySpecTclApp::CreateAnalyzer(CEventSink* pSink)
{
    CTclGrammerApp::CreateAnalyzer(pSink);
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

```cpp
// Function:
// void SelectDecoder(CAnalyzer& rAnalyzer)
// Operation Type:
// Override
/*
Purpose:
Selects a decoder and attaches it to the analyzer.
A decoder is responsible for knowing the overall structure of
a buffer produced by a data analysis system. The default code
constructs a CNSCLBufferDecoder object which knows the format
of NSCL buffers.
*/
void
CMySpecTclApp::SelectDecoder(CAnalyzer& rAnalyzer)
{
    CTclGrammerApp::SelectDecoder(rAnalyzer);
}

// Function:
// void AddCommands(CTCLInterpreter& rInterp)
// Operation Type:
// Override
/*
Purpose:
This function adds commands to extend Tcl/Tk/SpecTcl.
The base class function registers the standard SpecTcl command
packages. Your commands can be registered at this point.
Do not remove the sample code or the SpecTcl commands will
not get registered.
*/
void
CMySpecTclApp::AddCommands(CTCLInterpreter& rInterp)
{
    CTclGrammerApp::AddCommands(rInterp);
}

// Function:
// void SetupRunControl()
// Operation Type:
// Override.
/*
Purpose:
Sets up the Run control object. The run control object
is responsible for interacting with the underlying operating system
and programming framework to route data from the data source to
the SpecTcl analyzer. The base class object instantiates a
```
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

CTKRunControl object. This object uses fd waiting within the
Tcl/Tk event processing loop framework to dispatch buffers for
processing as they become available.

*/
void
CMySpecTclApp::SetupRunControl()
{ CTclGrammerApp::SetupRunControl();
}

// Function:
// void SourceFunctionalScripts(CTCLInterpreter& rInterp)
// Operation Type:
// Override
/*
Purpose:

This function allows the user to source scripts
which have access to the full Tcl/Tk/SpecTcl
command set along with whatever extensions have been
added by the user in AddCommands.

*/
void
CMySpecTclApp::SourceFunctionalScripts(CTCLInterpreter& rInterp)
{ CTclGrammerApp::SourceFunctionalScripts(rInterp);
}

// Function:
// int operator()()
// Operation Type:
// Override.
/*
Purpose:

Entered at Tcl/Tk initialization time (think of this
as the entry point of the SpecTcl program). The base
class default implementation calls the member functions
of this class in an appropriate order. It’s possible for the user
to extend this functionality by adding code to this function.

*/
int
CMySpecTclApp::operator()()
{
 return CTclGrammerApp::operator()();
}

CMySpecTclApp  myApp;
CTclGrammerApp& app(myApp); // Create an instance of me.
CTCIApplication* gpTCLApplication=&app; // Findable by the Tcl/tk framework.
Example C-26. MyEp.h

```
#ifndef __MYEP_H
#define __MYEP_H

#include <EventProcessor.h>
#include <TreeParameter.h>

#include <string>

class MyEp : public CEventProcessor
{
  private:
    CTreeParameter m_left;
    CTreeParameter m_right;
    CTreeParameter m_position;
    CTreeVariable m_scaleFactor;
    CTreeVariable m_offset;

  public:
    MyEp(std::string left, std::string right, std::string position,
          std::string scaleName, std::string offsetName);

    virtual Bool_t OnAttach(CAnalyzer& rAnalyzer); // Called on registration.
    virtual Bool_t OnBegin(CAnalyzer& rAnalyzer,
                           CBufferDecoder& rDecoder); // Begin Run.
    virtual Bool_t OnEnd(CAnalyzer& rAnalyzer,
                         CBufferDecoder& rBuffer); // End Run.
    virtual Bool_t OnPause(CAnalyzer& rAnalyzer,
                           CBufferDecoder& rDecoder); // Pause Run.
    virtual Bool_t OnResume(CAnalyzer& rAnalyzer,
                            CBufferDecoder& rDecoder); // Resume Run.
    virtual Bool_t OnOther(UInt_t nType,
                          CAnalyzer& rAnalyzer,
                          CBufferDecoder& rDecoder); // Unrecognized buftype.

    virtual Bool_t OnEventSourceOpen(std::string name);
    virtual Bool_t OnEventSourceEOF();
    virtual Bool_t operator()(const Address_t pEvent,
                              CEvent& rEvent,
                              CAnalyzer& rAnalyzer,
                              CBufferDecoder& rDecoder); // Physics Event.
```
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

Example C-27. MyEp.cpp

```cpp
#include <config.h>
#include "MyEp.h"
#include <Analyzer.h>
#include <BufferDecoder.h>
#include <Event.h>

using namespace std;

MyEp::MyEp(string left, string right, string position,
            string scaleName, string offsetName) :
    m_left(left, 0.0, 4095.0, "channels"),
    m_right(right, 0.0, 4095.0, "channels"),
    m_position(position, 0.0, 10.0, "mm"),
    m_scaleFactor(scaleName, 4095.0, "mm"),
    m_offset(offsetName, 2048.0, "mm")
{
}

Bool_t
MyEp::OnAttach(CAnalyzer& rAnalyzer)
{
    return kfTRUE;
}

Bool_t
MyEp::OnBegin(CAnalyzer& rAnalyzer,
               CBufferDecoder& rDecoder)
{
    return kfTRUE;
}

Bool_t
MyEp::OnEnd(CAnalyzer& rAnalyzer,
             CBufferDecoder& rBuffer)
{
    return kfTRUE;
}
```

209
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl

```c
Bool_t
MyEp::OnPause(CAnalyzer& rAnalyzer,
               CBufferDecoder& rDecoder)
{
    return kfTRUE;
}

Bool_t
MyEp::OnResume(CAnalyzer& rAnalyzer,
                CBufferDecoder& rDecoder)
{
    return kfTRUE;
}

Bool_t
MyEp::OnOther(UInt_t nType,
               CAnalyzer& rAnalyzer,
               CBufferDecoder& rDecoder)
{
    return kfTRUE;
}

Bool_t
MyEp::OnEventSourceOpen(STD(string) name)
{
    return kfTRUE;
}

Bool_t
MyEp::OnEventSourceEOF()
{
    return kfTRUE;
}

Bool_t
MyEp::operator()(const Address_t pEvent,
                 CEvent& rEvent,
                 CAnalyzer& rAnalyzer,
                 CBufferDecoder& rDecoder)
{
    if (m_left.isValid() && m_right.isValid() ) {
        double sum = m_left + m_right;
        double dif = m_left - m_right;

        if (sum != 0) {
            m_position = m_scaleFactor*dif/sum + m_offset;
        }
    }
    return kfTRUE;
}
```
Example C-28. Makefile

INSTDIR=/usr/opt/spectcl/3.2
# Skeleton makefile for 3.1

include $(INSTDIR)/etc/SpecTcl_Makefile.include

# If you have any switches that need to be added to the default c++ compilation
# rules, add them to the definition below:

USERCXXFLAGS=

# If you have any switches you need to add to the default c compilation rules,
# add them to the definition below:

USERCCFLAGS=$(USERCXXFLAGS)

# If you have any switches you need to add to the link add them below:

USERLDFLAGS=

# Append your objects to the definitions below:

OBJECTS=MySpecTclApp.o MyEp.o

# Finally the makefile targets.

SpecTcl: $(OBJECTS)
$(CXXLD) -o SpecTcl $(OBJECTS) $(USERLDFLAGS) \$(LDFLAGS)

clean:
  rm -f $(OBJECTS) SpecTcl

depend:
makedepend $(USERCXXFLAGS) *.cpp *.c

help:
echo "make"                = Build customized SpecTcl"
echo "make clean"          = Remove objects from previous builds"
echo "make depend"         = Add dependencies to the Makefile. "
# DO NOT DELETE

MySpecTclApp.o: MySpecTclApp.h MyEp.h
Appendix C. Using the TreeParameterPackage to add Event Processors to SpecTcl
Appendix D. IUCFMaseSpecTcl

This appendix describes the tailored SpecTcl that understands how to analyze data from an XLM-VV with IUCF MASE firmware (see the mase in Chapter 3). Specifically I will describe:

1. The MASE hardware that connects to the XLM-VV
2. How to describe the configuration of the MASE system connected to the XLM-VV
3. How to set a base parameter name for the MASE system parameters and histograms.

The XLM-VV is a readout engine for a system that is capable of reading very large arrays of channels. Each channel provides an analog value as well as timing information. The MASE subsystem implements a hierarchical readout system. The XLM interfaces to up to 32 MASE crates each containing a uniquely numbered COntroller Board (COB). Each MASE crate can host up to 16 CChannel Boards (CHB). Each CHB is responsible for 16 channels. This allows MASE to read detector arrays that consist of up to 8192 channels.

This large channel count makes it impractical to ask a user to specify individual channel names as for e.g. the caen module support. Furthermore MASE can be run only partially 'stuffed', meaning that a variable channel count must be supported by the software. Furthermore, the data read from the XLM at each VM-USB trigger may have data from several physics events.

These considerations led to the realization that a special SpecTcl tailoring would be needed to analyze MASE data. This SpecTcl is called IUCFMaseSpecTcl. This version of SpecTcl is special because:

1. It uses the SpecTcl mega-event support initially written for Lawrence Livermore labs neutron projects to unpack several physics events from a single VM-USB mega-event
2. It allows parameters for MASE to be specified as a base-name and a description of the MASE system which is used to generate a set of regularly named parameter names and raw spectra for those parameters.

To configure MASE channel/raw-spectrum names, use the adcChannels to set a single base name for all of the parameters in the MASE system. This name will be used as the top of a treeparameter tree that has names of the form: basename.cob-number.chb-number.channel

Having set the base parameter name, you must also describe the COBs and CHBs in the system. In the current implementation, the assumption is that COB’s are stuffed starting with COB 0 and that each Crate is ‘tightly stuffed’ from CHB0 though some last CHB. This allows for a very compact description of the system by setting two variables (array elements actually) in the daqconfig.tcl script.

The array maseCOBCount describes the number of COB boards attached to the specific MASE XLM
Warning

While the readout supports multiple MASE XLM systems at this time there is no mechanism to assemble data across the two XLM's in SpecTcl

For example

```
set adcChannels(mase) mase
set maseCOBCount(mase) 2
```

Sets the channel name prefix to `mase` for the `mase` module named `mase`. The system is described as attaching to COB 0 and COB 1.

The `maseCHBCounts` (again actually an array element) gives the number CHB boards in each MASE crate. This variable should be set to a Tcl list for each COB. Remember there’s an assumption of tight packing so if you skip a CHB board number you must still configure this list as if it were present. Elements of the list describe how:

```
set maseCHBCounts(mase) [list 2 3]
```

Says that COB 0 has CHB 0 and CHB 1 present, while COB 1 has CHB 0, CHB1 and CHB2.

The sample configuration will produce parameters and spectra named: `mase.00.00.00 ... mase.00.00.15`, `mase00.01.00 ...mase00.01.15`, `mase.01.00.00 .. mase01.02.15`