NSCL Daq Next Generation

NSCL Daq Next Generation

Table of Contents

List of Examples

I. utilities

Chapter 1. Epics Channel logging

chanlog is a utility that allows you to log the values of a set of EPICS channels. The chanlog manpage provides full reference material for chanlog program.

Chanlog allows you to:

- Interactively view a set of channel values.
- Write or append a set of channel values to a text file.
- Pipe a set of channel values to a program for processing.

Chapter 2. Providing EPICS channel information to Tcl Servers

controlpush is a program that is part of the EPICS support package. It uses a configuration file to connect to a set of EPICS channels, connects to some TCP/IP server and then sends Tcl commands to that server to maintain a set of Tcl arrays that allow the server process to maintain a copy of the state of those channels, their units and when their values were last updated.

controlpush is descsribed fully in: the controlpush manpage.

The intent is that controlpush be pointed at a TclServer. The TclServer is a wish shell extended with the ability to accept commands from TCP/IP connected clients.

You can use controlpush in conjunction with TclServer and application specific scripts to provide a visual display of control system parameters. epicsdisplay is an example of this technique. Another typical use of controlpush is in conjunction with readout skeletons that support tclserver functionality to provide a set of data in the event stream that logs the control system parameters and their changes throughout the run.

controlpush Maintains information in arrays that are indexed by channel name:

`EPICS_DATA(channelName)`

Is the most recently received update for `channelName`.

`EPICS_UNITS(channelName)`

Are the engineering units for `channelName`

`EPICS_UPDATED(channelName)`

Is the timestamp of the alst update received for `channelName`.

Chapter 3. daqstart - Starting programs with logging and monitoring

When running a critical component of a system, it is important to know if that component exits. Furthermore, if the exit is unintended, maintaining a log of the output and error streams for that component can aid in failure analysis.

The daqstart component provides the ability to run an arbitrary executable (or script), capturing the error and or output to a timestamped log and informing you if the executable or script exits. The daqstart reference page describes the detailed use of the daqstart program.

Output monitoring by daqstart uses the concept of *sinks*. While the current implementation only supports file sinks, future implementations may provide other types of sinks such as pipes or interfaces to the unix system log facility, or even database logs.

Output sinks are specified using the form: `type:destination` where `type` defines the sink type (e.g. `file`), and the `destination` is sink type specific (e.g. for a `file` sink the destination is the name of the file that will be the sink).

Now a simple example:

Example 3-1. Logging errors and informing on exit

```
daqstart --error=file:/user/fox/logs/critical.log --notify \
                   mycriticalprogram --anoption=avalue argument
```

The logfile will be `/user/fox/logs/critical.log`. If that file exists new data will be appended to the log. Once a command argument that is not recognized by the daqstart program appears on the command line, the remainder of the line is considered to be the application that will be logged.

Chapter 4. DvdBurner - Using Tcl to burn runs to DVD

DvdBurner is a Tcl package that allows you to burn NSCL data runs to DVD. The DvdBurner package breaks the data up among as many DVD's as needed, however it requires that the data from a single run fit completely on a single DVD. The package is completey described in the dvdburner package manpage.

To use DvdBurner you must make the package known to your script. Doing this involves both making the location of the package known to the Tcl package loader and requiring the package. The following example shows how to do it assuming that you've defined the envirionment varible DAQROOT to point to the directory in which the NSCL DAQ system is installed. At the NSCL, this directory is usually `/usr/opt/daq/someversion` where `someversion` is the version of nscldaq you are using.

Example 4-1. Requesting the DvdBurner package

```
set libdir [file join $::env(DAQROOT) TclLibs]
set auto_path [concat $libdir $auto_path]

package require DvdBurner
```

The example prepends the path to the NSCL DAQ libraries to the `auto_path` that lists the directories searched by the Tcl package loader. Prepending ensures that if there are accidental duplicate package names, we get the one in the NSCL DAQ software.

Typically, the only thing you'll want your script to do is burn a set of runs to DVD, partitioning the data amongst several DVDs as needed. The script below shows how to do write runs 100 through 150 to DVD:

Example 4-2. Writing runs to DVD using DvdBurner

```
DvdBurner::CreateDvds [list 100 150]
```

If the second parameter to `DvdBurner::CreateDvds` is omitted, all runs starting with first parameter are written to DVD. With no parameters, all runs are written to DVD:

Example 4-3. Writing all runs to DVD

```
DvdBurner::CreateDvds
```

Writes all runs to DVD.

Chapter 5. Utilities for burning data to DVD

While the NSCL does not officially support burning data to DVD, two utilities are provided that allow you to create informal archives of data taken with NSCLDAQ on DVD.

The command line utility dvdburn provides a simple command that burns a set of run to DVD, creating more than one DVD if needed, splitting DVDs between runs. See the dvdburn reference page for information about how to use this program.

A simple graphical user interface wizard for burning NSCLDAQ data to DVD is provided by burngui. burngui is really a front end for dvdburn. See the burngui reference page for complete information on that application.

Chapter 6. The epics display utility

The epicsdisplay utility provides a tabular display of an arbitrary set of epics channels. It is possible to configure a set of channels to be displayed on a chart recorder at the bottom of the display as well. Complete reference documentation is available at the epicsdisplay reference page

epicsdisplay reads in a configuration file. The configuration file is compatible with the controlpush application, which uses it to fetch epics channesl and make them known to epicsdisplay. epicsdisplay interprets additional information in the file as metadata that describes such things as:

- Alarms on the channel and how to display the alarm condition.
- Whether or not a channel should be displayed as a trace on a chart recorder graph at the bottom of the window.

Extensive online help is provided via the Help⟶Topics... menu entry.

epicsdisplay

Name

`epicsdisplay` — Display epics channels

Synopsis

epicsdisplay channel file

DESCRIPTION

Monitors a set of EPICS channels in tabular and, optionally, chart recorder form. channelfile is the name of a file that describes the epics channels to monitor. The file consists of two types of text lines; comments and channel lines. Comment lines begin with the hash character lines begin with a channel name followed by whitespace, followed by an optional comment. If the optional comment contains the special word Ã¢Ã¢chartÃ¢Ã¢, the channel is added to the chart recorder.

Channels can also have alarms defined on them. An alarm is said to fire whenÃ¢[m ever the channel value is outside of a Ã¢nominalÃ¢ window. Alarms are also defined using special keywords in the comment field of the channel. The keyÃ¢[m word nominal indicates that the next word on the line defines the channelÃ¢s nominal window. The nominal window definition can be defined either symmetriÃ¢[m cally or asymmetrically about a nominal value. The nominal window can also be defined in absolute terms or in terms of a percentage of nominal value. Describing nominal windows is most easily done via example.

`nominal 35:3`

Defines a symmetric nominal window about the nominal value 35. This nominal window is the range 32 through 38.

`nominal 35:10%`

Defines a nominal interval that is +/- 10% of 35... e.g. the interval 32.5 through 38.5

`nominal 35+3-4`

Defines a nominal window between 31 and 38. Similarly:

`nominal 35+5%-10%`

Defines the nominal interval 32.5 through 36.75. Note that for asymÃ¢[m metric intervals mixes of absolute and percentage tolerances are allowed. Furthermore, the - and + are interchangeable.

When a channel transitions to the alarmed state, actions defined on the comÃ¢[m ment line for the channel are taken. Actions are defined by the keyword alarÃ¢[m mactions followed by a comma separated list of actions. Supported actions are:

`page`

The alarm is added to a page that describes all the current alarms.

`beep`

A beep is periodically emitted while the channel is in the alarmed state.

`popup`

A popup window is displayed when the alarm is in the alarmed state.

`color=colorspec`

The alarm is displayed with the specified colorspec colored background. colorspec is a color using any color specification that is recognized by Tcl. (.e.g. color=red is the same as color=#ff0000).

OPERATING INSTRUCTIONS

The display is divided into three sections. The top section is the channel taÃ¢[m ble. The middle section the chart recorder section and the bottom section the chart control panel.

While it appears as if the table section has no controls, in reality it is possible to resize all of the columns of the table. Expanding the window width adds space to the right most, Comment column. All other columns may be expanded by dragging their borders with the mouse.

The chart recorder control panel allows the user to perform a few operations on the chart. The Clear button resets the time to 0 and clears the data accumulated so far.

The rightmost three controls manage the time scale of the chart widget. Changes to these widgets do nothing until the Set button is clicked. The widget states at that time determine the scale of the chart recorder. If the Ã¢Ã¢Auto RangeÃ¢Ã¢ checkbutton is set, the time scale will change dynamically to accomodate the full time range of values. If unchecked, the value in the entry widget labelled `Range (min): determines the width in time of the chart. The chart is scrolled as needed (by 10% of the range) to keep the most recent data visible.`

Dependencies

- Requires the controlpush utility.
- Requires the server port manager application.

Chapter 7. Scaler Display Software.

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.

Chapter 8. Sequencing runs

The NSCL Run sequencer provides a mechanism for scheduling as series of timed runs. Prior to the start of each run, the sequencer allows custom actions to be taken. While these actions normally set up hardware, there is no actual requirement this be the case. Any parameterizable action can be taken.

For a full description of the sequencer, see its reference manual page.. This chapter gives an overview that describes how to use and configure the sequencer.

8.1. Configuring the sequencer.

The sequencer requires the following configuration work:

- The actions executed prior to each run must be defined. Each action accepts a parameter which becomes a column of the run plan.
- A run plan. This is a specification of the parameters for each of the actions to be taken prior to starting a run. The run plan is displayed to the user as a table which can be edited.
- The sequencer must be incorporated into the Readout GUI (see the chapter on the Readout GUI).

Furthermore, each action must provide Tcl code that executes it and, optionally initializes its access to whatever it might control.

The actions are defined in a file named `sequencer.conf` in the current working directory at the time the sequencer starts. This file describes one action on each line. Each sequencer line contains several whitespace separated fields. In order:

- Name of the action. This name will be passed to the action's procedure.
- GUI name of the action. This name will be used to label the action's column in the run plan table.
- Set action (optional): This is the name of a Tcl **proc** used to perform the action. It is passed the name of the action (value of the first column), and the value of the action parameter. If omitted no action is taken at the beginning of the run.
- Initialize action (optional): This is the name of a Tcl **proc** that is called as the sequencer initializes. it is passed tha name of the action.

When the sequencer starts it will source in the Tcl script file `sequencerActions.tcl` from the current working directory. This file contains arbitrary Tcl scripts. It is expected to define all of the action **proc**s described in `sequencer.config`.

Run plan files can be created graphically in the sequencer table. Simply edit each cell of the table with the appropriate parameter for that run and that action. The File⟶Save... menu selection allows yoj to save run plans for later (re)use.

Run plan files are plain text files. Each line contains the parameterization of a run. Lines contain whitespace separated fields where each field is a parameter value for a run.

The run sequencer must be integrated with the Readout GUI. This is done by providing (or modifying an existing) `ReadoutCallouts`. See the Readout GUI reference page for information about how the readout GUI locates its `ReadoutCallouts` file.

The sequencer is provided as a Tcl package located in the `TclLibs` directory tree of the NSCL DAQ installation. To load it you will need to have this directory in your Tcl Load path. This can be done either by setting the `TCLLIBPATH` environment variable, or by having `ReadoutCallouts.tcl` add the appropriate directory to the `auto_path` loader list.

The example below takes a hybrid approach. We assume you've set an environment variable `DAQROOT` to be the top level directory of the NSCL DAQ installation. At this NSCL, this might be `/usr/opt/daq/9.0` for example. The example shows additions to the top of the `ReadoutCallouts.tcl` script that make the sequencer package loadable, and load it:

Example 8-1. Loading the sequencer package

```
set daqroot $env(DAQROOT)                ❶
set libdir [file join $daqroot TclLibs]  ❷

set auto_path [concat $libdir $auto_path] ❸

package require runSequencer             ❹
```

❶ The global `env` array contains a copy of the environment variables indexed by variable name. This line gets the `DAQROOT` environement variable.

❷ The Tcl **file join** joins filename path elements inserting the appropriate path separator, and quoting if needed. This line constructs the name of the directory in which NSCL DAQ packages are found.

❸ This line pre-pends the NSCL DAQ Tcl library directory to the list of directories (`auto_path`) searched by the Tcl. Prepending ensures that if there are any package name collisions, the NSCL DAQ packages will be found first.

❹ This line actually loads the sequencer package. By the time this line finishes, the sequencer configuration files, and action scripts will have been read, and the sequencer user interface created.

8.2. Using the sequencer.

The sequencer GUI consists of a table called the *run plan table* Edit this table to set the parameters for each action item column for each run in the plan (runs are rows across the table).

The File⟶Save... menu will save a run plan for later use. If you've already created a run plan, you can read it in with the File⟶Open... menu selection. Finally the File⟶Clear menu selection clears the run plan table.

The length of each run is set by the length of the timed run section on the main ReadoutGUI window. To execute a run plan, click the Execute button on the run plan table window. This button then is relabeled Abort, and can be used to abort a run plan if something goes wrong.

As the run plan executes, the run that is active is highlighted on the run plan table. When the run plan is complete, or if the run plan was aborted, the Abort is relabeled Execute and will restart the run plan if clicked.

Chapter 9. The tcl server application

TclServer is a deceptively simple program. It is a wish (Tk) interpreter that allows Tcp/IP connections from a local or remote client. Once connected, the client can send messages consisting of Tcl commands and receive the result of executing those commands from the server.

TclServer is used as the basis of a number of applications in the NSCLDAQ system including the scaler display program, the epicsdisplay program and other software written by specific groups. Tcl servers can be used as distributed data repositories (store data in server Tcl variables and/or arrays and get them back out), or as display programs that can be remote-controlled. The production readout framework has a Tcl Server component that can be enabled.

This chapter contains: A description of the TclServer protection model. For complete information about the TclServer see the TclServer reference pages .

The TclServer protection model is not very stringent. The assumption is that TclServers can only be reached by trusted systems. Specifically systems that have not been broken into and will not spoof host names. When a TclServer starts, it will only allow connections from `localhost`.

The TclServer extends the Tcl/Tk command set with the command **serverauth** this command allows you to authorize additional systems to connect to the server. The complete form of the **serverauth** command is described in the reference page for the **serverauth** command.. section.

The following command adds the node `spdaq20.nscl.msu.edu` to the set of hosts that are allowed to connnect to the server.

Example 9-1. Using serverauth to authorize a node

```
serverauth add spdaq20.nscl.msu.edu
```

II. libraries

Chapter 10. Integer byte order conversion library

The cvt package provides functions that convert data from foreign host to local host integer formats and back. These functions are designed to make use of a byte order signature that describes the byte order of the foreign host.

Reference material for this library is available in the reference manual page for cvt .

This chapter gives an high level view of how the library should be used.

10.1. Using the conversion library in your code

The conversion library is a set of C functions. To use it, you will need to include the library header in your source code with a line like:

Example 10-1. Including the cvt header

```
#include <cvt.h>
```

You will also need to link the conversion library into your executable program. Since the conversion headers and libraries are generally not installed in the normal search paths for headers and libraries, you will need to help the compiler find them at compile and link time. In this discussion, we will assume you have defined an environment variable DAQROOT to point to he NSCL DAQ installation directory (at the NSCL this will typically be in `/usr/opt/daq/some-version` where `some-version` is the version of the NSCLDAQ you are selecting.

Example 10-2. Compiling a C or C++ source file that includes `cvt.h`

```
gcc -c -I$DAQROOT/include myprogram.c

or

g++ -c -I$DAQROOT/include myprogram.cpp
```

Example 10-3. A makefile rule that builds a C++ program using the cvt package

```
theprogram: $(OBJECTS)
```

```
g++ -o theprogram $(OBJECTS) -L$(DAQROOT)/lib -lcvt -Wl,"-rpath=$(DAQROOT)/lib"
```

10.2. Byte order signatures and conversion blocks

Byte order signatures are known integer values that are available in the foreign system's byte oder. By examining these on a byte by byte basis, the byte ordering of the remote system can be determined. While typically systems have a consistent ordering of bytes in words and words in longwords, the library does not assume that.

A system that is supplying data to be converted with this library must provide a 16 bit and a 32 bit signature. These are the values `0x0102` and `0x01020304` repsectively written in that system's native byte ordering.

The function `makecvtblock` takes those signatures and delivers a DaqConversion block. The DaqConversion block is a data structure that provides conversion tables for foreign to host and host to foreign conversions.

Suppose you have data in a structure that contains these signatures as fields named `s_ssig` and `s_lsig`. The example below shows how to create a conversion block:

Example 10-4. Creating a DaqConversion

```
#include <cvt.h>
...
DaqConversion conversionBlock;
makecvtblock(data.s_lsig, data.s_ssig, &conversionBlock);
...
```

The DaqConversion block `conversionBlock` can be used in subsequent calls to do byte order conversions.

10.3. Data conversion

Data conversion functions are provided to convert both from the foreign system to the host, and from the host to the foreign system. Before we look at these functions, some definitions:

host

The system that is running our program.

foreign system

The system that produced the data we are trying to process. This could be the same system as the host or a different system. This could have the same or different native byte ordering than the host.

Functions are provided that allow you to convert shorts and longs in either direction. These functions are described more completely in the cvt reference page. Briefly however:

`ftohl`

Converts a longword (32 bit integer) from foreign to host format.

`ftohw`

Converts a short (16 bit word) from foreign to host format.

`htofl`

Converts a longword (32 bit integer) from host to foreign system format.

`htofs`

Converts a short (16 bit integer) from host to foreign system format.

Chapter 11. The NSCL Exception class library

C++ provides an error detection/recovery mechanism called exception handling. Exception handling, while not always easy to program correctly is well suited to object oriented software development.

Exceptions in C++ are objects that are thrown in search of catchers. As each call frame is searched for a catcher, and the exception propagates up the call chain. Object destructors for non-dynamically allocated objects get called ensuring proper object cleanup. If an exception can find no catcher, the application exits with an error.

The linguistic construction for emitting, or throwing and exception is the **throw** statement. For example:

```
throw SomeObject;
```

The linguistic mechanism for specifying exception catchers is the **try/catch** block:

```
try {
   ...
}
catch (SomeType variable) {
   ...
}
...
catch (...) {
...
}
```

If an exception is thrown within the block body of the **catch** statement, the catch blocks are searched for a matching data type, and the textually first matching block is executed. If none match and a **catch (...)** is specified it is invoked.

You can also re-throw an exception after processing using an empty **throw** statement. This is used in the following example to destroy some dynamically allocated memory.

```
SomeType* pType;
try {
   pType = new SomeType;
   ...
}
catch (...) {
    delete pType;
    throw;
}
```

Exception catching does allow polymorphism. For example, suppose we have a base class `MyBase` and a derived class `Derived`.

```
try {
    ...
    throw Derived;
    ...
}
catch (MyBase& reason) {
    ...
}
```

The catch block will execute, and if virtual methods are called, the `Derived` implementation will be used.

11.1. Incorporating the library in your programs

So much for an introduction to C++ exception handling. Based on the presentation so far, however there are clear advantages to defining a hierarchy of exceptions that derive from a common base class that provides a common set of reporting facilities. The NSCL Exception class library in SpecTcl and nscldaq does this. This library named `libException.so` that can be found in the lib directory of the NSCL DAQ or SpecTcl installations. Linking with that library is a matter of e.g.:

```
g++   ...  -L/usr/opt/spectcl/3.2/lib -lException -Wl,"-rpath=/usr/opt/spectcl/3.2/lib"
```

Be sure to substitute the correct top level directory for `/usr/opt/spectcl/3.2` in both places in the example above.

The base class of the library is called `CException`. All NSCL frameworks and applications enclose essentially the entire program or the body of each thread in a try/catch block that can catch and report the error information from a `CException` prior to exiting. Your code can catch this or derived class issues either for error reporting or error recovery purposes.

11.2. Exception classes

The exception classes forma hierarchy with `CException` as the base class. `CException` provides a unified interface to all exceptions that allow you to get human readable messages and status information that can be processed by computers (e.g. for exception recovery).

All exceptions have the following member functions (virtual):

- `virtual const const char* ReasonText();`
  Which returns a human readable error string that contains an error message describing why the exception was thrown and what was being done when the error that triggered the exception occured.
- `virtual const Int_t ReasonCode();`
  Which returns some exception specific error code, that can be processed by software for error recovery.

The example below shows the simplest handling of `CException` types. Note that the exception is caught by reference so that member functions retain their polymorphism.

Example 11-1. Catching `CException` and exiting

```
#include <Exception.h>
#include <iostream>
using namespace std;
...
try {
...
}
catch (CException& error) {
    cerr << err.ReasonText() << endl;
    exit(-1);
}
```

The example prints out a meaningful error message and exits the program.

Chapter 12. Access control and security

Security consists of authentication, and authorization. Authentication determines who the entity requesting service is. Authorization determines if the authenticated entity has a right to request the service it has requested.

The NSCLDAQ security software helps your application to perform simple authentication and authorization according to policies set by your application. The NSCLDAQ security software is not a high security system. It is primarily intended to avoid errors on multi users data taking systems. It is not intended to secure against malicious attacks.

The assumption is that your data acquisition system is already secured, from unauthorized users either by living behind a firewall, or by security management in the system itself.

Two class hierarchies work together to do authentication. Authenticators, and Interactors. Interactors accept authentication credentials from some source (credentials are anything that identify an entity within some authentication scheme). Authenticators examine those credentials to determine if they are legitimate.

12.1. Incorporting the software into your code

In order to incorporate this software into your application you will need to include various header files. The reference section for each class describes the headers you need. The headers live in the `include` subdirectory of the nscldaq installation. Suppose you have an environment variable `DAQROOT` defined that points to the top level directory of the NSCL DAQ installation (at the NSCL, this is `/usr/opt/daq/someversion` where `someversion` is the version installed), to compile modules that include headers from this library you must:

Example 12-1. Compilation switches for the security includes

```
g++ -c -I$DAQROOT/include ...
```

At link time, you must link the security library into your application. To do this you must supply switches to help the linker locate the library at both link and run time, as the library is typically a shared library. For example:

Example 12-2. Link switches for the security library

```
g++ -o myapplication ... -L$DAQROOT/lib -lSecurity -Wl,"-rpath=$DAQROOT/lib"...
```

12.2. Authenticators

The `CAuthenticator` class is the abstract base class of all authenticators. It provides an interface that all authenticators must meet. In typical operation, an application will select a concrete authenticator, and pair it with a concrete interactor. The application will then authenticate requestors using this pair of objects. Here’s some sample boilerplate code:

Example 12-3. Boilerplate DAQ Authorization code

```
#include <Authenticator.h>
#include <Interactor>

...
    CAutenticator* pAuthenticator =  selectAuthenticator();
    CInteractor*   pInteractor    =  selectInteractor();
    if (pAuthenticator->Authenticate(*pInteractor)) {
        // Authorized to use the service.
        ...
     }
    else {
     // Not authorized to use the service.
     ...
    }
    // Assuming the interactor an authenticator are dynamically allocated
    // by the selection functions.

    delete pAuthenticator;
    delete pInteractor;
```

In the example above, the functions not shown, `selectAuthenticator` and `selectInteractor` determine the actual authentication method and authorization policy. The implementation of these functions will vary from application to application.

The complete definitinon of `CAuthenticator` is provided in its reference pages. The library provides the following concrete authentication classes:

`CPasswordCheck`

The entity must provide a correct password. Note that the current set of interactors do not support encrypted interactors. This can be extended if required.

`CUnixUserCheck`

The entity must provide a valid username and password that is could login to the local unix system.

`CTclAccesListCheck`

Intended for use within a Tcl interpreter. The entity must supply some string that is an element of a Tcl list held in a Tcl variable.

`CAccessListCheck`

The entity must supply a string that is one of a set of strings given to the authenticator.

`CHostListCheck`

Same as `CAccessListCheck` but the access list is a set of IP addresses. The entity's credentials are translated to an IP address and looked up in the set of allowed items.

12.3. Interactors

Interactors are classes that are concrete classes derived from the abstract base class `CInteractor`. They are intended to obtain the authentication credentials from the entity requesting service. If the interactor is in some way interactive, it may also prompt the user for the elements of the credentials required.

The abstract base class `CInteractor` provides the following member functions as abstract virtual functions that will be implemented differently in each concrete interactor class:

```
virtual int Read(Uint_t nBytes, void* pData);
```

Reads data from the interactor. The `CAuthenticator` base class provides a convenience method that reads an entire line of text from the requestor.

```
virtual int Write(UInt_t nBytes, void* pData);
```

Writes data to the entity that's behind the interactor.

```
virtual int ReadWithPrompt(Uint_t nPromptSize, void* pPrompt, UInt_t nReadSize, void* pRea
```

Prompts for input if the interactor supports prompting and then accepts data from the entitiy.

In addition to CInteractor, the following concrete interactors have been supplied.

`CStringInteractor`

This interactor supplies a string that you have gotten by whatever means you got it. It can be used when interactors are not suitable for acquiring the stringified credentials.

`CFdInteractor`

This interactor accepts the credentials via an open file descriptor.

`CIOInteractor`

This interator uses a pair of other interactors, an output and an input interactor. Prompting is done on the output interactor and the credentials read from the input interactor.

Chapter 13. C++ encapsulation of a Tcl API subset

The tclPlus library provides a C++ object oriented encapsulation of a large subset of the Tcl application programming interface. This section provides reference material for this class library.

Sections of the NSCLDAQ and SpecTcl products both make extensive use of this library. It is therefore distributed with both prodcuts, from a common source base. Therefore, there are two ways to link to this library. First, using SpecTcl:

```
g++ -o yourprogram yoursources  \
               -L${SPECTCLHOME}/lib -ltclPlus -lException
```

where `${SPECTCLHOME}` represents the top level director of your SpecTcl installation. At the NSCL, for example, this could be `/usr/opt/spectcl/3.2`

Second, using nscldaq, replace `${SPECTCLHOME}` with the top level directory of your NSCLDAQ installation, e.g. `/usr/opt/daq/8.1`.

Some brief descriptions of the primary classes in this library follow:

`CTCLApplication`

This is a base class for complete applications that extend the Tcl/Tk interpreter. By appropriately subclassing it you can build your own standalong extended Tcl/Tk interpreters.

`CTCLException`

The Tcl/Tk API use return codes to indicate error conditions. This is error prone. The tclPlus library converts these return codes in to thrown exceptions of the type `CTCLException`. To handle these exceptions properly requires use of C++ try/catch blocks. A code fragment example of this is provided in the `CTCLException`(3) manpage.

`CTCLInterpreter`

The `CTCLInterpreter` object is at the core of the library. It is a wrapping of a Tcl_Interp* along with member functions that access many functions that logically operate on a Tcl interpreter.

`CTCLInterpreterObject`

The `CTCLInterpreterObject` wraps objects that require an interpreter to function correctly. It is a base class for many of the classes in the library. It provides common services for those objects.

`CTCLList`

In Tcl scripting, lists play a key role in providing a structured data type. The `CTCLList` object can be created on a string believed to be a list, and used to split a list into its elements or merge a set of words into a list.

`CTCLObject`

Tcl has the philosophy that everything can be treated as if it were a string. In older versions of the interpreter, everything *was* a string. This led to a great deal of inefficiency converting other data types to and from strings. The Tcl_Obj type was created to reduce this inefficiency and to reduce the amount of string copying that was necessary to invoke commands.

A Tcl_Obj is an object that stores the string representation and another representation type for a Tcl interpreter entity. Tcl_Obj also provides for object sharing with copy on modify semantics. This reduces much of the string copying overhead that was previously associated with executing Tcl interpreter commands.

The `CTCLObject` is a wrappig of a Tcl_Obj along with functions that operate on the underlying object.

`CTCLObjectProcessor`

Key to the concept of the Tcl interpreter as an application scripting language is the ability to add new commands to the interpreter that are application specific. The `CTCLObjectProcessor` class is an abstract base class that, when subclassed and instantiated adds new commands to the interpreter.

`CTCLVariable`

The `CTCLVariable` class provides access to Tcl script variables.

Chapter 14. NSCL DAQ Thread Library

The NSCL DAQ thread library supplies object oriented threading support. This chapter describes:

- The thread and synchronization model supported by the library
- What you need to do to incorporate the library into your application code.
- A summary of the classes in the library and links to the reference material for each of them.

14.1. The thread and synchronization model

The NSCL Daq software models each thread as an object from a class that is derived from the `Thread` base class. This class provides functions that allow a thread to be created, started, exited, joined to and detached.

The body of a thread is provided by you when you create your concrete `Thread` subclass. The thread body is just your implementation of the `virtual run` method. Threads exit by returning from the `run` method.

Given a running thread, the `join` thread allows the caller to block until the thread represented by the object exits. `joining` is a necessary part of thread cleanup, unless the thread invokes its own `detach` member.

In the following example, a thread is designed that will block itself for a second, print a message and exit. Code is shown that creates the thread, starts it, joins it an deletes it. Note that deleting a running thread object is a bad idea and has undefined consequences.

Example 14-1. The life of a thread

```
        ...
class MyThread : public Thread {      ❶
public:
    virtual void run();
};
void
MyThread::run()                      ❷
{
    sleep(1);
    std::cerr << "My thread " << getId() << " is exiting\n";
    return;
};
   ...

MyThread* aThread = new MyThread;   ❸
aThread->start();                     ❹

   ...

aThread->join();                    ❺
delete aTrhead;                     ❻

    ...
```

❶ By making `MyThread` a subclass of `Thread`, objects from class `MyThread` can be started as independent threads of execution

❷ The `run` member of a thread is an abstract method. Your thread classes must supply the behavior for this member. When the thread is started, the `run` method gains control in the context of the new thread.

❸ Creating a thread is simple. Just create an object that is of the thread class type.

❹ Starting a thread is equally simple, Just call the thread's `start` method. That starts the thread with an entry point that eventually calls the `run` method.

❺ It's not safe to destroy a thread that is executing. Calling a thread's `join` method blocks the caller's thread until the thread exits. Note that it is not safe for a thread to call its own `join` method since that will block the thread forever.

❻ Once a thread has exited, the object that ran it can be destroyed. That effectively destroys the thread. If the state of the exiting thread allows, it is possible to start the thread again after it has exited.

Non trivial threaded software will almost always need some means for threads to synchronize against one another. Consider the following trivial, but wrong, example:

Example 14-2. Why synchonization is needed

```
int someCounter = 0;

class MyThread : public Thread
{
    virtual void run() {
        for (int i=0; i < 10000; i++) {
            someCounter++;
        }
};

...
MyThread* th1 = new MyThread;
MyThread* th2 = new MyThread;
th1->start();
```

```
th2->start();
th1->join();
th2->join();
delete th1;
delete th2;
std::cerr << "someCounter = " << someCounter << std::endl;
...
```

In this program, two threads increment the variable someCounter in parallel 10,000 times each. You might expect the output of this program to always be 20000. Most of the time, it probably will be. Sometimes it will be something less than 20000.

Consider what

```
someCounter++;
```

actually does. The value of `someCounter` is fetched to a processor register, the register is incremented and finally, the register is stored back into `someCounter`.

Suppose thread `th1` fetches `someCounter`, and increments the register but before it has a chance to store the incremented value back into `someCounter` `th2` executes, fetches `someCounter` (the old value), increments the register and stores the value back. Now `th1` gets scheduled, and stores its value back.

This sequence of steps results in a lost increment. It is possible to construct sequences of execution, that result in a final value of `someCounter` holding any value from 10000 through 20000 depending on how access to `someCounter` is interleaved.

One way to fix this is to ensure that the increment of `someCounter` is *atomic* with respect to the increment. The NSCLDAQ threading library provides a synchonization primitive called a `SyncGuard` that can be used to implement the Monitor construct first developed by Per Brinch Hansen (see Wikipedia's Monitor (synchronization) (http://en.wikipedia.org/wiki/Monitor_%28synchronization%29) page.

Let's rewrite the previous example so that the increment is atomic with respect to the scheduler. To do this we will isolate the counter in a class/object of its own so that it is not possible to use it incorrectly

Example 14-3. Using `SyncGuard` to implement a monitor

```
class ThreadedCounter {
private:
    Synchronizeable  m_guard;      ❶
    int              m_counter;
public:
    void increment();
    int  get() const;
};
void
ThreadedCounter::increment()
{
    sync_begin(m_guard);          ❷
    m_counter++;
    sync_end();                  ❸
}
int
ThreadedCounter::get() const
{
    return m_counter;
}

ThreadedCounter someCounter;
class MyThread : public Thread
{
    virtual void run() {
        for (int i=0; i < 10000; i++) {
            someCounter.increment();     ❹
        }
};

...
MyThread* th1 = new MyThread;
MyThread* th2 = new MyThread;
th1->start();
th2->start();
th1->join();
th2->join();
delete th1;
delete th2;
std::cerr << "someCounter = " << someCounter.get() << std::endl;
...
```

❶ This member is the synchronization element. We will see it used in the `increment` member.

❷ The `sync_begin()` enters the monitor. Only one thread at a time is allowed to execute the code between a `sync_begin` and a `sync_end` for the same synchronizing object.

❸ This call leaves the monitor. The effect of the monitor is to make the increment atomic with respect to the scheduler.

❹ By using the `increment` function, the counter is incremented atomically.

14.2. Incorporating the library into an application.

To incorporate the thread library in your software, you will need to ensure that the compiler can locate the library headers and the library, both at link time and at load time.

The library headers are in the `include` directory of the NSCLDAQ installation tree. The library is called `libthreads` and is in the `lib` directory of the installation tree.

In the compilation example below, we assume that an environment variable named DAQROOT has been defined to point to the top level of the NSCLDAQ installation tree:

Example 14-4. Compiling and linking NSCLDAQ threaded software

```
g++ -o mythreadedapp -I$DAQROOT/include mythreadedapp.cpp -L$DAQROOT/lib -lthreads \
    -Wl,"-rpath=$DAQROOT/lib"
```

14.3. Pointers to the reference material

This section provides pointers to the reference sections. The following three classes define the public interfaces for the threading and synchronization library:

- `Thread` is the abstract base class from which you can construct application specific threads.
- `Synchronizable` is the class used to perform synchronization.
- `SyncGuard` is built on top of `Synchronizable` to provide monitor like synchronization semantics.

Chapter 15. Parsing and URIs

Elements of the NSCL acquisition system are named with Universal Resource Locators, or URIs. For the purposes of the NSCLDAQ, a URI is not really distinguishable from a URL (Universal Resource Locator). It has the form: `protocol://host:port/path`, or `protocol://host:port`

protocol

Is the means used to talk with a resource. For nscldaq this is most often `tcp`

host

Is the system that hosts the resource that is identified by the URI. The host can either be a dotted IP address, or the DNS name of a host on the network.

path

Identifies the resource within the host. This identification may differ depending on what the resource is. For a ring buffer, for example, the path is the name of the ring buffer.

port

Identifes the network port number used to contact the server that provides the resource to the network.

The URI library is a class that parses URI's and provides member functions that return the elements of a URI. Here's a trivial example of the URI library in use:

Example 15-1. Sample URI library program

```
#include <URL.h>
#include <URIFormatException.h>
#include <iostream>
#include <stdlib.h>
int main(int argc, char** argv)
{
    if (argc != 2) {
        cerr << "Usage:\n";
        cerr << "  urltest URI\n";              ❶
        exit(-1);
    }

    try {                                            ❷
       URL uri(argv[1]);                             ❸
       cout << "Protocol: " << uri.getProto()
            << " host : "   << uri.getHostName()
            << " path: "    << uri.getPath()       ❹
            << " port: "    << uri.getPort() << endl;
    }
    catch(CURIFormatException& error) {
        cerr << "URI was not valid: "
             << except.ReasonText()           ❺
             << endl;
        exit(-1);
    }
    exit(0);
}
```

❶ The test program accepts the URL to parse as its command line parameter. If the user does not provide a URL or provides too many parameters, the progrm’s usage is printed on `stderr` before the program exits.

❷ The `URL` constructor parses the URI. If, however the URI is not valid syntax, or refers to a host that does not exist, it will throw an exception (`CUIRFormatException`). The construction and manipulation of the URI is therefore wrapped in a **try**/**catch** block.

❹ This section of code takes the parsed URI and demonstrates how to pull the elements of the URI out and print them.

❺ This code catches errors that may be thrown by the URI parse. The reason for the error is printed out. For more information see the `CURIFormatException` reference page.

To incorporate the URI library into your source code, you must compile with an appropriate set of switches to allow the compiler to locat your header files, and the linker to locate the URL library both at link time and at load time. If the example program above is called `urltest.cpp` and if you have an environment variable DAQROOT that points to the top level directory of the NSCLDAQ install tree the command below will compile and link the program.

Example 15-2. Building `urltst.cpp`

```
g++ -o urltest -I$DAQROOT/include urltest.cpp -L$DAQROOT/lib -lurl -Wl,"-rpath=$DAQROOT/lib'
```

III. servers

Chapter 16. Service Port Manager.

The TclServer application makes it trivial for users to create and use application specific servers. When doing so, however one problem that crops up is which service port should be used by the service when it listens for connections. The service port used must be unique across all servers within a specific system.

The NSCL PortManager application attempts to solve this by managing a block of ports on behalf of servers running in a specific system and advertising the application and user names associated with ports that it has allocated. By interacting with the port manager, servers can be assured of getting a unique port. By interacting with the port manager, clients can determine the likely port on which a server they are interested in is listening.

The Port manager conists of the following components, which are described in this reference material:

- The port manager daemon manages a block of ports for the system in which it is run. The reference material for the application describes the switches that control the port manager's operation, the log/status files it creates and the protocol that should be used to interact with the daemon.
- A C++ class library that allows C++ software to interact with the port manager either as a server or as a client.
- The portAllocator package, which allows Tcl scripts to interact with a port manager either as a server or a client.

IV. Reference Pages

I. 1daq

daqstart

Name

`daqstart` — Monitor essential programs

Synopsis

daqstart **`options...`** ***`command`***

DESCRIPTION

This utility facilitates the collection of logging/debugging information from programs that were not originally intended to produce logs. daqstart runs a target program. Stdout and stderr of the target program are initialized as pipes that are monitored by the daqstart program. Output on each pipe is collected a line at a time and relayed to daqstarts stdout or stderr as appropriate. If a logging sink has been attached to the standard file, lines are timestamped and logged to that sink as well. This utility is also very useful for programs that are run from desktop shortcuts under window managers such as e.g. kde.

On target program exit, if an error sink is specified, an exit message is logged to the error sink. The exit message contains such information about why the program exited as UNIX can provide (exit status and signal), and is also timestamped.

If the `--notify` option is specified, the exit message is also sent to daqstart's stderr. Additionally, if the DISPLAY environment variable is defined and the PopUp program is executable in the current path, it is invoked to pop up an Xwindows dialog with similar information.

OPTIONS

Each option has a short and a long form. Short form options are preceded by a single `-`. Long options are preceeded by `--`. If a value is required, it should be butted up against short options or separated from long options by an `=`. For example: `-osomesink` and `--output=somesink` are equivalent.

`-h --help`

Display short help and exit.

`-V --version`

Print program version number and exit

`-e`*`Sink`* `--error=`*`Sink`*

Specify a logging sink for the program's standard error output. If specified, lines sent by the program to stderr are timestamped and teed to this sink as well. See SINK SPECIFICATIONS for more information about specifying sinks.

`-o`*`Sink`* `--output=`*`Sink`*

Specify a logging sink for the program's standard output output. If specified, lines sent by the program to stdout are timestamped and teed to this sink as well. See SINK SPECIFICATIONS for more information about specifying sinks.

`-n --notify`

Enables exit notification for the command. If specified, the user is notified of command exit. Commend exit is always logged to the error sink. `--notify` causes an message to be sent to stderr that includes the command name, the exit status and, if exit was due to an uncaught signal, the signal that caused the process to exit. If the environment variable `DISPLAY` is defined and the program PopUp is in the path, it is run to produce an X11 pop up message as well.

SINK SPECIFICATIONS

A sink is specified by a string containing two fields separated by a colon: *`sinktype:sinkname`* The left field is the sink type, while the right field is a sink name interpreted within the sink type. For example, the sink specification: `file:/user/fox/output.log` specifies the sink is a file, and that the file is `/user/fox/output.log`.

Sink types that are currently supporteed include:

`file`

A file. The sink name is the name of the file.

If all this seems like a bit much to specify a file, recall that this is version 1.0 of the program and future versions will probably support other sink types.

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 sclcient 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.

RunNumber

> This variable contains he number of the current run. If not yet known, it has the value >Unknown%lt; 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.

EXAMPLES

The sample below starts out sclclient taking data from spdaq22 and feeding it to a tclserver on u6pc2 at port 2700:

Example 1. Starting sclclient

```
/usr/opt/daq/bin/sclclient -s tcp://spdaq22:2602/ -h u6pc2 -s 2700
```

SEE ALSO

ScalerDisplay(1tcl), tclserver(1)

II. 1epics

chanlog

Name

`chanlog` — Write a set of channels to file

Synopsis

chanlog ***`channelfile`*** ***`outfile`***

DESCRIPTION

Logs the names, values and units of a set of EPICS channels to file. In the command invocation, *`channelfile`* is a file that contains the names of the channels to lookup and log. If *`channelfile`* is `-`, then the channels are taken from stdin. *`outfile`* is the name of a file to which the channels will be logged. If *`outfile`* is `-`, the output is sent to stdout.

The channel file is an ordinary text file that contains the names of EPICS channels, each channel is separated by whitespace (note that a newline counts as whitespace). No comments are allowed, in contrast to the form of input file expected by e.g. epicsdisplay. The following is a sample input file:

```
Z001DV Z002DH
Z003DV
Z004QA

Z012QB
```

This file will output the values and units for 5 channels: `Z001DV, Z002DH, Z003DV, Z004QA` and `Z004QA`.

Output will consist of a line of text for each channel in the input file. Each line will consist of a set of white-space separated components. The first component is the name of a channel followed by a colon. The second element is the channel value. The final element is the channel engineering units.

If the value of a channel cannot be gotten, the remainder of the line following the channel name will be the error message generated for that channel for example: `Z001DH: ca_pend_io failed User specified timeout on IO operation expired`.

EXAMPLES

All of the examples assume that a file named `channels.txt` has been created that contains the set of channel names of interest, and that chanlog is in your path.

Example 1. Viewing a set of channel values interactively

```
chanlog channels.txt -
```

Example 2. Writing a set of channels to a file

```
chanlog channels.txt somefile.out
```

Example 3. Appending a set of channels to a file

```
chanlog channels.txt - >>somefile.out
```

Example 4. Piping a set of channels to a program for processing

```
chanlog channels.txt - | someprogram
```

DEPENDENCIES

- Epics software must be installed and located by the installer
- The Epics utility caRepeater should be in your path. This is in the architecture specific `bin` directory of the Epics installation.

controlpush

Name

`controlpush` — Push epics data into a Tcl Server (e.g. production readout).

Synopsis

controlpush ***`[options...] files...`***

OPTIONS

Options that govern the actions of the program are:

`-h --help`

Writes brief program help to stdout.

`-V --version`

Writes the program version number to stdout.

`-p`*`portnum`* `--port=`*`portnum`*

`portnum` is the number of the port to which controlpush will attempt to connect. See OPERATION below to know what controlpush does once it has connected

`-i`*`interval`* `--interval=`*`intervale`*

Specifies the refresh *`interval`* in seconds.

`-n`*`host`* `--node=`*`host`*

Specifies the node to which the **controlserver** will connect.

`-a` (`--authorize`)

Specifies that controlpush is connecting to a Tcl server that requires user authentication. This should be used when controlpush is connecting to a tclserver at nscldaq 8.1 or later.

OPERATION

controlpush reads its configuration file and establishes epics lists that will update automatically via epics notifications. controlpush next connects to its server and periodically sends a batch of Tcl **set** commands to maintain the following Tcl global arrays:

`EPICS_DATA(channelName)`

Is the most recently received update for `channelName`.

`EPICS_UNITS(channelName)`

Are the engineering units for `channelName`

`EPICS_UPDATED(channelName)`

Is the timestamp of the alst update received for `channelName`.

controlpush continues to run until the socket that is connected to its server is closed, at which point it exits.

CONFIGURATION FILES

controlpush configuration files are simple text files. Each line is either a or a channel.

Comments consist of blank lines or lines whose first non-whitepsace is the pound (`#`) character.

Channel lines, have as their first non-whitespace word, the name of an EPICS channel. All text following the channel name on a channel line is ignored by this program. Note, however that other software (notably epicsdisplay) may parse additional characters for their own purposes.

DEPENDENCIES

- The installer must be able to locate the EPICS installation.
- The caRepeater, normally in the architecture specific subdirectory of the EPICS installation should be in the user's `PATH` environment variable.

III. 1tcl

DaqPortManager

Name

`TCP/IP service port manager.` — Manage TCP/IP service ports and advertise their allocations

Synopsis

tclsh DaqPortManager ***`options...`***

DESCRIPTION

DaqPortManager is a persistent server that should be run at system startup time. The program manages a block of TCP/IP ports. Clients connect to the server to request ports as well as to ask the server to return the current port assignments.

Command options (see OPTIONS) describe how th eprogram starts up. The command options allow the user to define the block of ports that will be managed, to determine which port the port manager itself will listen on for connections and to define where the software will writ its log file. Several other files are maintained by the program (see FILES).

OPTIONS

`-ports` *`range`*

Defines the range of ports that will be managed by the port manager program. This should be specified as a pair of integer number separated by a `-` (dash). e.g. `30001-31000`. If not provided defaults to `30001-31000`.

`-listen` *`port`*

Defines the port on which the port manager itself will listen for connections. This defaults to `30000`. The port manager is itself a server. Most of the software that connects to it assumes the default port will be used to communicate with it, therefore, changing this value will most likely require other changes elsewhere in the NSCLDAQ software

`-log` *`logfile`*

The `-log` switch is followed by a path to a file that will be used to log the server's actions. The server logs connections, port allocations, port allocation failures, port releases and illegal requests by clients. The default log file is `/var/log/nscldaq/portmanager.log`

`-usagefile` *`usagefilename`*

The `-usagefile` switch is followed by a path to a file that will be used to hold the instantaneous port usage. This file can be examined to show the current port usage. Each line in this file is a three element TCL list of port, application and user for an allocated port. If the `-usagefile` switch is not provided, the server defaults to `/var/tmp/daqportmgr/ports.txt`.

PROTOCOL

The server protocol is quite simple. Once connected, two transactions are recognized. Each transaction is requested by the client and fulfilled by the server. Requests are a single line (terminated with a `\n`) and cann be one of the following:

GIMME ***`appname user`***

GIMME requests the server allocate a port. When allocated, the server will believe that the port is allocated to the application named appname run by the user user. If successful, the server replies:

OK portnum

Where the OK indicates success and the portnum is the number of the port allocated.

FAIL reason

Indicates a failure where reason is a TCL quoted item describing why the port could not be allocated.

Once a client has allocated a port it must hold a persistent connection to the server. When a client owning ports drops this connection its ports are freed by the server. While possibly inconvenient, this protocol prevents clients from accidently holding on to ports past exit.

LIST

Requests that the server list port usage. The server will issue a multiline reply to this request. The first line will be of the form:

OK n

Where n is the number of ports that are currently allocated. The n lines that follow each consists of a 3 element TCL list containing in order, the port number, the application name and the name of the user that is running the application. Note that if there are multiple instances of the same application run by the same user, They will be qualified by appending an underscore and an application instance num? ber.

EXAMPLES

```
tclsh8.4 /usr/opt/daq/current/bin/DaqPortManager -ports 31000-40000 &
```

Starts the port manager listening on port 30000 managing ports in the range 31000 through 40000. The port manager is run in the background as a daemon process.

FILES

The server maintains:

```
/var/run/nscldaq/daqportmgr.pid  - contains the port manager#033;s process id.
/var/log/nscldaq/portmanager.log - Default logfile
/var/tmp/daqportmgr/listen.port  - Contains the listen port for the server.
/var/tmp/daqportmgr/ports.txt    - Default port usage file.
```

KNOWN DEFECTS

The server should be able to default to listen on the service daqportmgr in /etc/services if present before falling back on port 30000

The !uniquification! of the application name for the LIST command may vary from query to query as applications drop out.

Only connections to/from "localhost" are considered local. The server does not bother to determine if any other sources are local even though they may be.

If multiple instances of the port manager are run, only last one started will be listed in `/var/run/nscldaq/daqportmgr.pid`

SEE ALSO

CPortManager(3daq) CPortManagerException(3daq) portAllocator(3tcl)

dvdburn

Name

`dvdburn` — Command line tool to burn NSCLDAQ data DVDs.

Synopsis

dvdburn [firstrun [lastrun]]

DESCRIPTION

Burns the data associated with a set of runs to DVD. The DvdBurner package is used to do the burn, so all restrictions and dependencies for that package apply.

If no parameters are supplied, all runs are burned. If *`firstrun`* is supplied, all runs with run numbers at least *`firstrun`* are burned. Finally if both *`firstrun`* and *`lastrun`* are supplied, all run numbers that are at least *`firstrun`* and at most *`lastrun`* are burned.

Dependencies

See the DvdBurner manpage.

burngui

Name

`burngui` — Graphical front end to dvdburn

Synopsis

burngui

DESCRIPTION

This is a graphical user interface that runs the dvdbur program. This program should be reasonably easy to use. If not complain and more complete documentation will be written.

Dependencies.

See the DvdBurner package. All restrictions apply.

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).

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:

```
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:

```
display_single Tabname channelname
```

Where `Tabname` is the Tab name of a scaler page and c `hannelname` is the name of a scaler channel.

Ratio lines are defined via the display_ratio command:

```
display_ratio  Tabname numerator_chanel denominator_channel
```

Blank lines are defined via the blank command:

```
blank Tabname
```

Strip charts are defined using the stripparam stripratio and stripconfig commands. The format of the stripparam command is:

```
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).

```
stripratio numerator denominator
```

The ratio of the rates in the two channels numerator and denominator are added to the seet 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 intially 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 `#RRRRGGGGBBBB` 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
```

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:

```
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 commnds 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:

```
stripconfig -timeaxis 3600;   # Trick to get the widget defined....
$stripchartWidget grid configure -hide 0
$stripchartWidget legend configure -position left
```

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 contaning 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)

`ScalerDletaTime`

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:

```
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:

```
               set  scalerWin [toplevel .scaler]
               source /usr/opt/daq/current/Scripts/scaler.tcl
```

Creates the scaler display in a new top level widget called .scaler

`maxStripPoints`

Sets a limit on the number of points the strip chart recorder data series can have. If this is zero (the default), there is no limit. If non zero, when more than `maxStripPoints` time intervals have been added to the data series, every other point is removed from the first 1/2 of the time. This means that:

1. The most recent 1/2 of data is always at full resolution
2. The oldest data is shown at progressively poorer time resolution as successive data trims are performed.

EXAMPLE(S)

```
#
#   Define the scaler channels:
#
#   These can be in any order, IÃ¢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 Ã¢channelÃ¢  - to define a chanel name/buffer position correspondence
#             Ã¢pageÃ¢     - To define a scaler page.
#             Ã¢display_singleÃ¢ - To define a single scaler line in a page.
#

#
channel gas.PIN.cfd       0
channel gas.qA.cfd       16
channel gas.qB.cfd       17
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.
#  IÃ¢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 donÃ¢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

#  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.gC.cfd      gas.qD.cfd
display_single GasCell gas.Ge.cfd
display_ratio  GasCell gas.Ge.cfd      gas.Ge.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)

tclserver

Name

`tclserver` — Start a Tcl Server.

Synopsis

tclserver [`-p`*`port`* `-userauth`]

DESCRIPTION

Starts a Tcl server. The server will listen for connections. Authorized connections will be allowed to ’poke’ Tcl commands to the server which will execute them and return the result to the client.

OPTIONS

`-p`*`port`*

Specifies the port on which the server will listen.

- If omitted this defaults to 2048.
- If *`port`* is the text string `Managed`, the daq port manager is contacted to allocated a free port on which to listen.
- If *`port`* is an integer, it is taken as th eport on which to listen.

`-userauth`

If present, the client must, in addition to being authorized via the **serverauth** command send as the first line of text the name of the user running the client. This username must match the name of the uyser that is running the server.

This is intended to reduce the chances that more than one user on the same system might cause problems by talking with each other’s tclservers.

serverauth

Name

`serverauth` — Control tcl server authorization.

Synopsis

serverauth add *`hostorip`*

serverauth remove *`hostorip`*

serverauth list

DESCRIPTION

Manipulates the server authorization database. The server authorization database determines which nodes are allowed to connect to the tcl server. This command lives in the tclserver itself. Usually a startup script or GUI element projected by the server sets up the appropriate authorization lists.

serverauth is an ensemble of commands that has the following ensemble commands:

serverauth add *`hostorip`*

Adds *`hostorip`* to the set of hosts that are allowed to connect to this instance of the server. *`hostorip`* can be either a DNS hostname like `spdaq22.nscl.msu.edu` or it can be a dotted IP address like: `35.8.35.123`.

serverauth remove *`hostorip`*

Removes the host *`hostorip`* from the set of systems that are authorized to connect as clients to the tcl server. *`hostorip`* is a DNS name or dotted IP, as for the **serverauth add** command. Matching is based on the IP address so the command should work just fine in the presence of e.g. CNAMEs.

serverauth list

Lists the systems that are authorized to connect. The command returns as a result a Tcl list. Each element of the list is a pair. The first element of the pair is the hostname of a host authorized to connect (actually a hostorip provided). The second element of the list is the IP address of the associated host.

IV. 3daq

CopyrrightNotice

Name

`CopyrightNotice` — Generate license/author credits.

Synopsis

```
#include <CopyrightNotice.h>



 class CopyrightNotice {
  static void Notice(std::ostream& out, const char* program, const char* version, const char* year);
  static void AuthorCredit(std::ostream& out, char* program, ...);
}
```

Description

The `CopyrightNotice` class provides static methods for generating license notices and copyright strings when programs start.

Public member functions

```
  static void Notice(std::ostream& out, const char* program, const char* version, const char
```

Outputs a copyright notice to the output stream *`out`*. *`program`* is the name of the program. *`version`* is a version string, and *`year`* is the copyright year.

```
  static void AuthorCredit(std::ostream& out, char* program, ...);
```

Outputs credit for the authors. The *`out`* and *`program`* parameters have the same meaning as for `Notice`. These parameters are followed by a null terminated variable length list of arguments that must all be const char* pointer to names of the authors.

EXAMPLES

Create a copyright notice on stderr. This is from the `main` or other function that has access to the `argv` array:

Example 1. Creating a coypright notice on stderr

```
CopyrightNotice::Notice(cerr, argv[0], "2.1", "2004");
```

Create an author credit on stderr for the authors Ron and Kanayo:

Example 2. Creating an author credit on stderr

```
CopyrightNotice::AuthorCredit(cerr, argv[0], "Kanayo", "Ron", NULL);
```

cvt

Name

`cvt` — Integer byte order conversions

Synopsis

```
#include <cvt.h>
```

`void` **`makecvtblock`**`( uint32_t` *`lsig`* `,  uint16_t` *`ssig`* `,  DaqConversion*` *`conversion`* `);`

`bool` **`hostsameasforeign`**`( DaqConversion*` *`conversion`* `);`

`uint32_t` **`ftohl`**`( DaqConversion*` *`convertdata`* `,  uint32_t` *`datum`* `);`

`uint16_t` **`ftohs`**`( DaqConversion*` *`convertdata`* `,  uint1y_t` *`datum`* `);`

`uint32_t>` **`htofl`**`( DaqConversion*` *`convertdata`* `,  uint32_t` *`datum`* `);`

`uint16_t` **`htofs`**`( DaqConversion*` *`convertdata`* `,  uint16_t` *`datum`* `);`

Description

This library provides a suite of functions that do integer conversions between the native integer formats of different systems. Typically systems are classified as *big-endian* or *little-endian*.

Big-endian systems store their data with the most significant bytes of a multi-byte integer in lower addresses, while little-endian systems least significant bytes in lower addresses. This library uses *signature* data in the data to determine the byte ordering of foreign host (the host that created the data), and how it differs from the local byte ordering.

Signature data are data that have a known value, stored in the system's native byte order. Analyzing these values byte by byte allows the software to determine the byte order of the foreign system. The code defines the values: `CVT_WORDSIGNATURE` and `CVT_LONGSIGNATURE` respectively to be uint16_t and uint32_t signatures respectively. Separate 16 and 32 bit signatures allows the library to deal with any pathalogical systems that may have word orders that differ from byte ordering.

Public functions

```
void makecvtblock( uint32_t lsig ,  uint16_t ssig ,  DaqConversion*
conversion );
```

Creates a conversion block from the foreign system signatures you've extracted from the data. Note that NSCLDAQ control data includes byte and long signatures you can use for this purpose. *`lsig`* and *`ssig`* are the 32 and 16 bit signatures respectively.

`conversion` must point to a DaqConversion structure that will be over-written with the conversion block that describes how to convert from the byte order described by *`lsig`* and *`ssig`* to the host's byte order.

```
bool hostsameasforeign( DaqConversion* conversion );
```

Determines if the conversion block pointed to by *`conversion`* describes a system with the same byte order as the host system.

```
uint32_t ftohl( DaqConversion* convertdata ,  uint32_t datum );
```

Converts a 32 bit value *`datum`* from the byte order of the foreign host used to produce the DaqConversion structer pointed to by *`convertdata`* to the local host's byte order. The converted value is returned.

```
uint16_t ftohs( DaqConversion* convertdata ,  uint1y_t datum );
```

Converts a 16 bit integer *`datum`* from the byte order described by the system used to produce the DaqConversion block pointed to by *`convertdata`* into the local system's byte order and returns that 16 bit integer.

```
uint32_t> htofl( DaqConversion* convertdata ,  uint32_t datum );
```

Converts the 32 bit integer *`datum`* from the local host's native byte order to the byte order described by the foreign host used to create the DaqConversion conversion block *`convertdata`*, and returns the converted value.

```
uint16_t htofs( DaqConversion* convertdata ,  uint16_t datum );
```

Converts the 16 bit integer *`datum`* from the local host's native byte order to the byte order described by the foreign host used to create the DaqConversion conversion block *`convertdata`*, and returns the converted value.

Types and public data

`CVT_WORDSIGNATURE` is a 16 bit byte order signature you will need to supply in any data you originate, and `CVT_LONGSIGNATURE` is a 32 bit byte order signature you will need to supply in any data you originate.

CException

Name

`CException` — Abstract base class for the exception class hierarchy.

Synopsis

```
#include <Exception.h>



 class CException {
  CException(const char* pszAction);
  CException(const std::string& rsAction);
  virtual const  const char* ReasonText();
  virtual const Int_t ReasonCode();
  const const char* WasDoing();
}
```

Description

This class is the abstract base class of the exception class hierarchy. The class hierarchy provides textual exception descriptions (that can be displayed to a user), as well as support for a numerical code that can be easily processed by software. The textual description is composed by the `ReasonText` function in an exception specific way. Usually the resulting message includes a description of the message along with context information held by this base class and retrieved via `WasDoing`.

Public member functions

```
  CException(const char* pszAction);
  CException(const std::string& rsAction);
```

Both of the constructors described above save their parameter as the context specific part of the description. This string can be retrieved via `WasDoing`. Normally the string should give some indication of what the program was doing when the exception was thrown.

```
  virtual const  const char* ReasonText();
```

A virtual function that is supposed to return the reason the exception was thrown. For the base class this is the string `Unspecified Exception`.

```
  virtual const Int_t ReasonCode ();
```

Returns a number that corresponds to the exception reason within a specific class of exceptions (e.g. for a `CErrnoException` this would be the value of `errno` at the time the exception was constructed). For the base class this will always be -1.

```
  const const char* WasDoing();
```

Returns the context string that was passed to our constructor. This will normally be called by a derived class's `ReasonText` function when constructing the text.

CErrnoException

Name

`CErrnoException` — Exceptions that wrap the Unix `errno`

Synopsis

```
#include <ErrnoException.h>



 class CErrnoException {
  CErrnoException(const pszAction);
  CErrnoException(const std::string& rsAction);
  virtual   const const char* ReasonText();
  virtual const Int_t ReasonCode();
}
```

Description

This exception class wraps the Unix `errno` global variable. errno is used by most Unix system calls to provide detailed status information. Constructing this class saves a copy of the `errno` variable. `ReasonCode` will return the saved errno value. `ReasonText` will construct a string that includes the text associated with the `errno`.

Public member functions

```
  CErrnoException(const pszAction);
  CErrnoException(const std::string& rsAction);
```

These constructors save the value of the `errno` variable and construct the base class using their parameter as the exception context string (returned by `WasDoing`).

```
  virtual   const const char* ReasonText();
```

Returns the `sterror` applied to the saved `errno`.

```
  virtual const Int_t ReasonCode();
```

Returns the saved `errno` variable value.

CRangeError

Name

`CRangeError` — Reports and exception for a value out of allowed range.

Synopsis

```
#include <CRangeError.h>



 class CRangeError {
  CRangeError(Int_t nLow, Int_t nHigh, Int_t nRequested, const char* pDoing);
  CRangeError(Int_t nLow, Int_t nHigh, Int_t nRequested, const std::string& rDoing);
  virtual   const const char* ReasonText();
  virtual const Int_t ReasonCode();
}
```

Description

Provides an exception that can be used to report parameter range errors (normally for integer parameters). The exception will construct a reason text that includes the erroneous value as well as the valid range of values.

Public member functions

```
  CRangeError(Int_t nLow, Int_t nHigh, Int_t nRequested, const char* pDoing);
  CRangeError(Int_t nLow, Int_t nHigh, Int_t nRequested, const std::string& rDoing);
```

Creates a `CRangeError` that can be thrown. The only difference between the two constructors is the way in which the context information is passed. In the first, by a standard C null terminated string. In the second by a C++ `std::string` object.

The remaining parameters describe the actual exceptional condition:

`nLow`

Is the low end of the allowed range that was violated.

`nHigh`

Is the high end of the allowed range that was violated.

`nRequested`

Is the value that violated the range described by *`nLow`* and *`nHigh`*

```
  virtual   const const char* ReasonText();
```

Returns the reason for the exception. This will be a string of the form: `Range error: %v is outside the range [%l..%h]` where %v, %l %h are shorthands for the value, low and high limits respectively.

```
  virtual const Int_t ReasonCode();
```

Returns one of the two possible reasons for the exception:

`CRangeError`::`knTooHigh`

The value requested was above the upper limit of the range.

`CRangeError`::`knTooLow`

The value requested was below the lower limit of the range.

Note that if the requested value provided to the constuctor does not satisfy either of these relationships, invoking this member function will result in an assertion failure.

CStateException

Name

`CStateException` — Exception for invalid state transitions.

Synopsis

```
#include <CStateException.h>



 class CStateException {
  CStateException(const char* state, const char* allowedStates, const char* pAction);
  virtual const const char* ReasonText();
  virtual const Int_t ReasonCode();
}
```

Description

Many programming problems can be solved by programming a *state machine*. A state machine, or finite state automaton, simply put, is an entity that has a well defined set of states it can live in. Input stimulii may trigger well defined transitions from one state to another state. For example the *begin* stimulus can force a run-state machine in the *halted* state to make a transition to the *active* state.

When programming a state machine the programmer must take into account exceptional cases where stimulii that are not allowed for the current state are received or, in some cases, errors in implementing the state machine cause the state machine to enter a state that is not legal.

The `CStateException` provides a class that can be instantiated and then thrown in the event one of these cases occurs.

Public member functions

```
CStateException(const char* state, const char* allowedStates, const char* pAction);
```

Creates a `CStateException` that can be thrown.

`state`

The state attempted. This can be a state attemped via an illegal transition or it could be an illegal state actually entered as a result of an error in programming.

`allowedStates`

The allowed states at this time.

`pAction`

The context in which the error was detected and thrown.

```
virtual const const char* ReasonText();
```

As usual, this returns the textual reason for the failure. This will be a string of the form: `Invalid object state when: %c relevant erroneous state: %s relevant allowed state: %a` Where `%c` is the context text supplied at construction time. `%s` is the state that caused the error. `%a` are the allowed states prior to this error.

```
virtual const Int_t ReasonCode();
```

The error code for this exception type is always -1.

CStreamIOError

Name

`CStreamIOError` — I/O error on a C++ stream.

Synopsis

```
#include <CStreamIOError.h>



 class CStreamIOError {
   CStreamIOError(IoStreamConditions eReason, const char* pDoing, std::ios& rStream);
  CStreamIOError(IoStreamConditions eReason, const std::string& rDoing, std::ios& rStream);
  virtual const const char* ReasonText();
  virtual const Int_t ReasonCode();
  const IoStreamConditions getReason();
  std::ios& getStream();
  const const char* getErrorMessage();
}
```

Description

Captures an error condition reading from or writing to a C++ stream (<iostream>). The errors are captured via an enumeration `CStreamIOError`::IoStreamConditions. See "Types and Pulbic Data" below for more information about the values this can have.

Public member functions

```
   CStreamIOError(IoStreamConditions eReason, const char* pDoing, std::ios& rStream);
  CStreamIOError(IoStreamConditions eReason, const std::string& rDoing, std::ios& rStream);
```

Constructs an exception object that can later be thrown. *`eReason`* is the `CStreamIOError`::IoStreamConditions value that describes the error being reported. See "Types and Public Data" for more information about allowed values.

`pDoing` or *`rDoing`* pass in the eror context string that can be reported back to the catcher.

`rStream` is a reference to the stream on which the error was detected.

```
  virtual const const char* ReasonText();
```

Returns the reason text associated with the error. This will be a string that describes the value of the *eReason* used to construct the exception object.

```
virtual const Int_t ReasonCode();
```

Returns the error condition cast to an int. You can also use the class specific `getReason` described below.

```
const IoStreamConditions getReason();
```

Returns the reason the error was thrown.

```
std::ios& getStream();
```

Returns a reference to the failing stream. Note that if the catch handler is far enough up the call stack the stream may have been destroyed already. consider something as simple as:

```
try {
    ifstream something("somefile");
    somefunction(something);       // can throw...
    ...
}
catch (CStreamIOError& error) {
...
}
```

Due to the semantics of exception handling, the stream reference that would be returned would be to an already destroyed stream.

```
const const char* getErrorMessage();
```

Returns a full error message string that includes the reason for the error and the context string as well.

Types and public data

The type: `CStreamIOError::IoStreamConditions` is used to capture error information. Calling `ReasonCode` and then casting that value to an `CStreamIOError::IoStreamConditions` will give you one of the following values:

`CStreamIOError::EndFile`

An end file condition has been detected.

`CStreamIOError::BadSet`

The bad bit in the stream's I/O status mask is set.

`CStreamIOError::FailSet`

The fail bit is set in the stream's I/O status mask.

CURIFormatException

Name

`CURIFormatException` — Report errors in universal resource identifiers (uri)s.

Synopsis

```
#include <uriFormatException.h>



 class CURIFormatException {
  CURIFormatException(std::string uri, const char* file, int line);
  CURIFormatException(std::string uri, std::string port, const char* file, int line);
  CURIFormatException(std::string uri, const char* host, const char* file, int line);
  CURIFormatException(const CURIFormatException& rhs);
  virtual ~CURIFormatException();
  CURIFormatException& operator=(const CURIFormatException& rhs);
   const int operator==(const CURIFormatException& rhs);
  const int operator!=(const CURIFormatException& rhs);
  virtual const const char* ReasonText();
  virtual const Int_t ReasonCode();
}
```

Description

This exception is a subclass of `CException`. It is specialized to report exceptions encountered parsing Uniform Resource Identifiers (uris). The reason text is human readable, but the reason code is always `-1`.

Typcially the exception is used to either present an error message to the user. Once the exception is caught, tyipical program action is to either prompt the user to correct the faulty uri or to exit, depending on the level of interactivity assumed by the application.

Public member functions

Constructors. A rich set of constructors is provided to build the exception according to the actual error. In all the constructors above, the current action is constructed from the *`file`* and *`line`* parameters which are the names of the constructing file and line number at which the construction occured. You should use the preprocessor `__FILE__` and `__LINE__` macros for the *`file`* and *`line`* parameters unless you have some special needs.

`CURIFormatException`std::string*`uri`*const char**`file`*int*`line`*

Constructs the exception when the URI *`uri`* just can't be parsed.

`CURIFormatException`std::string*`uri`*std::string*`port`*const char**`file`*int*`line`*

Constructs the exception when the URI *`uri`* has a string *`port`* which should be a port number but can't be properly processed as a port.

`CURIFormatException`std::string*`uri`*const char**`host`*const char**`file`*int*`line`*

This constructor creates the exception objects for the URI *`uri`* when the *`host`* should be a host name but cannot be resolved to an IP address. The assumption is that the URI being constructed will actually be used.

`CURIFormatException`const CURIFormatException&*`rhs`*

Copy Construction

Canonical operations. The `CURIFormatException` class supports assignment and comparisons for equality and inequality such that two equal exceptions where created in the same place for the same reason.

`CException` interface.

```
  virtual const const char* ReasonText();
```

Returns a human readable exception string.

```
  virtual const Int_t ReasonCode();
```

Returns `-1`.

CMonitorException

Name

`CMonitorException` — Exceptions for synchronization class abuse.

Synopsis

```
#include <MonitorException.h>
```

```
 class CMonitorException {
  MonitorException(int correctOwner, int actualOwner, const char* file, const char* line);
  MonitorException(const MonitorException& rhs);
  virtual ~MonitorException();
  MonitorException& operator=(const MonitorException& rhs);
  const int operator==(const MonitorException& rhs);
  const int operator!=(const MonitorException& rhs);
  virtual const const char* ReasonText();
  virtual const Int_t ReasonCode();
}
```

Description

The NSCL DAQ base software includes support for an object model wrapped around the pthreads threading subsystem. This allows NSCL DAQ software to be written that can take advantage of multiple threads of execution. The threading library is described in the chapter; NSCL DAQ Thread Library. That chapter points to additional reference pages that provide reference information for the classes themselves.

Any non-trivial threaded application must synchronize at critical points in its operation in order to maintain consistent views of non-primitive data. The NSCL DAQ thread library provides a `Synchronizable` class and associated `SyncGuard` class to support this synchronization. These classes attempt to detect abuse of the synchronization model they support. The report such abuse by throwing `CMonitorException` objects.

`CMonitorException` is derived from the CException class, and therefore supports and implements all of its virtual members.

Public member functions

Constructors. In the description of the methods below; *`correctOwner`* is a thread id that represents the thread that should be holding ownership of a synchronizable object. *`actualOwner`* is the thread id of the thread that actually does own the object, and *`file`* and *`line`* are ordinarily the values of the `__FILE__` and `__LINE__` macros at the time the the exception is constructed.

`MonitorException`int*`correctOwner`*int*`actualOwner`*const char**`file`*const char**`line`*

The normal constructor for this object. The exception will report that the ownership of the synchronizable object should have been the currently executing thread but was a different thread.

`MonitorException`const MonitorException&*`rhs`*

Copy construction. The object constructed will be a duplcate of the *`rhs`* object.

Canonical methods. The `CMonitorException` class supports assignment, equality comparison and inequality comparison. Assignment creates a duplicate of the right hand side of the assignment operator. Two `CMonitorException` objects are equal if the Reason text they would generate are equal. Inequality is defined as the logical inverse of equality.

Implementation of the `CException` interface.

```
virtual const const char* ReasonText();
```

Returns the reason for the exception. This is intended to be human readable text that describes the error as well as where an why it was thrown.

```
virtual const Int_t ReasonCode();
```

Normally this returns an error code that can be processed by computer code. At present, there's only one reason for the exception to be thrown, so this always returns `-1`.

CAuthenticator

Name

`CAuthenticator` — Abstract base authenticator class.

Synopsis

```
#include <Authenticator.h>



 class CAuthenticator {
  CAuthenticator();
  CAuthenticator(const CAuthenticator& rhs);
  virtual ~CAuthenticator();
  CAuthenticator& operator=(const CAuthenticator& rhs);
  virtual  = 0 Bool_t Authenticate(CInteractor& interactor);
  protected: std::string GetLine(CInteractor& interactor);
}
```

Description

`CAuthenticator` is the base class for all authenticators. You can use any of the authenticators described in these reference pages, or you can create your own by deriving a new authenticator from this base class.

The class provides a public interface and protected services for derived classes. These will be described in "Member functions" below.

Member functions

`CAuthenticator`(void);

Default construtor. This is supplied so the compiler won't complain if derived class constructors don't explcitly chain to their base class constructor.

`CAuthenticator`const CAuthenticator&*`rhs`*

Copy constructor this is supplied so that derived classes can implement a copy constructor. In the future, if additional services are added to the base class that require state, this constructor can be chained to by derived classes to ensure that the appropriate deep copy is done.

```
virtual ~CAuthenticator();
```

Destructor is provide as virtual to ensure that destructors automatically chain up the class hierarchy.

```
CAuthenticator& operator=(const CAuthenticator& rhs);
```

Assignment operator is supplied for much the same reason as the copy constructor.

```
virtual  = 0 Bool_t Authenticate(CInteractor& interactor);
```

`Authenticate` is the interface that derived classes must implement. The method must obtain credentials from the entity that want service via the *`interactor`* object supplied. If the entity supplies authorized credentials, the function should return `kfTRUE`. If the entity supplies bad or unauthorized credentials, `kfFALSE` should be returned.

```
protected: std::string GetLine(CInteractor& interactor);
```

This protected service reads a complete line of text from the *`interactor`*. The newline at the end of the input line is absorbed from the interactor but silently discarded.

Types and public data

The `CInteractor` object is an object that is intended to obtain credentials from some source. Other sections of this manpage describe interactors, both their abstract base class (`CInteractor`), and derived concrete classes.

The Security chapter of the full documentation manual describes the entire authentication package and how to use it as well as providing a brief listing of the various authenticators and interactors that are currently implemented.

CPasswordCheck

Name

`CPasswordCheck` — Authenticate against a stored password.

Synopsis

```
#include <PasswordCheck.h>


 class CPasswordCheck {
  CPasswordCheck(const std::string& password, const std::string prompt = std::string(""), Bool_t WithP
  CPasswordCheck(const CPasswordCheck& rhs);
  ~CPasswordCheck();
  CPasswordCheck& operator=(const CPassswordCheck& rhs);
  const std::string getPassword();
  const std::string getPromptString();
  const Bool_t getWithPrompt();
  void setPassword(const std::string password);
  void setPromptString(const std::string prompt);
  virtual Bool_t Authenticate(CInteractor& interactor);
  void DisablePrompt();
  void EnablePrompt();
}
```

Description

Authenticators of type `CPasswordCheck` assume that the credential is a string that must exactly match some passphrase string. The passphrase can be dynamically modified if desired, according to the authentication policy of the application.

The passphase credential is gotten via an interactor. It is possible to control whether or not the credential is prompted for or not. If prompted, it is also possible to control the prompt string.

Public member functions

`CPasswordCheck`const std::string&*`password`*const std::string*`prompt = std::string("")`*Bool_t*`WithPrompt = kfFALSE`*

Constructor for the password checking authenticator. The *`password`* parameter provides the initial password which can be changed by later calling the `setPassword` member function. The *`prompt`* and

`WithPrompt` parameters control prompting on the interactor. If *`WithPrompt`* is `kfFALSE`, no prompting will be done. If *`WithPrompt`* is `kfTRUE`, *`prompt`* will be used as the prompt string.

`CPasswordCheck`const CPasswordCheck&*`rhs`*

Does a copy construction of a `CPasswordCheck` object. This is a deep copy.

```
~CPasswordCheck();
```

Cleans up any storage or other resources allocated by the object.

```
CPasswordCheck& operator=(const CPasssswordCheck& rhs);
```

Assignment operator. The *`rhs`* object is the source (right hand side) of the assignement operator. When complete, the object is an identical copy of the *`rhs`*.

```
const std::string getPassword();
```

Returns the current passphrase string.

```
const std::string getPromptString();
```

Returns the current prompt string.

```
const Bool_t getWithPrompt();
```

Returns `kfTRUE` if prompting for the passphrase is enabled. `kfFALSE` otherwise.

```
void setPassword(const std::string password);
```

Changes the passphrase. Future calls to `Authenticate` must obtain a credential string that matches the new passphrase *`password`*

```
void setPromptString(const std::string prompt);
```

Providses a new prompt string: *`prompt`*. This only matters if prompting is enabled.

```
virtual Bool_t Authenticate(CInteractor& interactor);
```

Authenticates, using *`interactor`* to obtain the candidate passphrase from the entity requesting service.

```
void DisablePrompt();
```

Disables passphrase prompting.

```
void EnablePrompt();
```

Enables passphrase prompting. The most recently set value of the prompt string will be used with the interactor to prompt for a password.

SEE ALSO

CAuthenticator(3daq), CInteractor(3daq)

CUnixUserCheck

Name

`CUnixUserCheck` — Authenticate against a unix user name and password.

Synopsis

```
#include <UnixUserCheck.h>
```

```
 class CUnixUserCheck {
  CUnixUserCheck();
  CUnixUserCheck(const std::string& usernamePrompt, const std::string& passwordPrompt, Bool_t promptUs
  CUnixUserCheck(const CUnixUserCheck& rhs);
  ~CUnixUserCheck();
  CUnixUserCheck& operator=(const CUnixUserCheck& rhs);
  const std::string getUserPrompt();
  const std::string getPasswordPrompt();
  const Bool_t getPromptUser();
  const Bool_t getPromptPassword();
  virtual Bool_t Authenticate(CInteractor& interactor);
  void setPrompting(Bool_t fUserPrompt = kfTrue, Bool_t fPasswordPrompt = kfTrue);
  void SetUserPrompt(const std::string& rNewPrompt = std::string("Username: "));
  void SetPasswordPrompt(const std::string& rNewPrompt = std::string("Password: "));
}
```

Description

Authenticates any user whose encrypted password can be retrieved by `getpwnam`. The meaning of this may differ depending on whether or not NIS is used to do authentication, and may vary from operating system to operating system.

Public member functions

`CUnixUserCheck`(void);

Constructs a default `CUnixUserCheck` object. The object is construted with both username and password prompting enabled with suitable default prompt strings. You may modify these prompt strings after construction, as well as the prompting behavior. See below for more information.

`CUnixUserCheck`const std::string&*`usernamePrompt`*const std::string&*`passwordPrompt`*Bool_t*`promptUser`*Bool_t*`promptPasswo`*

Full bodied construction of a `CUnixUserCheck` object. The username and password prompts are provided by *`usernamePrompt`* and *`passwordPrompt`* respectively. While *`promptUser`* and *`promptPassword`* control which prompts are acutally emitted.

`CUnixUserCheck`const CUnixUserCheck&*`rhs`*

Copy constructor, creates a news object that is identical to *`rhs`*

```
~CUnixUserCheck();
```

Destroys the object and release any resources that were allocated by the object.

```
CUnixUserCheck& operator=(const CUnixUserCheck& rhs);
```

Assigns to the object from the *`rhs`* object.

```
const std::string getUserPrompt();
```

Returns the current value of the user name prompt string.

```
const std::string getPasswordPrompt();
```

Returns the current value of the password prompt string.

```
const Bool_t getPromptUser();
```

Returns `kfTRUE` if the user name prompt is enabled, and `kfFALSE` if the user name prompt is disabled.

```
const Bool_t getPromptPassword();
```

Returns `kfTRUE` if the password prompt is enabled, and `kfFALSE` if the username prompt is disabled.

```
virtual Bool_t Authenticate(CInteractor& interactor);
```

Authenticates the requestor using the *`interactor`* to obtain a username and password. Dependig on the state of the prompt flags, the username and password may be prompted for with prompt strings. See the constructors and the functions below for more on prompt strings.

```
void setPrompting(Bool_t fUserPrompt = kfTrue, Bool_t fPasswordPrompt = kfTrue);
```

If *`fUserPrompt`* is `kfTRUE`, the user name is prompted for. If `kfFALSE`, the username is read without prompting.

If *`fPasswordPrompt`* is `kfTRUE`, authentication will prompt for a password. If `kfFALSE`, the password will be read without prompting.

```
  void SetUserPrompt(const std::string& rNewPrompt = std::string("Username: "));
```

`rNewPrompt` replaces the user prompt string. If username prompting is enabled, this string will be used as the prompt in future authentications.

```
  void SetPasswordPrompt(const std::string& rNewPrompt = std::string("Password: "));
```

`rNewPrompt` replaces the current password prompt string. If password prompting is enabled, the new string will be used to prompt for passwords in future authentications.

CTclAccessListCheck

Name

`CTclAccessListCheck` — Authenticate against a Tcl List.

Synopsis

```
#include <TclAccessListCheck.h>



 class CTclAccessListCheck {
  CTclAccessListCheck (Tcl_Interp* pInterp, const std::string& rName);
  ~CTclAccessListCheck();
  const CTCLVariable* getVariable();
  const CTCLInterpreter* getInterpreter();
  virtual Bool_t Authenticate(CInteractor& rInteractor);
}
```

Description

This authenticator is intended to be used in conjunction with Tcl scripts and Tcl servers. The authenticator uses a Tcl variable that contains a list of values. The entity that desires service must present one of the values in that list as its credential.

Public member functions

`CTclAccessListCheck` Tcl_Interp**`pInterp`*const std::string&*`rName`*

Constructs a Tcl Access list authenticator. *`pInterp`* is the interpreter handle pointer. This will be wrapped in a `CTCLInterpreter` object. For more information about `CTCLInterpreter`, See the Tcl++ chapter and the CTCLInterpreter reference page.

The *`rName`* parameter is the name of the TCL variable that contains the list of allowed credentials. Note that the list does not yet need to have been created and, in fact, the variable *`rName`* need not yet exist. The variable and list only need to exist at authentication time.

```
~CTclAccessListCheck();
```

Releases any resources allocatd by the authenticator as it's destroyed.

```
const CTCLVariable* getVariable();
```

Returns a pointer to the `CTCLVariable` that wraps the Tcl variable. For more information about `CTCLVariable` objects, see Tcl++ chapter and the CTCLVariable reference page.

```
const CTCLInterpreter* getInterpreter();
```

Retrieves a pointer to the Tcl Interpreter wrapped in a `CTCLInterpreter` object.

```
virtual Bool_t Authenticate(CInteractor& rInteractor);
```

Uses the *`rInteractor`* to fetch a credential from the requestor. If the credential is a member of the list in the Tcl variable with which the authenticator was constructed, this method will return `kfTRUE` If not, `kfFALSE`

Types and public data

`CTCLInterpreter` and `CTCLVariable` are objects that are part of the Tcl++ library. See the Tcl++ chapter of the documentation for more information about them.

CAccessListCheck

Name

`CAccessListCheck` — Authenticate against a list of allowed credentials.

Synopsis

```
#include <AccessListCheck.h>


 class CAccessListCheck {
  CAccessListCheck();
   CAccessListCheck(const CAccessListCheck::StringMap& rSourceMap);
  CAccessListCheck(const CAccessListCheck& aCAccessListCheck);
  ~CAccessListCheck();
  CAccessListCheck& operator=(const CAccessListCheck& aCAccessListCheck);
  const StringMap getAccessList();
   virtual Bool_t Authenticate(CInteractor& rInteractor);
  virtual void AddAclEntry(const std::string& rEntry);
  virtual void DeleteAclEntry(const std::string& rEntry);
}
```

Description

The `CAccessListCheck` class implements authentication against one of a set of legal credentials. This is simlar in nature to the CTclAccessListCheck class, however the list is stored internally in the class rather than in a Tcl variable.

Public member functions

`CAccessListCheck`(void);

Constructs an access listchecking authenticator with an empty access control list. The functions `AddAclEntry` and `DeleteAclEntry` can be used to maintaint the contents of the access list.

`CAccessListCheck`const CAccessListCheck::StringMap&*`rSourceMap`*

Creates an access list checking authenticator. The initial contents of the access control list are *`rSourceMap`*. See "Types and public data" below for information about the StringMap data tpe.

`CAccessListCheck`const CAccessListCheck&*`aCAccessListCheck`*

Constructs an access list chekcing authenticator. The access list is a copy of the one from *`aCAccessListCheck`*

```
~CAccessListCheck();
```

Releases any storage or resources that were allocated by the object.

```
CAccessListCheck& operator=(const CAccessListCheck& aCAccessListCheck);
```

Assigns the current access list authenticator from *`aCAccessListCheck`*

```
const CAccessListCheck::StringMap getAccessList();
```

Returns a copy of the current access list. See "Types and public data" below for a description of the StringMap data type.

```
   virtual Bool_t Authenticate(CInteractor& rInteractor);
```

Obtains the credentials string from the *`rInteractor`* and determines if there is match for it in the current access control list. If there is, `kfTRUE` is returned. If not, `kfFALSE` is returned.

```
  virtual void AddAclEntry(const std::string& rEntry);
```

Adds an access control entry to he list. This is virtual to allow derived classes to override how this is done. the *`rEntry`* string is added to the access control list.

```
  virtual void DeleteAclEntry(const std::string& rEntry);
```

Removes the access control entry *`rEntry`* from the access control list.

Types and public data

CAccessListCheck::StringMap contains the access control list entries. This is a type that is defined in the `AccessListCheck.h` header as:

```
class CAccessListCheck  : public CAuthenticator
{
public:
  typedef std::set<std::string> StringMap;
...
};
```

CHostListCheck

Name

`CHostListCheck` — Authenticate from a list of TCP/IP hosts

Synopsis

```
#include <HostListCheck.h>
```

```
 class CHostListCheck {
  CHostListCheck();
   CHostListCheck (const CHostListCheck& aCHostListCheck);
  ~CHostListCheck();
   CHostListCheck& operator=(const CHostListCheck& aCHostListCheck);
  virtual Bool_t Authenticate(CInteractor& rInteractor);
   Bool_t Authenticate(const std::string& rHostname);
   virtual void AddAclEntry(const std::string& rHostname);
  virtual void DeleteAclEntry(const std::string& rHostname);
  Bool_t Authenticate(in_addr Address);
  void AddIpAddress(in_addr Address);
  void DeleteIpAddress(in_addr address);
}
```

Description

`CHostListCheck` is derived from the `CAccessListCheck`. The difference between the two classes is that `CHostListCheck` maintains an access control list made up of internet IP addresses. The credentials fetched from the interactor must be a host name that translates to an IP address, or a dotted IP address string that matches an address in the access control list.

Note that while the class allows users to invoke the original set of ACL Maintenance functions, calling these produces a warning on stderr. The preferred way to maintain the access control list is via the `AddIpAddress` and `DeleteIpAddress` member functions.

Public member functions

`CHostListCheck`(void);

Creates a `CHostListCheck` object with an empty access control list.

`CHostListCheck` const CHostListCheck&*`aCHostListCheck`*

Constructs a `CHostListCheck` object that is an exact copy of *`aCHostListCheck`*.

```
~CHostListCheck();
```

Releases all storage and any other resources that have been allocated by the object.

```
CHostListCheck& operator=(const CHostListCheck& aCHostListCheck);
```

Assigns *`aCHostListCheck`* to the object. This currently implies a deep copy of all member data.

```
virtual Bool_t Authenticate(CInteractor& rInteractor);
```

Fetches a host name or dotted IP address string from *`rInteractor`*. If the host fetched translates to an IP address that is in the access control list, the method returns `kfTRUE` otherwise, `kfFALSE` is returned. Host names, or addresses that cannot map to IP addresses will naturally return `kfFALSE`.

```
Bool_t Authenticate(const std::string& rHostname);
```

Same as above, however authentication is done with a host string *`rHostname`* that is already available to the application.

```
virtual void AddAclEntry(const std::string& rHostname);
```

Warns via stderr, that this is not the preferred way to add an IP address to the host list, and then adds the string *`rHostname`* without interpretation to the access control list. To fit in with the rest of the class, *`rHostname`* should be a string of the form `0xaaaaaaaa` where `aaaaaaaa` is the IP address in network byte order, with sufficient leading zeroes to ensure that there are eight digits.

```
virtual void DeleteAclEntry(const std::string& rHostname);
```

Warns the user that this is not the preferred way to manage the acdess control list and then deletes *`rHostname`* from the list if it exists. See `AddAclEntry` for a discussion of the correct form of Acl entries.

```
Bool_t Authenticate(in_addr Address);
```

Authenticates given the IP address *`Address`* in network byte order.

```
void AddIpAddress(in_addr Address);
```

Adds an IP address to the access control list.

```
void DeleteIpAddress(in_addr address);
```

Removes an IP address from the access control list.

Types and public data

In all cases in_addr is the IP address in *network byte order*.

CInteractor

Name

`CInteractor` — Base class for security interactions.

Synopsis

```
#include <Interactor.h>



 class CInteractor {
  CInteractor();
  CInteractor(const CInteractor& aCInteractor);
  virtual ~CInteractor();
  CInteractor& operator=(const CInteractor& aCInteractor);
  virtual  = 0 int Read(UInt_t nBytes, void* pData);
  virtual  = 0  int Write(UInt_t nbytes, void* pData);
  virtual int ReadWithPrompt(UInt_t nPromptSize, void* pPrompt, UInt_t nReadSize, void* pReadData);
  virtual void Flush();
}
```

Description

`CInteractor` is an abstract base class for the objects that authentication objects use to acquire credentials from clients. The class provides a set of interfaces that all concrete interactor classes must implement.

In one sense, the `CInteractor` hierarchy models obtaining credentials from a client as operations on a bi-directional byte stream. `Read`, and `Write` methods allow the application to read and write data from the credential.

Public member functions

`CInteractor`(void);

Place holder for constructors.

`CInteractor`constCInteractor&*`aCInteractor`*

Place holder for copy constructors.

```
virtual ~CInteractor();
```

Establishes destructor chaining through the class hierarchy.

```
CInteractor& operator=(const CInteractor& aCInteractor);
```

Declares the intent of the class hierarchy to offer an assignment operator.

```
virtual  = 0 int Read(UInt_t nBytes, void* pData);
```

Interface declaration for reading data from the interactor. *`nBytes`* is the number of bytes to attempt to read. *`pData`* Is a pointer to an output buffer into which the data will be read. The method is supposed to return the number of bytes that were actually read into *`pData`*. This may be no more than *`nBytes`*. In general, negative values indicate an error, and zero indicates that there is no data to read.

```
virtual  = 0  int Write(UInt_t nbytes, void* pData);
```

Provides the interface for a method to write data to the interactor. *`nbytes`* is the number of bytes to write, while *`pData`* points to a buffer from which the data are written. The return value is the number of bytes actually written. This will be negative for errors, zero if no data could be written, and some number that is at most *`nbytes`* if data transfer took place.

```
virtual int ReadWithPrompt(UInt_t nPromptSize, void* pPrompt, UInt_t nReadSize, void* pRea
```

A convenienece method for first writing a prompt string and then reading a response from the interactor. *`nPromptSize`* and *`pPrompt`* describe the prompt string and *`nReadSize`* and *`pReadData`* the input buffer for the response. The method is supposed to return the number of bytes read from the interactor. This will be have values like those of the `Read` method.

The default implementation is a write followed by a read.

```
virtual void Flush();
```

This member function supports flushing any output buffered in the interactor. The default implementation does nothing which is suitable for unbuffered interactors.

CStringInteractor

Name

`CStringInteractor` — Provide an interactor that processes strings.

Synopsis

```
#include <CStringInteractor.h>


 class CStringInteractor {
  CStringInteractor(const std::string& am_sString);
  CStringInteractor(const CStringInteractor& aCStringInteractor);
  ~CStringInteractor();
  const int operator==(const CStringInteractor& aCStringInteractor);
  const std::string getString();
  const int getReadCursor();
  void Rewind();
  virtual int Read(UInt_t nBytes, void* pBuffer);
  virtual int Write(UInt_t nBytes, void* pBuffer);
}
```

Description

String interactors allow you to treat strings gotten by whatever means as sources for interactors. Note that writes to a string interactor are not errors, they just don't do anything. This makes string interactors behave consistently if used as interactive entitites.

While string interactors behave exactly like any other interactor, they have additional member functions that recognize their string nature.

Public member functions

`CStringInteractor`conststd::string&*`am_sString`*

Constructs a string interactor given a string *`am_sString`*. The string will be the data 'read' by the interactor.

`CStringInteractor`constCStringInteractor&*`aCStringInteractor`*

Constructs a string interactor that is an exact state duplicate of *`aCStringInteractor`*. The duplication extends not only to the string but to the position within the string from which data will be returned to satisfy read operations (the cursor).

```
~CStringInteractor();
```

Releases all storage and resources required by the interactor prior to its finaly destruction.

```
const int operator==(const CStringInteractor& aCStringInteractor);
```

Assignment operator. The object will become an exact duplicate of the *`aCStringInteractor`*. This duplication extends to the cursor.

```
const std::string getString();
```

Informational method that returns the full string managed by the interactor.

```
const int getReadCursor();
```

Informational method that returns the read cursor. The read is the offset into the string managed by the interactor from which the next read will be satisfied. If, for example, an interactor `*interactor` was constructed on the string `astring`, the first character of the next read will be `astring[interactor->getReadCursor()]`

```
void Rewind();
```

Resets the read cursor to zero, allowing the string to be re-read.

```
virtual int Read(UInt_t nBytes, void* pBuffer);
```

Returns data from the string. At most *`nBytes`* of data are read beginning at the read cursor. If there are fewer, than *`nBytes`* of data left in the string, the entire remainder of the string is read. The data is copied from the string to *`pBuffer`*. The number of bytes actually read is returned as the method's function value.

After the read has been completed, the read cursor is advanced by the number of bytes returned. If there are no more bytes available in the string, the function will return `0` and no data will be transfered to *`pBuffer`*

```
virtual int Write(UInt_t nBytes, void* pBuffer);
```

This function simply returns the value of *`nBytes`*. This simulates successful completion of the write, although no data will actually be transferred.

EXAMPLES

This example shows how to determine that an interactor is a string interactor and if so, rewind it:

Example 1. Calling `CStringInteractor` specific members

```
CInteractor*       pAnInteractor = getInteractor();
...
CStringIteractor*  pString       =
         dynamic_cast<CStringInteractor*>(pAnInteractor);
if (pString) {
   pString->Rewind();
}
```

CFdInteractor

Name

`CFdInteractor` — Interact with file descriptor

Synopsis

```
#include <FdInteractor.h>
```

```
 class CFdInteractor {
  CFdInteractor(int fd);
  CFdInteractor(const CFdInteractor& rhs);
  CFdInteractor& operator=(const CFdInteractor& rhs);
  const int getFd();
  int Read(UInt_t nBytes, void* pBuffer);
  int Write(UInt_t nBytes, void* pData);
  void Flush();
}
```

DESCRIPTION

The `CFdInteractor` is an interactor that accepts and, if requested, provides data to anything that can be represented by a file descriptor. Reads and writes go directly to the file descriptor, the only buffering that is done is what is done by whatever handles the file descriptor at the operating system level.

PUBLIC MEMBER FUNCTIONS

`CFdInteractor`int*`fd`*

Construct an object of this class connected to the file descriptor *`fd`*.

`CFdInteractor`constCFdInteractor&*`rhs`*

Constructs a copy of the file descriptor *`rhs`*. The underlying file descriptor is duped so that the original one can be closed without affecting the copy.

```
CFdInteractor& operator=(const CFdInteractor& rhs);
```

Assigns to this from the *`rhs`*. The existing file descriptor is closed, and the file descriptor associted with *`rhs`* is duped so that actions by this object will be independent of those of the *`rhs`*

```
const int getFd();
```

This informational member returns the file descriptor associated with the object.

```
int Read(UInt_t nBytes, void* pBuffer);
```

Attempts to read *`nBytes`* of data from the file into *`pBuffer`*. The actual number of bytes read is returned as the value of the method. If `0` is returned, the fd has hit an eof or is in non-blocking mode with no data ready to be read. If negative, an error condition exists and the reason for the error will be in the global variable/macro `errno`

```
int Write(UInt_t nBytes, void* pData);
```

Attempts to write *`nBytes`* of data from *`pData`* to the file. The actual number of bytes of data written are returned to the caller. If an error has occured, the result will be negative. If the result is zero, likely the file descriptor is open in non=blocking mode, but cannot now be written to (e.g. it's a pipe without a reader).

```
void Flush();
```

Flushes any buffered output to the file. In practice this does nothing for file descriptors.

CIOInteractor

Name

`CIOInteractor` — Separate prompt and input interactors.

Synopsis

```
#include <IOInteractor.h>


 class CIOInteractor {
  CIOInteractor(CInteractor& rInput, CInteractor& rOutput);
  virtual ~CIOInteractor();
  const CInteractor* getOutput();
  const CInteractor* getInput();
  virtual int Read(UInt_t nBytes, void* pBuffer);
  virtual int Write(UInt_t nBytes, void* pBuffer);
  virtual void Flush();
}
```

Description

This class models an interactor that is made up of a read-only and a write-only interactor. Prompt/written data goes to the write-only interactor while reads go to the read-only interactor. A sample usage would be for an interactive application where the output interactor would be a `CFdInteractor` connected to `stdout`, and the input interactor `CFdInteractor` connected to `stdin`

Note that this class has no copy constructor as it is not possible to ensure that all iteractor classes now and in the future will have copy constructors. In most cases, this is not a restriction.

Public member functions

`CIOInteractor`CInteractor&*`rInput`*CInteractor&*`rOutput`*

Constructs the interactor, *`rInput`* is the interactor that will be used to get the input credentials. *`rOutput`* the interator to which prompts will be sent.

```
const CInteractor* getOutput();
```

Informational function that returns a pointer to the output interactor used to construct this object.

```
const CInteractor* getInput();
```

Informational function that returns a pointer to the input interactor used to construct this object.

```
virtual int Read(UInt_t nBytes, void* pBuffer);
```

Attempts to read *`nBytes`* from the input interactor into the buffer pointed to by *`pBuffer`*. The return value is determined by the type of the actual interactor, but generally is the actual number of bytes read. Usually a negative value indicates an error condition of some sort, and a zero indicates there is no data to be read, either because the end of the data source has been reached or because the input source is a non-blocking entity.

```
virtual int Write(UInt_t nBytes, void* pBuffer);
```

Writes *`nBytes`* of data from *`pBuffer`* to the output interactor. The return value depends on the actual interactor. Usually negative values indicate an error, zero usually indicates output to a non-blocking entity that is not able to accept output at that time, and values less than *`nBytes`* represent devices with some blocking/buffering factor that has been exceeded by the write.

```
virtual void Flush();
```

Flushes data in output buffes the output interactor may have.

CTCLApplication 3

Name

`CTCLApplication` — Base class for TCL/Tk applications.

Synopsis

```
#include <tcl.h>
#include <TCLApplication.h>
...
```

```
class CTCLApplication  : public CTCLInterpreterObject
{
public:
  CTCLApplication ();
  ~CTCLApplication ( );
  virtual   int operator() ()  =0;
};
```

DESCRIPTION

`CTCLApplication` is an abstract base class that facilitates the creation of applications that extend the Tcl interpreter. The ‘main program’ of SpecTcl is derived from this class, for example.

Initializing a Tcl application generallly consists of a bunch of boilerplate that initializes the interpreter, and then a bunch of application specific code to register extensions to the interpreter. `CTCLApplication` provides the main boilerplate. It is expected that you derive a class from `CTCLApplication` Implement `operator()` to register application specific commands, and then create exactly one instance of your application class named, and a global pointer to that object named gpTCLApplication.

For example, suppose you have created a class named MyTclApp:

```
// This code is at the global level:
...
MyTclApp app;                              // Makes an instance of this
CTCLApplication* gpTCLApplication = &app;  // Pointer expected by framework.
...
```

Will ensure that the `operator()` of your application object will be called with the interpreter already initialized.

METHODS

int `operator()`()

This function is pure virtual and must be overridden by your derived class. `operator()` is expected to install all required extensions to the interprter and return to it to start the main event loop. The return value from this should be `TCL_OK` if the application was successfully initialized or `TCL_ERROR` if the program encountered an error that should prevent the interpreter main loop from starting

SEE ALSO

CTCLInterpreter, CTCLInterpreterObject, CTCLObjectProcessor, CTCLVariable

CTCLException

Name

`CTCLException` — Class for reporting exceptional conditions in Tcl applications via the C++ try/catch mechanism.

Synopsis

```
#include <TCLException.h>
...
class CTCLException  : public CTCLInterpreterObject ,public CException
{
public:
  CTCLException (CTCLInterpreter& am_rInterpreter,
                 Int_t am_nReason,
                 const char* pString);
  CTCLException(CTCLInterpreter& am_rInterpreter,
                Int_t am_nReason,
                const std::string& rString);
  CTCLException (const CTCLException& aCTCLException );
  virtual ~CTCLException ( );

  CTCLException operator= (const CTCLException& aCTCLException);
  int operator== (const CTCLException& aCTCLException);

  void AddErrorInfo (const char* pMessage)  ;
  void AddErrorInfo(const std::string& rMessage);
  void AddErrorInfo(const CTCLString& rMessage);

  void SetErrorCode (const char* pMessage,
                     const char* pMnemonic="???",
                     const char* pFacility="TCL",
                     const char* pSeverity="FATAL")  ;
  void SetErrorCode(const std::string rMessage,
                    const std::string &rMnemonic=std::string("???"),
                    const std::string &rFacility=std::string("TCL"),
                    const std::string &rSeverity=std::string("FATAL"));

  virtual   const char* ReasonText () const;
  virtual   Int_t ReasonCode () const  ;
};
```

DESCRIPTION

The `CTCLException` class allows you to instantiate and throw exceptions that are distinguishable as coming from the TCL library and its extensions. In most cases the TclPlus library itself will convert error conditions detected by the Tcl API and intantiate and throw an appropriate exception.

The following example shows how to execute code that is aware of these exceptions. In this case, the code just reports the error message and continues.

```
try {
    // In here is TclPlus invoking code.
}
catch (CTCLException& e) {
    cerr << "TclPlus error caught: " << e.ReasonText() << endl;
}
```

The following example shows a typical code segment that throws a `CTCLException`:

```
int status = Tcl_xxxxxxx(pInterp->getInterpreter()....); // Some Tcl call.
if (status != TCL_OK) {
    throw CTCLException(*pInterp, status,
                        "Call to Tcl_xxxxxx returned an error");
}
```

Note that constructing a `CTCLException` object incorporates the Tcl result string at the time into the text returned by the `ReasonText()` member function.

METHODS

```
CTCLException (CTCLInterpreter& rInterpreter,
               Int_t nReason,
               const char* pString);
CTCLException(CTCLInterpreter& rInterpreter,
              Int_t nReason,
              const std::string& rString);
```

```
CTCLException (const CTCLException& aCTCLException );
```

These construct a `CTCLException`. *`rInterpreter`* is a reference to the intepreter that was used in the operation that resulted in the error. The result string of that interpreter will be saved as part of the text returned by the `ReasonText` member function.

The *`nReason`* is a reason for the exception. Typically this will be `TCL_ERROR` however other error codes can be created and used for application specific problems. This is the value that will be returned by the `ReasonCode` member function.

`rString` and *`pString`* are intended to provide information about the context of the error, and will be incorporated into the text strin greturned from `ReasonText`.

`aCTCLException` is a reference for the sourc object of the copy constructor.

```
CTCLException operator= (const CTCLException& rhs);
int operator==(const CTCLException& rhs);
```

These two functions provide a mechanism to assign exceptions and to compare them for equality. *`rhs`* is the object that is the source of the assignment or the object to which `this` is being compared. Equality is defined as the two exceptions having the same underlying interpreter, and same reason text.

```
void AddErrorInfo (const char* pMessage)  ;
void AddErrorInfo(const std::string& rMessage);
void AddErrorInfo(const CTCLString& rMessage);
```

These functions are wrapperf ro the API function `Tcl_AddErrorInfo` the *`pMessage`*, and *`rMessage`* parameters provide the message that is added to the *`errorInfo`* variable.

```
void SetErrorCode (const char* pMessage,
                   const char* pMnemonic="???",
                   const char* pFacility="TCL",
                   const char* pSeverity="FATAL")  ;
void SetErrorCode(const std::string rMessage,
                    const std::string& rMnemonic=std::string("???"),
                    const std::string& rFacility=std::string("TCL"),
                    const std::string& rSeverity=std::string("FATAL"));
```

These function set the `errorCode` Tcl interpreter variable. The convention these function support is to set the error code to a list that consists of a message (*`pMessage`* and *`rMessage`*, mnemonic for the message (*`pMnemonic`* or *`rMnemonic`*, the Facility (*`pFacility`* or *`rFacility`*)that is throwing the error and the severity (*`pSeverity`* or *`rSeverity`*) of the error.

```
virtual   const char* ReasonText () const;
virtual   Int_t ReasonCode () const  ;
```

These two functions are intended for use by exception catch blocks. `ReasonText` provides human readable text that describes the exception. `ReasonCode` provides a numerical code that describes the exception. Often this just has the value `TCL_ERROR`

SEE ALSO

Tcl_AddErrorInfo(3tcl), Tcl_SetErrorCode(3tcl)

CTCLInterpreter

Name

`CTCLInterpreter` — Encapsulate a Tcl interpreter.

Synopsis

```
#include <string>
#include <vector>
#include <TCLInterpreter.h>

class CTCLInterpreter
{
public:
```

```
CTCLInterpreter ();
CTCLInterpreter (Tcl_Interp* am_pInterpreter  );

Tcl_Interp* getInterpreter()
std::string Eval (const char* pScript) ;
std::string Eval(const CTCLString& rScript);
std::string Eval(const std::string& rScript);
std::string EvalFile (const char* pFilename)   ;
std::string EvalFile(const CTCLString& rFilename);
std::string EvalFile(const std::string& rFilename);

std::string GlobalEval (const char* pScript)   ;
std::string GlobalEval (const CTCLString& rScript) ;
std::string GlobalEval(const std::string& rScript) ;

std::string RecordAndEval (const char* pScript, Bool_t fEval=kfFALSE);
std::string RecordAndEval(const CTCLString& rScript,
                          Bool_t fEval=kfFALSE);
std::string RecordAndEval(const std::string& rScript,
                          Bool_t fEval = kfFALSE);

std::string ExprString (const char* pExpression)   ;
std::string ExprString(const CTCLString& rExpr);
std::string ExprString(const std::string& rExpr);

Long_t ExprLong (const char* pExpression)   ;
Long_t ExprLong (std::string& rExpression);
Long_t ExprLong (const CTCLString& rExpr);

DFloat_t ExprDouble (const char* pExpression)   ;
DFloat_t ExprDouble (const CTCLString& rExpression);
DFloat_t ExprDouble(const std::string& rExpression);

Bool_t ExprBoolean (const char*  pExpression)   ;
Bool_t ExprBoolean (const CTCLString& rExpression);
Bool_t ExprBoolean(const std::string& rExpression);

std::string TildeSubst (const char* pFilename) const  ;
std::string TildeSubst (const CTCLString& rName) const;
std::string TildeSubst (const std::string& rName) const;
std::string EvalFile (const char* pFilename)   ;
std::string EvalFile(const CTCLString& rFilename);
std::string EvalFile(const std::string& rFilename);

std::string GlobalEval (const char* pScript)   ;
std::string GlobalEval (const CTCLString& rScript) ;
std::string GlobalEval(const std::string& rScript) ;

std::string RecordAndEval (const char* pScript, Bool_t fEval=kfFALSE);
std::string RecordAndEval(const CTCLString& rScript,
                          Bool_t fEval=kfFALSE);
std::string RecordAndEval(const std::string& rScript,
                          Bool_t fEval = kfFALSE);
```

```
  std::string ExprString (const char* pExpression)   ;
  std::string ExprString(const CTCLString& rExpr);
  std::string ExprString(const std::string& rExpr);

  Long_t ExprLong (const char* pExpression)   ;
  Long_t ExprLong (std::string& rExpression);
  Long_t ExprLong (const CTCLString& rExpr);

  DFloat_t ExprDouble (const char* pExpression)   ;
  DFloat_t ExprDouble (const CTCLString& rExpression);
  DFloat_t ExprDouble(const std::string& rExpression);

  Bool_t ExprBoolean (const char*  pExpression)   ;
  Bool_t ExprBoolean (const CTCLString& rExpression);
  Bool_t ExprBoolean(const std::string& rExpression);

  std::string TildeSubst (const char* pFilename) const  ;
  std::string TildeSubst (const CTCLString& rName) const;
  std::string TildeSubst (const std::string& rName) const;
  Tcl_Interp* operator-> ();
  operator Tcl_Interp* ();
};
```

DESCRIPTION

`CTCLInterpreter` encapsulates a Tcl_Interp* in an object. Method invocations on that object provide access to many of the Tcl interpreter. See METHODS below for more information about htis.

METHODS

`CTCLInterpreter ()`

`CTCLInterpreter` (Tcl_Interp* *`pInterp`*)

Constructs an interpreter object. The first form of this constructor creates a new Tcl_Interp* using `Tcl_CreateInterp()` and wraps the object around it. All members of the object will operate on that newly created interpreter. The second form, wraps an object around *`pInterp`*, a previously created Tcl_Interp*. Note that in either case on destruction, `Tcl_DeleteInterp()` is called on the wrapped interpreter.

Tcl_Interp* `getInterpreter`()

Returns the interpreter that is being wrapped by this object. This interpreter can be used as an *`interp`* parameter for any `Tcl_xxxxxx` call in the Tcl API.

```
std::string Eval(const char* pScript) ;
std::string Eval(const CTCLString& rScript);
std::string Eval(const std::string& rScript);
```

Evaluates the script passed as a parameter. The only differences between these functions is the form of the script parameter. Each function will return the result of the script. If there is an error in the script, a `CTCLException` will be thrown that will describe what happened. For example:

```
std::string commands;
CTCLInterpreter interp;     // New intepreter.
...
// after commands has been built up:

string result
try {
    result = interp.Eval(commands);
    cout << "Eval of " << commands << " was "
         <<  result << endl;
}
catch (CTCLException &e) {
    cerr << "Eval of " << commands << " failed: "
         << e.ReasonText() << endl;
}
// If no exception, result is usable as the output of the eval.
```

```
std::string EvalFile(const char* pFilename)   ;
std::string EvalFile(const CTCLString& rFilename);
std::string EvalFile(const std::string& rFilename);
```

Sources the specified file in and executes it as a script in the interpreter that is wrapped by the object. The only difference between these functions is how the name of the file is passed. The return value is the script result. A `CTCLException` will be thrown in the event the script reports an error. See the example in `CTCLInterpreter::Eval` to see how to catch and report this kind of exception.

```
std::string GlobalEval(const char* pScript);
std::string GlobalEval(const CTCLString& rScript);
std::string GlobalEval(const std::string& rScript);
```

This function evaluates a script at the global level. Note that `CTCLInterpreter::Eval`, and `CTCLInterpreter>::EvalFile` evaluates the script at whatever call level the interpreter is currently executing at. The only difference between the methods above is how the script is passed. The functions all return the interpreter result after the script executes. If the script reports an error, a `CTCLException` will be thrown.

```
std::string RecordAndEval (const char* pScript,
                          Bool_t fEval=kfFALSE);
std::string RecordAndEval(const CTCLString& rScript,
                          Bool_t fEval=kfFALSE);
std::string RecordAndEval(const std::string& rScript,
                        Bool_t fEval=kfFALSE);
```

Records a script in the Tcl interpreter history and, if *`fEval`* is `kfTRUE`, evaluates it as well. The return value is the interpreter result, which is only meaningful if the script was evalutated. If the script reports an error, a `CTCLException` is thrown.

```
std::string ExprString(const char* pExpression);
std::string ExprString(const CTCLString& rExpr);
std::string ExprString(const std::string& rExpr);
```

Evaluates an expression (as if with the **expr** Tcl command), and returns the result of the evaluation as a string. If the expression has an error, a `CTCLException` will be thrown. The only difference between these functions is how the expression is passed.

```
Long_t ExprLong(const char* pExpression)   ;
Long_t ExprLong(std::string& rExpression);
Long_t ExprLong(const CTCLString& rExpr);
```

Evaluates an expression (as if with the **expr** Tcl command). If the result can be converted into an integer, it is returned as a Long_t. If the expression either cannot be converted to an integer (e.g. it's a non-numerical expression), or if the expression contains an error, a `CTCLException` will be thrown.

```
DFloat_t ExprDouble(const char* pExpression)   ;
DFloat_t ExprDouble(const CTCLString& rExpression);
DFloat_t ExprDouble(const std::string& rExpression);
```

Evaluates the parameter as an expression (as if with the **expr** Tcl command). If the result can be converted to a floating point value it is returned as the function value. If not, or if there is an error in the expression, a `CTCLException` is thrown.

```
Bool_t ExprBoolean(const char*  pExpression)   ;
Bool_t ExprBoolean(const CTCLString& rExpression);
Bool_t ExprBoolean(const std::string& rExpression);
```

Evaluates the parameter as an expression (as if with the **expr** Tcl command). If the result can be interpreted as a boolean, it is returned as the function value. If not, or if there is an error in the expression, a `CTCLException` is thrown.

```
std::string TildeSubst(const char* pFilename) const  ;
std::string TildeSubst(const CTCLString& rName) const;
std::string TildeSubst(const std::string& rName) const;
```

Performs tilde substitution on its parameter. Tilde substitution means that leading characters are expanded to the current user's home directory path, while a leading followed by a word that is a username will be expanded to the home directory path of that user. The expanded value is returned. Note thatthe use of this member is deprecated as the underlying Tcl library function is also deprecated.

```
  Tcl_Interp* operator->();
  operator Tcl_Interp* ();
```

These two functions allow objects that are `CTCLInterpreter` objects to be treated as if they were Tcl_Interp*'s. `operator->` supports dereferncing to fields of the wrapped interpreter (note that this is now deprecated within Tcl itself). `operator Tcl_Interp*` supports dynamic type conversion from a `CTCLInterpreteter` object and a Tcl_Interp* pointer.

DEFECTS

It is not possible to avoid destroying the interpreter when the object is destroyed.

SEE ALSO

CTCLException, CTCLApplication, CTCLChannel, CTCLCommandPackage, CTCLFileHandler, CTCLList, CTCLObject, CTCLObjectProcessor, CTCLTimer, CTCLVariable

CTCLInterpreterObject 3

Name

`CTCLInterpreterObject` — Base class for objects that are associated with a Tcl Interpreter.

Synopsis

```
#include <CTCLInterpreterObject.h>
...
class CTCLInterpreterObject
{
public:
  CTCLInterpreterObject ();
  CTCLInterpreterObject (CTCLInterpreter* pInterp );
  CTCLInterpreterObject (const CTCLInterpreterObject& src );

  CTCLInterpreterObject& operator=
                        (const CTCLInterpreterObject& rhs);
  int operator== (const CTCLInterpreterObject& rhs) const;
  CTCLInterpreter* getInterpreter() const;
  CTCLInterpreter* Bind (CTCLInterpreter& rBinding);
  CTCLInterpreter* Bind (CTCLInterpreter* pBinding);

};
```

DESCRIPTION

`CTCLInterpreterObject` is a base class for any object that requires a CTCLInterpreter (Tcl interpreter) to operate. Almost all objects in the Tcl++ library are derived from this base class.

METHODS

`CTCLInterpreterObject()`

Constructor for an interpreter object that will be bound to an underlying interpreter at a later time. See the `Bind` functions for more information about binding interpreters. More normally, if you already have an interpreter you will construct using that interpreter.

`CTCLInterpreterObject` (CTCLInterpreter* *`pInterp`*)

Constructs a `CTCLInterpreterObject` given that *`pInterp`* is an existing interpreter encapsulated in a `CTCLInterpreter`.

`CTCLInterpreterObject` (const CTCLObject& *`src`*)

Constructs a new `CTCLInterpreterObject` that is an exact copy of *`src`*.

`CTCLInterpreterObject&` `operator=` (const CTCLInterpreterObjectd& *`rhs`*)

Provides a mechanism for assigning a `CTCLInterpreterObject` a copy of the *`rhs`* `CTCLInterpreterObject`. The return value is just a reference to the left hand side of the assignment. This permits operator chaining.

int `operator==`(const CTCLInterpreterObject& rhs)

Compares a `CTCLInterpreterObject` to another (*`rhs`*). If the underlying interpreters are the same, the objects are said to be equal and `1` is returned. If not, `0` is returned.

CTCLInterpreter* `getInterpreter`() const

Returns a pointer to the underlying `CTCLInterpreter` object. See the `CTCLInterpreter`(3) manpage for more information about the services offered by that class.

CTCLInterpreter* `Bind`(CTCLInterpreter& *`rBinding`*)

CTCLInterpeter* `Bind`(CTCLInterpreter* *`pBinding`*)

Binds the object to a new interpreter. Typically this will only be called when the object was constructed without an initial interpreter. This is because most objects really are related to some interpreter and cannot be willy-nilly rebound. The return value is a pointer to the `CTCLInterpreter` the object was previously bound to. This will be `NULL` if the object was not initially bound.

DEFECTS

No `operator!=` was defined.

SEE ALSO

CTCLInterpreter, CTCLApplication, CTCLChannel, CTCLCommandPackage, CTCLFileHandler, CTCLList, CTCLObject, CTCLObjectProcessor, CTCLTimer, CTCLVariable

CTCLList

Name

`CTCLList` — Provide access to Tcl List parsing.

Synopsis

```
#include <TCLList.h>
...
class CTCLList  : public CTCLInterpreterObject
```

```
{

public:
  CTCLList (CTCLInterpreter* pInterp);
  CTCLList (CTCLInterpreter* pInterp, const  char* am_pList  );
  CTCLList (CTCLInterpreter* pInterp, const std::string& rList);
  CTCLList (const CTCLList& aCTCLList );

  CTCLList& operator= (const CTCLList& aCTCLList);
  int operator== (const CTCLList& aCTCLList);
  int operator!= (const CTCLList& aCTCLList);

  const char* getList() const;

  int Split (StringArray& rElements)  ;
  int Split (int& argc, char*** argv);

  const char* Merge (const StringArray& rElements)  ;
  const char* Merge(int argc, char** argv);

};
```

DESCRIPTION

Tcl Lists are white space separated words. It is definition of words and quoting issues that makes the parsing of lists less than straightforward.

Fortunately, Tcl provides several list processing functions. The concept of a list and access to list processing functions are encapsulated in the `CTCLList` class.

METHODS

```
CTCLList (CTCLInterpreter* pInterp);
CTCLList (CTCLInterpreter* pInterp,
          const  char* pList  );
CTCLList (CTCLInterpreter* pInterp,
          const std::string& rList);
CTCLList (const CTCLList& rhs);
```

These four functions provide various ways to create a `CTCLList` object. The first constructor creates an empty list. The next two, create a list that has an initial value given by either the `NULL` terminated string *`pList`*, or the std::string object *`rList`*. The final constructor creates a list that is a duplicate of the list described by the object *`rhs`*.

```
CTCLList& operator= (const CTCLList& rhs);
int operator== (const CTCLList& rhs);
int operator!= (const CTCLList& rhs);
```

These function provide assignment (`operator=`), equality comparison (`operator==`), and inequality comparison (`operator!=`) with another `CTCLList` object, *`rhs`*. Assignment is defined as copying the string format of the list. Equality comparison is defined as the both interpreter and strings being equal. Inequality is defined as !operator==.

```
const char* getList() const;
```

`getList` returns an immutable pointer to the string rerpesentation of the list. Note that the const qualfier on the pointer means that attempts to dereference the pointer which would modify the list result in error messages. For Example:

```
        CTCLList aList(pInterp, "some list");
        const char* pData = aList.getList();
        *pData = ’S’;         // Compiler error!!!!
```

```
int Split (StringArray& rElements)  ;
int Split (int& argc,
           char*** argv);
```

Splits a list up into its component words. *`relements`* is a std::vector<std::string> into which the elements will be split. *`argc`* is a reference to an integer into which the number of elements will be put. *`argv`* is a pointer to a char** into which will be placed a pointer to dynamically allocated storage containing a list of *`argc`* pointers to the words in the string. This storage must be released by the caller with `Tcl_Free`. For example:

```
CTCLList someList(pInterp, someInitialContents);
...
int argc;
char** argv;
someList.Split(argc, &argv);
//
//  ... do something with the data
//
...
//
// Done with the list elements.
//
Tcl_Free((char*)argv);
```

```
const char* Merge (const StringArray& rElements)  ;
const char* Merge(int argc,
                 char** argv);
```

Merges a bunch of words in to a list. If necessary, quoting is performed to ensure that words that have whitespace or other special characters will be correctly formatted into the list. *`rElements`* is a std::vector<std::string> of words that will be merged into the string. *`argc`* is a count of the number of words, and *`argv`* is a pointer to an array of pointers to the words stored as `NULL` terminated strings. The return value is the final string representation of the list after the merge operation has been performed.

SEE ALSO

CTCLInterpreter(3), CTCLInterpreterObject(3), Tcl_Free(3tcl)

CTCLObject

Name

`CTCLObject` — Encapsulate Tcl Dual ported objects.

Synopsis

```
#include <TCLObject.h>
...
class CTCLObject : public CTCLInterpreterObject
{

public:
  CTCLObject ();
  CTCLObject (Tcl_Obj* am_pObject);
  CTCLObject (const CTCLObject& aCTCLObject );
  virtual  ~CTCLObject ( );

  CTCLObject& operator= (const CTCLObject& aCTCLObject);
  int operator== (const CTCLObject& aCTCLObject) const;

  Tcl_Obj* getObject();
  const Tcl_Obj* getObject() const;

   CTCLObject& operator= (const std::string& rSource)    ;
   CTCLObject& operator= (const char* pSource)    ;
   CTCLObject& operator= (int nSource)    ;
   CTCLObject& operator= (const CTCLList& rList)    ;
   CTCLObject& operator= (double dSource)     ;
   CTCLObject&  operator=(Tcl_Obj* rhs);

  operator std::string ()    ;
  operator int ()    ;
  operator CTCLList ()    ;
  operator double ()    ;

  CTCLObject& operator+= (const CTCLObject& rObject)    ;
  CTCLObject& operator+= (int nItem)    ;
  CTCLObject& operator+= (const std::string& rItem)    ;
  CTCLObject& operator+= (const char* pItem)    ;
  CTCLObject& operator+= (double  Item)    ;

  CTCLObject clone ()    ;

  CTCLObject operator() ()    ;

  CTCLObject   getRange(int first, int last);

  CTCLObject&  concat(CTCLObject& rhs); // Concat lists.
  std::vector<CTCLObject>  getListElements();
  CTCLObject&  setList(std::vector<CTCLObject> elements);
  int          llength();
  CTCLObject   lindex(int index);
  CTCLObject&  lreplace(int first, int count, std::vector<CTCLObject> newElements);

};
```

DESCRIPTION

Tcl as a scripting language carries a deeply embedded philosophy that everything can be treated as a string. Nonetheless, in many cases, entities manipulated by the interpreter are more efficiently manipulated when they have other types of internal representations. For example strings which represent floating point numbers in extended computations are more efficiently represented directly as float or double variables.

Tcl uses *dual ported* objects to capture this efficiently. A Tcl object is a thing that has a string representation and at most one other typed representation (e.g. list, integer, floating point). Conversions from string to this representation are done once and cached as long as possible, so that when an object has been used as a particular type there is essentially no additional conversion cost to use it as that type again.

While a Tcl_Obj and therefore a `CTCLObject` can exist independent of an interpreter, many member functions require the object be bound to an interpreter, or they will fail with an assertion failure causing the program to abort. Use the base class `Bind` (`CTCLInterpreterObject`::`Bind`) member to bind the object to an existing interpreter, usually as soon as possible.

Tcl objects also can be shared with a lazy copy on write scheme so that overhead associated with duplicating objects (e.g. when using them as parameters to Tcl commands) is minimized.

`CTCLObject` exposes an object oriented interface to the Tcl dual ported object.

METHODS

```
CTCLObject ();
CTCLObject (Tcl_Obj* pObject);
CTCLObject (const CTCLObject& rhs );
```

Constructs a Tcl object wrapped in a `CTCLObject`. *`pObject`* is an existing Tcl_Obj pointer that will be wrapped. *`rhs`* is an existing `CTCLObject` that will be used to create another reference to the same underlying object. Note that in the last two of these forms, a new Tcl_Obj is not created. Instead, `Tcl_IncrRefCount` is used on the previously existing object to mark it as shared. All member functions which modify the underlying object will create a new object (copy on write semantics), and decrement the reference count of the original object. Destroying a `CTCLObject` invokes `Tcl_DecrRefCount` on the underlying Tcl_Obj object. This may or may not result in destruction of that underlying object depending on the resulting reference count.

```
CTCLObject& operator= (const CTCLObject& rhs);
int operator== (const CTCLObject& rhs) const;
```

These two members provide assignment and equality comparison for `CTCLObject` instances with another object *`rhs`*. Assignment operates efficiently by decrementing the reference count on the prior object, incrementing the reference count for *`rhs`*, and copying its Tcl_Obj* only. Equality comparision is true if the underlying objects have the same string representation.

```
Tcl_Obj* getObject();
const Tcl_Obj* getObject() const;
```

Retrieves a mutable or immutable pointer to the underlying object. If you intend to retain this pointer for longer than the lifetime of the `CTCLObject` object from which it comes or longer than the lifetime of the execution of the calling function you should invoke `Tcl_IncrRefCount` to mark the object shared and prevent its destruction until you no longer need it, at which point you should invoke `Tcl_DecrRefCount`.

You should not modify the underlying object as that violates the copy on write semantics expected of Tcl_Obj objects. Instead, use `Tcl_DuplicateObj` to create a new object (decrementing the reference count of the previous object), and modify that one instead. The following code snippet shows this:

```
int      len;
Tcl_Obj* pObject = someObject.getObject();
string   value   = string(Tcl_GetStringFromObj(pObject, &len));
value           += "new text";
pObject          = Tcl_DuplicateObj(pObject); // Split off a new object.
Tcl_SetStringObj(pObject, (char*)value.c_str(), -1);
```

```
CTCLObject& operator= (const std::string& rSource)    ;
CTCLObject& operator= (const char* pSource)    ;
CTCLObject& operator= (int nSource)    ;
CTCLObject& operator= (const CTCLList& rList)    ;
CTCLObject& operator= (double dSource)     ;
CTCLObject& operator=(Tcl_Obj* rhs);
```

Assigns a new value to the object. The reference count of the previously encapsulated object is decremented and a new object is created into which the right hand side value is loaded. This preserves copy on write semantics. *`rSource`* and *`pSource`* load the new object with a string valued entity. No attempt is made to create another representation for the object (yet). *`nSource`* loads the object with an integer value and its string representation. *`rList`* loads the object with a list representation and its string representation. *`dSource`* loads the object with a double precision floating point value and its string representation. *`rhs`* simply copies in the new object pointer and increments its reference count.

```
operator std::string ()    ;
operator int ()    ;
operator CTCLList ()    ;
operator double ()    ;
```

These function provide implicit and explicit type conversions between a `CTCLObject` instance and other types. The type conversions attempt to extract the appropriately typed value from the underlying object. If successful, the value is returned. On failure, a `CTCLException` is thrown. For example:

```
CTCLObject object = "3.14159";  // String rep.
object.Bind(pInterp);           // Some of these need an interp.
double     pi     = object      // (operator double()).
object            = "george";   // string rep.
try {
   int trash = object;          // fails.
}
catch (CTCLException& e) {
    // this catch block will execute.
}
```

```
CTCLObject& operator+= (const CTCLObject& rObject)    ;
CTCLObject& operator+= (int nItem)    ;
CTCLObject& operator+= (const std::string& rItem)    ;
CTCLObject& operator+= (const char* pItem)    ;
CTCLObject& operator+= (double  Item)    ;
```

Creates the list representation of the underlying object, converts either *`rObject`*, *`nItem`*, *`rItem`*, *`pItem`*, *`Item`* to its string representation and appends it as a list entry to the object.

```
  CTCLObject clone ()    ;
```

A wrapper for `Tcl_DuplicateObj`. The object is duplicated and its duplicate is returned wrapped by a `CTCLOjbect`.

```
CTCLObject operator() ()    ;
```

The object's string representation is compiled by its bound interpreter to Tcl byte code and executed as a script by that bound interpreter. Note that the byte code compilation is cached so that subsequent invocations of the script will not require recompilation unless other references force a different second representation on the object (e.g. fetching it as a list). The result of the script execution is returned as a new `CTCLObject` If script compilation failed, or script execution resulted in an error, a `CTCLException` will be thrown describing this.

```
CTCLObject getRange(int first,
                    int last);
```

Returns a new object that consists of a subrange of the string representation of the original object. *`first`* is the index of the first character of the substring returned. *`last`* is the index of the last character of the substring. See `Tcl_GetRange` for more information, note however that some values of *`first`* or *`last`* will be treated specially, and that the underlying string representation operated on is a *Unicode* string for which some characters in some languages may require more than one byte.

```
CTCLObject&  concat(CTCLObject& rhs); // Concat lists.
```

Concatenates the *`rhs`* as a list element to the object. A refrence to the new object is returned. Copy on write semantics are maintained.

```
std::vector<CTCLObject>  getListElements();
```

Converts the object into its underlying list representation. The elements of the list are loaded into a vector of `CTCLObject` objects and returned. If the underlying string representation does not have a valid list representation, (e.g. "{this cannot be converted") a `CTCLException` is thrown.

```
CTCLObject&  setList(std::vector<CTCLObject> elements);
```

Loads the object with a string and list representation whose words are the appropriately quoted string representation of *`elements`*. A reference to the new object is returned. Copy on write semantics are maintained.

```
int llength();
```

If necessary, converts the object to its list representation and returns the number of elements in that list. If it is not possible to convert the string represenation of the object into a valid list, a `CTCLException` is thrown.

```
CTCLObject lindex(int index);
```

If necessary, creates the list representation of the object and returns a new object that is element number *`index`* of that list. If the object cannot be converted into a list, a `CTCLException` is thrown.

```
CTCLObject& lreplace(int first,
                     int count,
                     std::vector<CTCLObject> newElements);
```

If necessary, converts the object to its list representation. If that conversion fails a `CTCLException` is thrown. The set of elements specified by *`first`* and *`count`*, are replaced by the words held in the vector *`newElements`*. *`newElements`* can, of course, be an empty vector in order to remove *`count`* elements starting at *`first`* from the list. A reference to the resulting object is returned. Copy on write semantics are enforced.

SEE ALSO

CTCLException(3), CTCLInterpreter(3), CTCLInterpreterObject(3), Tcl_DecrRefCount(3tcl), Tcl_DuplicateObj(3tcl), Tcl_GetRange(3tcl), Tcl_IncrRefCount(3tcl), Tcl_NewObj(3tcl), Tcl_SetStringObj(3tcl)

CTCLObjectProcessor

Name

`CTCLObjectProcessor` — Abstract base class to encapsulate the Tcl object command interface exposed by `Tcl_CreateObjCommand`.

Synopsis

```
#include <TCLObjectProcessor.h>
...
class CTCLObjectProcessor : public CTCLInterpreterObject
{
public:
  CTCLObjectProcessor(CTCLInterpreter& interp,
                      std::string      name,
                      bool             registerMe=true);
  virtual ~CTCLObjectProcessor();

  void Register();               // Register command on the interpreter.
  void unregister();             // Unregister command from the interp.
  std::string getName() const;  // Return the name of the object.
  Tcl_CmdInfo getInfo() const;  // Return info about the command.

  virtual int operator()(CTCLInterpreter& interp,
                         std::vector<CTCLObject>& objv) = 0;
  virtual void onUnregister();

};
```

DESCRIPTION

Tcl supports the addition of commands to the interpreter. `CTCLObjectProcessor` supports an object oriented encapsulation of this part of the API. To add a command to an interpreter, write a subclass of `CTCLObjectProcessor`. This subclass should override `operator()`, and optionally `onUnregister`. to implement the desired behavior for the new command.

Create an instance of this new class and invoke its `Register` member to add it to the interpreter onto which it is bound. Whenever a script executes the new command that object's `operator()` is invoked to process the command. If the interpreter is destroyed, or if the command is ever unregistered, the `onUnregister` function is called to perform any required global cleanup.

METHODS

```
CTCLObjectProcessor(CTCLInterpreter& interp,
                    std::string      name,
                    bool             registerMe=true);
```

Constructs a new command processor. *`interp`* is the interpreter on which the command will be registered when the `Register` member is invoked. *`name`* is the name of the command. If *`registerMe`* is not supplied or is supplied but is `true`, the command will be registered as part of the construction process. If *`registerMe`* is supplied and is `false`, the command is not immediately added, and `Register` must be called later to incorporate it into the interpreter.

```
void Register();
void unregister();
```

`Register` incorporates the command into the interpreter. If the command is already registered, a `CStateException` is thrown.

`unRegister` removes the command from the interpreter. This causes `onUnregister` to be called. if the command is registered at destruction time, destruction implies a call to `unRegister` (and therefore `onUnregister`).

```
std::string getName() const;
Tcl_CmdInfo getInfo() const;
```

`getName` returns the name of the command that will invoke this object's `operator()`. If the command has been registered, and subsequently renamed at the script level, this function will reflect the rename.

`getInfo` returns information about the command see `Tcl_GetCommandInfo` for more information about what is returned and what it means.

```
virtual int operator()(CTCLInterpreter& interp,
                       std::vector<CTCLObject>& objv) = 0;
```

This pure virtual function must be overridden in concrete object command processors. The function is called to execute the command that this object is performing. *`interp`* provides a reference to the interpreter on which the command is being run. *`objv`* is a reference to a std::vector<CTCLObject>. Each element of *`objv`* is a `CTCLObject` containing a word of the command line that invoked us.

The function should return `TCL_OK` on success and `TCL_ERROR` on failure. Other return values are possible for e.g. commands that implement new control structures however this is beyond the scope of this manpage. If the command processor wants to make a result available to the interpreter, it can create a `CTCLResult` object, fill it in and commit it.

```
virtual void onUnregister();
```

This function is called when the interpreter is being destroyed or if the command is being unregistered either due to object destruction or a call to `unregister`. The default behavior is to do nothing, but this can be overidden in your derived class if desired.

SEE ALSO

CTCLCompatibilityProcessor(3), CTCLInterpreter(3), CTCLInterpreterObject(3), CTCLObject(3), CTCLProcessor(3), CTCLResult(3), Tcl_CreateObjCommand(3tcl), Tcl_GetCommandInfo(3tcl)

CTCLVariable

Name

`CTCLVariable` — Encapsulate Tcl interpreter variables.

Synopsis

```
#include <TCLVariable.h>

class CTCLVariable  : public CTCLInterpreterObject
{
public:
  CTCLVariable (std::string am_sVariable,  Bool_t am_fTracing  );
  CTCLVariable (CTCLInterpreter* pInterp,
                std::string am_sVariable,  Bool_t am_fTracing  );
  CTCLVariable (const CTCLVariable& aCTCLVariable );

  CTCLVariable& operator= (const CTCLVariable& aCTCLVariable);
  int operator== (const CTCLVariable& aCTCLVariable) const;

  std::string getVariableName() const;
  Bool_t IsTracing() const;

  void setVariableName (const std::string am_sVariable);
  virtual   char*  operator() (char* pName,
                               char* pSubscript,
                               int Flags)  ;

   static  char* TraceRelay (ClientData pObject, Tcl_Interp* pInterpreter,
                             tclConstCharPtr  pName,
                             tclConstCharPtr pIndex,
                             int flags)  ;

  const char* Set (const char* pValue, int flags=TCL_LEAVE_ERR_MSG |
                                                 TCL_GLOBAL_ONLY)  ;
  const char* Set (const char* pSubscript, char* pValue,
                   int flags=TCL_LEAVE_ERR_MSG | TCL_GLOBAL_ONLY)  ;
  const char* Get (int flags=TCL_LEAVE_ERR_MSG | TCL_GLOBAL_ONLY,
                   char* pIndex=0)  ;
  int Link (void* pVariable, int Type)  ;
  void Unlink ()  ;
  int Trace (int flags=TCL_TRACE_READS | TCL_TRACE_WRITES | TCL_TRACE_UNSETS,
             char* pIndex = (char*)kpNULL)  ;

  void UnTrace ()  ;
};
```

DESCRIPTION

`CTCLVariable` allows an existing or new Tcl interpreter variable to be encapsulated so that it can be accessed, traced or linked in C++ code.

METHODS

```
CTCLVariable(std::string sName,
             Bool_t fTracing  );
CTCLVariable (CTCLInterpreter* pInterp,
             std::string sName,
             Bool_t fTracing  );
CTCLVariable (const CTCLVariable& aCTCLVariable );
```

In the first two cases, *`sName`* is the name of the variable that will be wrapped by this object. The variable name can contain namespace qualifications as well as indices. If *`fTracing`* is true, the object is set to record that it is tracing the variable. Normally this parameter should be allowed to default to `kfFALSE`, and the trace member functions used to set explicit traces. For the final form of the constructor (copy constructor), *`aCTCLVariable`* is a `CTCLVariable` that will be copied into this object.

In the first form of the constructor, one must later call the `Bind` function (see CTCLInterpreterObject), to bind the variable to a specific interpreter prior to accessing it.

```
CTCLVariable& operator=(const CTCLVariable& rhs);
```

Assigns the *`rhs`* object to this one. A reference to the left hand side of the assignment is returned. The semantics of assignment are not that the variable values are assigned, but that the left side of the assignment becomes a functional equivalent of *`rhs`*, that is it stands for the same object and has the same traces (if any) set. Thus, if *`rhs`* wraps the interpreter variable `a` and the object on the left side wraps interpreter variable `b` after the assignment executes, the left side object will be wrapping `a`. A reference to the object on the left hand side of the assignment is returned.

```
  int operator==(const CTCLVariable& rhs) const;
```

Compares this object for functional equality with *rhs*. Functional equality is defined as the two objects referring to the same variable, in the same interpreter, and having traces set on the same operations.

```
  std::string getVariableName() const;
```

Returns the name of the Tcl variable that is wrapped by this object.

```
int Trace(int flags=TCL_TRACE_READS | TCL_TRACE_WRITES | TCL_TRACE_UNSETS,
          char* pIndex = (char*)kpNULL)  ;
void UnTrace ()  ;
Bool_t IsTracing() const;
virtual char*  operator() (char* pName,
                        char* pSubscript,
                        int Flags)  ;
```

This set of functions supports variable tracing. In Tcl, a trace is a function that is called when some event of interest occurs on a varialbe. The possible events are read, write, and unset. To effectively use variable tracing, you must create a subclass of `CTCLVariable`, override its `operator()` member to handle the trace and call `Trace` to initiate tracing.

The `Trace` member initiates tracing on the variable. *`flags`* describes when the trace should fire. See the manpage for `Tcl_TraceVar` for information about the legal flag values.

`Untrace` cancels all traces on the variable represented by this object.

`IsTracing` returns `kfTRUE` if tracing is being performed on the variable.

When a trace fires, the `operator()` member will be called. This is why you must override the `CTCLVariable` base class to do anything useful with a trace. The parameters to the call are; *`pName`* is the name of the variable that has been traced. *`pSubscript`* is the array subscript in the event the trace fires on an array or element of an array, and is `NULL` otherwise. *`Flag`* describes why the trace fired. Again, see the `Tcl_TraceVar` manpage for more information. Note that for write traces, the variable has already been set. Modifying the value of the traced variable within a trace function will not fire any additional traces. The `operator()` function must return a `NULL` pointer if the trace is successful. It must return a pointer to an error message if the trace is not successful. An example of an unsuccessful trace might be a write trace that ensures that only particular values are assigned to the variable.

```
const char* Set(const char* pValue,
                int flags=TCL_LEAVE_ERR_MSG |
                          TCL_GLOBAL_ONLY)  ;
const char* Set(const char* pSubscript,
               char* pValue,
               int flags=TCL_LEAVE_ERR_MSG | TCL_GLOBAL_ONLY)  ;
```

Sets the value of the variable to the string pointed to by *`pValue`* The second form of this assumes that the `CTCLVariable` represents an array and the *`pSubscript`* parameter specifies the subscript of the array that is being set. The *`flags`* parameter is fully documented in the Tcl manpage for Tcl_SetVar

```
const char* Get(int flags=TCL_LEAVE_ERR_MSG | TCL_GLOBAL_ONLY,
              char* pIndex=0)  ;
```

Retrieves the current value of a variable. If the *`pIndex`* parameter is supplied, the variable wrapped by `CTCLVariable` is assumed to be an array and *`pIndex`* points to the subscript of the element to retrieve. The *`flags`* parameter is fully documented in the Tcl_GetVar manpage. The return value of the function is a null terminated character string that is the current value of the variable. If the variable does not exist, then a `NULL` is returned.

```
int Link(void* pVariable,
         int Type)  ;
void Unlink()  ;
```

`Link` and `Unlink` support variable linking. Variable linking is when a Tcl variable is made to track the value of a C/C++ variable or C++ member variable of an object. `Link` establishes the link. *`pVariable`* points to the C or C++ variable or member variable to link to this `CTCLVariable`. The *`Type`* parameter is one of following values: `TCL_LINK_INT`, `TCL_LINK_DOUBLE`, `TCL_LINK_BOOLEAN`, `TCL_LINK_WIDE_INT`, or `TCL_LINK_STRING` indicating the type of the variable to which *`pVariable`* points. For all but `TCL_LINK_STRING`, *`pVariable`* points to a variable of the type indicated, and that variable will be linked. for `TCL_LINK_STRING`, *`pVariable`* points to a char* which should be initialized to point to `NULL`. The Tcl interpreter will use `Tcl_Alloc` and `Tcl_Free` to maintain a dynamically allocated string pointed to by that pointer which reflects the value of the variable. If the C/C++ program modifies this string, it must `Tcl_Free` the prior value and `Tcl_Alloc` a new value with the new variable value.

DEFECTS

No `operator!=` has been defined.

There is no protection against multiple links... the most recent link for an underlying Tcl variable is the one effective.

SEE ALSO

CTCLInterpreter(3), CTCLInterpreterObject(3), Tcl_GetVar(3tcl), Tcl_LinkVar(3tcl), Tcl_SetVar(3tcl), Tcl_TraceVar(3tcl)

CTCLProcessor

Name

`CTCLProcessor` — Provide *`argc`*, *`argv`* extension commands to Tcl.

Synopsis

```
#include <TCLProcessor.h>
...
class CTCLProcessor : public CTCLInterpreterObject
{
public:
  CTCLProcessor(const std::string sCommand, CTCLInterpreter* pInterp);
  CTCLProcessor(const char*       pCommand, CTCLInterpreter* pInterp);
  virtual ~CTCLProcessor();

  std::string getCommandName() const;

  virtual int operator()(CTCLInterpreter& rInterpreter,
                         CTCLResult&      rResult,
                         int argc, char** argv) = 0;
  virtual void OnDelete();
  void Register();
  void Unregister();

  static  std::string ConcatenateParameters (int nArguments,
                                             char* pArguments[])  ;
  int ParseInt (const char* pString, int* pInteger)  ;
  int ParseInt (const std::string& rString, int* pInteger)

  int ParseDouble (const char* pString, double* pDouble)  ;
  int ParseDouble (const std::string& rString, double* pDouble)

  int ParseBoolean (const char* pString, Bool_t* pBoolean)  ;
  int ParseBoolean (const std::string& rString, Bool_t* pBoolean)

  static int MatchKeyword(std::vector<std::string>& MatchTable,
                          const std::string& rValue,
                          int NoMatch = -1);


};
```

DESCRIPTION

`Do not use this for new commands`

The `CTCLProcessor` provides a compatibility interface to the old Tcl style *`argc`*, *`argv`* style of command extension. New commands should be written using the `CTCLObjectProcessor` class instead.

To extend the interpreter using this mechanism, you must derive a class from `CTCLProcessor` and minimally override and implement its `operator()` pure virtual function. You may optionally overrid its `OnDelete` member as well. Having written the class, you must create an object of that class and register it on an interpreter. Once the class is registered, invocations of the command under which it was registered will invoke your `operator()`.

If the interpreter is destroyed or the object destroyed, or unregistered, the `OnDelete` member will be called. `CTCLProcessor` defines and implements this function with an empty body, so it is only necessary for you to override and implement this if you have some cleanup actions that must be done when the command is deleted.

This class is now implemented in terms of a `CTCLObjectProcessor` derived class called a `CTCLCompatibilityProcessor`. It is less efficient to use this class than to use a class derived directly from a `CTCLObjectProcessor`. This class is therefore not recommended for use with new extensions, but is only provided for compatibility with existing extensions written before `CTCLObjectProcessor` was developed.

METHODS

```
CTCLProcessor(const std::string sCommand,
              CTCLInterpreter* pInterp);
CTCLProcessor(const char*       pCommand,
              CTCLInterpreter* pInterp);
```

Constructs a `CTCLProcessor`. *`sCommand`* or *`pCommand`* are the initial name of the command. Note that the Tcl interpreter supports command renaming at the script level, so there is no gaurentee that this will always be the name of the command. *`pInterp`* is the interpreter on which the command will be registered when the `Register` function is called.

```
std::string getCommandName() const;
```

Returns the initial name of the command. Unlike `CTCLObjectCommand`::`getName`() this function does not track changes in the command name.

```
virtual int operator()(CTCLInterpreter& rInterpreter,
                       CTCLResult& rResult,
                       int argc,
                       char** argv) = 0;
virtual void OnDelete();
```

`operator()` is a pure virtual function and therefore must be overidden and implemented in concrete command implementations. *`rInterpreter`* is a reference to the interpreter that is executing this command. *`rResult`* is a reference to a `CTCLResult` object that represents the result of the interpreter. Any text stored into this object will be made available to the interpreter as the result of the command. *`argc`* and *`argv`* are the number of words on the command line and a pointer to an array of pointers to the command words respectively.

`operator()` should be written to return `TCL_OK` if it is successful and `TCL_ERROR` if it encounters an error. Other return values are possible and meaningful for commands that implement flow of control structures, but documenting these is beyond the scope of this manpage. See the return(3tcl) manpage for more information about these.

`OnDelete` is called whenever the interpreter or the object is being destroyed, or the object's `Unregister` function has been called. `CTCLProcessor` provides a default implementation for `OnDelete` which does nothing. It is only necessary to override and implement this function if you require specific action when the command is being unregistered.

```
void Register();
void Unregister();
```

These functions register and unregister the command with the intepreter respectively.

```
static  std::string ConcatenateParameters(int nArguments,
                                           char* pArguments[])  ;
```

Concatenates all of the *`nArguments`* words in the *`pArguments`* array into a std::string and returns it. The words are space separated.

```
int ParseInt(const char* pString,
             int* pInteger)  ;
int ParseInt(const std::string& rString,
             int* pInteger)
```

Parses the character string *`pString`* or *`rString`* as a 32 bit signed integer into *`pInteger`*. Returns `TCL_OK` if successful, or `TCL_ERROR` if the string coult no be parsed. In that case, the result string of the interpreter will report why the string could not be parsed.

```
int ParseDouble(const char* pString,
                double* pDouble)  ;
int ParseDouble(const std::string& rString,
                double* pDouble)
```

Parses the input string, either *`pString`* or *`rString`* as a double precision floating point value, storing the result in the double pointed to by *`pDouble`*. On success, `TCL_OK` is returned. On failure, `TCL_ERROR` and the interpreter result is a textual reason for the failure.

```
int ParseBoolean(const char* pString,
                 Bool_t* pBoolean)  ;
int ParseBoolean(const std::string& rString,
                 Bool_t* pBoolean)
```

Parses either *`pString`* or *`rString`* as a boolean value. The result is stored in boolean pointed to by *`pBoolean`*. `TCL_OK` is returned on success, `TCL_ERROR` on error. If `TCL_ERROR` was returned, the interpreter result is the textual reason for the failure.

```
static int MatchKeyword(std::vector<std::string>& MatchTable,
                        const std::string& rValue,
                        int NoMatch = -1);
```

Searches for the string *`rValue`* in the vector of strings *`MatchTable`*, and returns the index in the vector at which the match occured. If no match could be found, the value *`NoMatch`* is returned.

Within SpecTcl, this is often used to match command switches.

SEE ALSO

CTCLCompatibilityProcessor(3), CTCLObjectProcessor(3), CTCLPackagedCommand(3), CTCLResult(3), return(3tcl)

CTCLChannel

Name

`CTCLChannel` — Provide a C++ abstraction wrapper for Tcl Channels.

Synopsis

```
#include <TCLChannel.h>
...
class CTCLChannel : public CTCLInterpreterObject
{
public:

  CTCLChannel(CTCLInterpreter* pInterp,
              std::string      Filename,
              const char*      pMode,
              int              permissions);
  CTCLChannel(CTCLInterpreter* pInterp,
              int              argc,
              const char**           pargv,
              int              flags);
  CTCLChannel(CTCLInterpreter* pInterp,
              int              port,
              std::string      host)
  CTCLChannel(CTCLInterpreter* pInterp,
              int              port,
              Tcl_TcpAcceptProc* proc,
              ClientData       AppData);
  CTCLChannel(CTCLInterpreter* pInterp,
              Tcl_Channel      Channel);
  CTCLChannel(const CTCLChannel& rhs);
  virtual ~CTCLChannel();

  Tcl_Channel getChannel() const;
  bool ClosesOnDestroy() const;

  int Read( void** pData, int nChars);
  int Write(const void* pData, int nBytes);

  bool atEof();
  void Flush();
  void Close();
  void Register();
  void SetEncoding(std::string Name);
  std::string GetEncoding();
```

DESCRIPTION

The Tcl API supplies an I/O abstraction layer on top of the operating systems I/O subsystem. This layer makes use of what Tcl documentation refers to a s *channels*. A channel represents a connection to an I/O endpoint (source or sink of data or both). The `CTCLChannel` class allows you to wrap a C++ class around a Tcl channel.

Using a `CTCLChannel`, rather than direct operating system I/O allows you to improve the portability of your program, as well as allowing I/O from the TCL scripting level to be cleanly mixed with I/O at the C/C++ level.

METHODS

```
CTCLChannel(CTCLInterpreter* pInterp,
            std::string      Filename,
            const char*      pMode,
            int              permissions);
CTCLChannel(CTCLInterpreter* pInterp,
            int              argc,
            const char**           pargv,
            int              flags);
CTCLChannel(CTCLInterpreter  * pInterp,
            int              port,
            std::string      host);
CTCLChannel(CTCLInterpreter* pInterp,
            int              port,
            Tcl_TcpAcceptProc* proc,
            ClientData      AppData);
CTCLChannel(CTCLInterpreter* pInterp,
            Tcl_Channel      Channel);
CTCLChannel(const CTCLChannel& rhs);
```

These constructors wrap a `CTCLChannel` object around a Tcl channel. The variety of constructors reflects the variety of endpoints around which a Tcl_Channel can be wrapped.

The first of the constructors attaches the object to a file via `Tcl_OpenFileChannel`. *`pInterp`* is the channel the file is associated. *`Filename`* is the name of the file the channel is connected to. *`pMode`* is the connection mode which can be any of of the mode values for the Tcl **open** scripting command. *`permissions`* represents the permissions mask fo rthe file in POSIX format. See open(2) for information about the possible mode bit values.

The second form of the constructor connects a channel that is a pipe to a program. The program and its command line arguments are specified via the *`argc`* and *`pargv`* parameters. The *`flags`* parameter specifies how the stdio channels of the program are or are not disposed into the channel. Valid bits are: `TCL_STDIN`, `TCL_STDOUT`, `TCL_STDERR`, `TCL_ENFORCE_MODE`. See the `Tcl_OpenCommandChannel` manpage for information about the meaning of these bits.

The third form of the constructor constructs a channel that connects to a TCP/IP server, via `Tcl_OpenTcpClient`. The *`port`* parameter specifies the port number on which the server is listening, while *`host`* is the name of the host to which the connection should be formed. The host name can either be a DNS resolvable host name or the textual encoding of the TCP/IP address of the host (e.g. string("spdaq22.nscl.msu.edu") or string("35.9.56.56"). This function will block until the connection is accepted by the server.

The fourth form of the constructor creates a channel that is a Tcp/IP server. *`port`* is the port number on which the server listens for connections. *`proc`* is a function that will be called when a connection has been accepted by the Tcl event loop. *`AppData`* is application data that is passed, without interpretation to *`proc`*. See the `Tcl_OpenTcpServer` manpage for more information about how the *`proc`* is called.

The fifth form of the constructor creates a `CTCLChannel` object by wrapping an existing Tcl_Channel; *`Channel`* which has been obtained directly from the Tcl application programming interface.

The final form of the constructor copies an existing `CTCLChannel` object so that the two objects refer to the same channel. The object is created so that it will not close the channel on destruction. Note however that the source channel may, depending on how it was constructed. It is up to the application programmer to ensure that channels are closed at appropriate times.

```
Tcl_Channel getChannel() const;
```

Returns the underlying Tcl_Channel this object is wrapped around. Once this is obtained, it can be usd in any Tcl API call that requires a channel.

```
bool ClosesOnDestroy() const;
```

When a `CTCLChannel` is copy constructed into being it is flagged such that destruction will not close the underlying channel. the return value from this function is `true` if the object will close the underlying channel on destruction and `false` if the object will not close the underlying channel on destruction.

```
int Read( void** pData,
          int nChars);
int Write(const void* pData,
          int nBytes);
```

`Read` transfers data from the channel to the users's buffer; *`pData`*. *`nChars`* is the number of characters of data that will be transferred. `Write` transfers *`nBytes`* *bytes* data to the chnnel from *`pData`*. Both function return the number of characters actually transferred. An important note about the *`nChars`* parameter: If the channel is not opened as a binary channel, this parameter is the number of UTF-8 characters transferred. Depending on the characters actually transmitted, this may not be the same as the number of bytes transmitted. The return value is the number of units (bytes or characters) actually transferred.

```
bool atEof();
```

Returns `true` if the underlying channel is at the end of file.

```
void Flush();
```

Tcl channels are internally buffered. The `Flush` member flushes internal output buffers to the underlying I/O endpoint.

```
void Close();
```

Closes the underlying channel. Note that this is normally done on destruction unless the channel object was created via copy construction. If the channel was registered to be visible to the interpreter, it is unregistered as well.

```
void Register();
```

Makes the channel visible to the interpreter. This allows the user to return the channel name to the script level at which point it can be used in Tcl script commands that operate on channels.

```
  void SetEncoding(std::string Name);
  std::string GetEncoding();
```

These functions allow the user to get and set the encoding for the channel. See the **fconfigure** Tcl man page for more information about this.

SEE ALSO

close(3tcl), fconfigure(3tcl), open(2), Tcl_OpenCommandChannel(3tcl), Tcl_OpenFileChannel(3tcl), Tcl_OpenTcpClient(3tcl), Tcl_OpenTcpServer(3tcl)

CTCLCommandPackage

Name

`CTCLCommandPackage` — Group several related Tcl command extensions and common services they may require together.

Synopsis

```
#include <TCLCommandPackage.h>
...
typedef std::list <CTCLProcessor*>   CommandList;
typedef CommandList::iterator  CommandListIterator;

class CTCLCommandPackage  : public CTCLInterpreterObject
{

public:
  CTCLCommandPackage (CTCLInterpreter* pInterp,
                      const std::string& rSignon=std::string("Unnamed pkg"));
  CTCLCommandPackage(CTCLInterpreter* pInterp,
                     const char* pSignon = "Unnamed pkg");
  virtual ~ CTCLCommandPackage ( );
  CTCLCommandPackage (const CTCLCommandPackage& aCTCLCommandPackage );
  CTCLCommandPackage& operator= (const CTCLCommandPackage& aCTCLCommandPackage);
  int operator== (const CTCLCommandPackage& aCTCLCommandPackage);

  std::string getSignon() const;
  CommandList getCommandList() const;
protected:
  void setSignon (std::string am_sSignon);

public:
  void Register ()  ;
  void Unregister ()  ;
  void AddProcessor (CTCLProcessor* pProcessor);
  void AddProcessors(CommandList& rList);
  CommandListIterator begin ();
  CommandListIterator end ();
};
```

DESCRIPTION

Extensions to Tcl often come in a set of related commands. These commands may require access to a common set of services. The `CTCLCommandPackage` along with the `CTCLPackagedCommand` provide a pair of base classes that facilitate the construction of such commands.

The pattern to follow to derive class from `CTCLCommandPackage` This class defines and implements common services for the related commands. The constructor of the derived class will also create instances of classes derived from `CTCLPackagedCommand`. These objects define and implement the related commands. These command processors will be added to the package via `AddProcessor`, and `AddProcessors`.

When the `CTCLCommandPackage::Register` function is called, all of the commands added to the package will be registered as well. When a command processor is invoked, it can call its `getMyPackage` member function to obtain a pointer to the owning package and therefore access to the services this package provides.

METHODS

```
  CTCLCommandPackage(CTCLInterpreter* pInterp,
                      const std::string& rSignon=std::string("Unnamed pkg"));
  CTCLCommandPackage(CTCLInterpreter* pInterp,
                     const char* pSignon = "Unnamed pkg");
  CTCLCommandPackage (const CTCLCommandPackage& aCTCLCommandPackage );
```

Constructs instances of the package. *`pInterp`* is a pointer to the interpreter object on which these commands will be registered. *`signon`* is a text string that will be emitted to `stderr` when the package is asked to register its commands. This string is typically a credit or copyright notice for the package. It can be empty if the user desires.

The first and second form of the constructor only differ in how the signon message is passed. The final form of the constructor is a copy constructor. While copy construction is legal it is anticipated that this will not normally be used as command packages are usually *singleton* objects.

```
std::string getSignon() const;
```

Retrieves the signon string from the current object.

```
CommandList getCommandList() const;
```

Retrieves the list of commands that are managed by this package.

```
void setSignon (std::string am_sSignon);
```

Allows derived classes to set the signon string after construction is complete.

```
void Register ()  ;
void Unregister ()  ;
```

`Regsiter` registers all of the commands in the package with the package's interpreter. `Unregister` unregisters these commands. It is therefore not advisable to change the set of commands in the package between registration and unregistration.

```
void AddProcessor(CTCLProcessor* pProcessor);
void AddProcessors(CommandList& rList);
```

These functions add command processors to the package. Any type of processor can be added to the package, however usually `CTCLPackagedCommand` derived objects are in order to provide a mechanism to access the package services. *`pProcessor`* is a pointer to a single processor while *`rList`* is a reference to a list of such processors.

```
CommandListIterator begin ();
CommandListIterator end ();
```

Returns STL list iterators to the beginning and off the end of the set of command packages. List iterators behave roughly like pointers. In this case, pointers to `CTCLProcessor*`. Incrementing an interator 'points' it to the next item in the list. A full discussion of STL iterators is well beyond the scope of this man page. See references below.

SEE ALSO

CTCLProcessor(3), CTCLPackagedCommand(3)

References

```
Musser, Derge, Saini: STL Tutorial and Reference Guide
Addison-Wesley Professional Computing Series; 2001 ISBN 0-201-37923-6
```

CTCLCompatibiltyProcessor

Name

`CTCLCompatibilityProcessor` — Adaptor between `CTCLOjbectProcessor` and `CTCLProcessor`.

Synopsis

```
#include <TCLCompatibilityProcessor.h>
...
class CTCLCompatibilityProcessor : public CTCLObjectProcessor
{

public:

  CTCLCompatibilityProcessor(CTCLProcessor& actualCommand);
  virtual ~CTCLCompatibilityProcessor();

  virtual int operator()(CTCLInterpreter& interp,
                         std::vector<CTCLObject>& objv);
  virtual void onUnregister();
};
```

DESCRIPTION

Note that the information on this page is only to be used to "grandfather" old `CTCLProcessor` objects into Tcl. New command processors should be based on the CTCLObjectProcessor.

A rewrite of the TCL++ library in March 2006 eliminated the use of Tcl functions that are scheduled to be deprecated in the known future of Tcl. One set of functions scheduled for deprecation are those that revolve around direct access to the Tcl interpreter result, and command processors that use the *`argc`*, *`argv`* interfaces.

Since a number of existing command extensions are, no doubt built on top of the `CTCLProcessor` *`argc`* *`argv`* interface, a mechanism that allows the use of existing commands was built as well.

The current implementation of a `CTCLProcessor` includes as member data a `CTCLCompatibilityProcessor` object. This object registers itself as a `CTCLObjectProcessor` for the command intended by the `CTCLProcessor` and serves as an adaptor between these two command types, marshalling command objects into an *`argc`*, *`argv`* pair, creating and committing a `CTCLResult` object.

For most users of this library, this is completely transparent, this documentation is provided for completeness, however.

METHODS

```
CTCLCompatibilityProcessor(CTCLProcessor& actualCommand);
```

Constructs a compatibility processor that adapts the object based command interface for `actualCommand`. `actualCommand` must already be bound to an interpreter.

```
virtual int operator()(CTCLInterpreter& interp,
                       std::vector<CTCLObject>& objv);
```

Called when the command is executed. `objv` is marshalled into an argc, argv pair. A `CTCLResult` is created and the actual command's `operator()` is called to execute the command. On return, the result is committed, and all dynamic storage released prior to returning the status from the command processor.

```
virtual void onUnregister();
```

Called when the command is unregistered. The actual command's `OnDelete` member is called.

SEE ALSO

CTCLObject(3), CTCLObjectProcessor(3), CTCLProcessor(3), CTCLResult(3),

REFERENCES

Gamma, Helm, Johnson, Vlissides *Design Patterns Elements of Reusable Object-Oriented Software*
Addison-Wesley Professional Computing Series 1995 ISBN 0-0201-63361-2

```
See Chapter 4 the Adapter pattern.
```

CTCLFileHandler

Name

`CTCLFileHandler` — Base class for building object oriented Tcl File event handlers.

Synopsis

```
#include <TCLFileHandler.h>
...

class CTCLFileHandler  : public CTCLInterpreterObject
{
  CTCLFileHandler(CTCLInterpreter* pInterp,
                   UInt_t am_nFid = STDIN_FILENO);
  CTCLFileHandler(CTCLInterpreter* pInterp,
                  FILE* pFile);
  CTCLFileHandler (const CTCLFileHandler& aCTCLFileHandler );
  ~CTCLFileHandler ( );
  CTCLFileHandler& operator= (const CTCLFileHandler& aCTCLFileHandler);
  int operator== (const CTCLFileHandler& aCTCLFileHandler) const;

  UInt_t getFid() const;

  virtual   void operator() (int mask)   = 0;

  void Set (int mask)  ;
  void Clear ()  ;
};
```

DESCRIPTION

Tcl supplies an event loop. It is possible to add events specifications to this loop. One very useful event type is based on readability or writability of a file descriptor. The `CTCLFileHandler` allows you to create an object oriented file handler, and register it with the event loop so that you can gain control when, for example, a file becomes readable.

As `CTCLFileHandler` is an abstract base class, it is necessary to create a derived class. The derived class should implement the `operator()` which will be called when the specific event is fired. An instance of this derived class should be created, and then the `Set` and `Clear` members used to establish and remove the event handler.

METHODS

```
CTCLFileHandler(CTCLInterpreter* pInterp,
                 UInt_t nFid = STDIN_FILENO);
CTCLFileHandler(CTCLInterpreter* pInterp,
                FILE* pFile);
CTCLFileHandler (const CTCLFileHandler& aCTCLFileHandler );
```

Constructs a file handler object. *`pInterp`* is a pointer to the interpreter on which the file handler will be registered. The file can be specified either by *`nFid`*, a file descriptor, or *`pFile`* an stdio File stream pointer.

A copy constructor allows the creation of a copy of the file handler object given *`aCTCLFileHandler`* an existing one. This is normally not useful.

```
CTCLFileHandler& operator= (const CTCLFileHandler& aCTCLFileHandler);
int operator== (const CTCLFileHandler& aCTCLFileHandler) const;
```

These functions support assignment and equality comparison. Note that these functions are usually not very useful for file handlers.

```
UInt_t getFid() const;
```

Returns the file id that is associated with the event.

```
virtual   void operator() (int mask)   = 0;
```

The user's derived class must override this to provide the desired funtionality when event fires. `mask` indicates which event fired the function and can be an or of the following: `TCL_READABLE` if the file can be read without blocking, `TCL_WRITABLE` if the file can be written without blocking, or `TCL_EXCEPTION` if some exceptional condition occured on the file.

```
  void Set (int mask)  ;
  void Clear ()  ;
```

`Set` establishes the event handler for the set of conditions described in *`mask`*. The valid bits for *`mask`*, are described in the documentation for the *`mask`* parameter to `operator()`.

SEE ALSO

CTCLObject(3), CTCLInterpreter(3) Tcl_CreateFileHandler(3tcl), Tcl_DeleteFileHandler(3tcl)

CTCLHashTable

Name

`CTCLHashTable` — Object oriented interface to Tcl's hash table functions.

Synopsis

```
#include <TCLHashTable.h>

template <class T>
class CTCLHashTable
{
public:
  CTCLHashTable () ;
  CTCLHashTable (  Tcl_HashTable am_HashTable  );
  CTCLHashTable (const CTCLHashTable& aCTCLHashTable );
  virtual ~CTCLHashTable ( );

  CTCLHashTable operator= (const CTCLHashTable& aCTCLHashTable);
  int operator== (const CTCLHashTable& aCTCLHashTable);

  Tcl_HashTable* getHashTable() const;

  void Enter (const std::string& rKey, rCTCLTHashTableItem rValue);
  const CTCLTHashTableItem* Find (const std::string& rsKeyword) const;
  CTCLTHashTableItem* Delete (const std::string& rsKeyword);
  CTCLTHashTableIterator begin ();
  CTCLTHashTableIterator end ();
  std::string Statistics ();
};
```

DESCRIPTION

Hash tables are tables of keyword value pairs that are organized such that the lookup time for any key in the table is *amortized constant*. Hash tables operate by running the key through a function called the *hash function*, and storing the key/value pair as an element of an array indexed by the result of that hash function (*hash index*). Depending on the implementation of the hash table, different methods are used to resolve cases where two keys result in the same hash index.

Tcl includes support libraries for hash tables with string keys and arbitrary value types (e.g. structures, pointers etc. etc.). One example of the use of this sort of data structure is Tcl's storage of array variables. Each array is a hash table indexed by the hash index of the array subscripts. In this way Tcl supports subscripts that are arbitrary strings without any search overhead when referencing an element of the array.

The `CTCLHashTable` and related classes provide an object oriented interface to the Tcl API for hash tables. This class is a *template class*. The template parameter is the type of data that will be associated with each hash key. For example, to create a has key of `CSpectrum*` (pointers to SpecTcl Spectra):

```
CTCLHashTable<CSpectrum*> spectrumHashTable;
```

METHODS

```
  CTCLHashTable () ;
  CTCLHashTable (  Tcl_HashTable aHashTable  );
  CTCLHashTable (const CTCLHashTable& aCTCLHashTable );
```

Three methods for creating `CTCLHashTable` objects are defined. The first of these creates a new, empty hash table. The second, takes the handle to an existing hash table; Tcl_HashTable *`aHashTable`* and wraps a `CTCLHashTable` around this existing hash table providing an object oriented interface to that hash table. The final constructor, a copy constructor, creates a `CTCLHashTable` that refers to the same underlying Tcl_HashTable as the *`aCTCLHashTable`* parameter.

```
  CTCLHashTable operator= (const CTCLHashTable& rhs);
  int operator== (const CTCLHashTable& rhs);
```

`operator=` assigns *`rhs`* to an existing object. The semantics of assignment are that followingt assignment, `*this` and *`rhs`* will refer to the same underlying hash table.

`operator==` compares two hash tables, `*this` and *`rhs`* for equality. The semantics of equality are that the two `CTCLHashTable` objects refer to the same underlying Tcl hash tables.

```
  Tcl_HashTable* getHashTable() const;
```

Gets the underlying Tcl_HashTable that is wrapped by a `CTCLHashTable` object.

```
void Enter (const std::string& rKey,
            CTCLTHashTableItem<T> rValue);
```

Adds an entry to a hash table. *`rKey`* is the lookup key that will be associated with the entry. *`rValue`* is the data that is associated with that key. Note that T is the template type that was used to create the hashtable. E.g. if the hash table is a `CTCLHashTable<float>`, *`rValue`* must be a `CTCLHashTableItem<float>`. Note that Tcl hash tables do not support duplicate keys. If a hash table entry with the key *`rKey`* already is in the table it is replaced.

```
  const CTCLTHashTableItem* Find (const std::string& rsKeyword) const;
```

Looks up a hash table item by key. If a hash table item with the key *`rsKeyword`* exists, a pointer to its entry is returned. If *`rsKeyword`* has not yet been `Enter`ed in the hash table, a `NULL` pointer is returned.

```
CTCLTHashTableItem* Delete (const std::string& rsKeyword);
```

Removes the hash table entry with the key *`rsKeyword`*. If the item existed, a pointer to it is returned. If the item does not exist in the hash table a `NULL` pointer is returned.

```
CTCLTHashTableIterator begin ();
CTCLTHashTableIterator end ();
```

`begin` returns an *iterator* that "points" to the first entry in the hash table. dereferencing the iterator yields the pointer to a HashTableItem. The iterator can be incremented via ++ so that it advances to the next item in the table.

`end`returns an iterator that points past the end of the table and can be used to determine when iteration is complete.

Iterators are pointer like objects. See the STL reference below for more information about them. The following example Takes a Hash table and counts up the number of elements it contains.

```
CTCLHashTableIterator i = table.begin();    // Table a CTCLHashTable
int                   n = 0;
while (i != table.end()) {
    n++;
}
// N is a count of elements in the table.
```

```
std::string Statistics ();
```

Returns a string that contains statistics about the hash table. This is a wrapper for `Tcl_HashStats`

SEE ALSO

CTCLHashTableItem(3), CTCLHashTableIterator(3), Tcl_HashStats(3tcl)

REFERENCES

```
Niklaus Wirth Algorithms + Data Structures = Programs
Prentice Hall Series in Automatic Computation 1976 See section 4.6

Musser, Derge, Saini: STL Tutorial and Reference Guide
Addison-Wesley Professional Computing Series; 2001 ISBN 0-201-37923-6
```

CTCLHashTableItem

Name

`CTCLHashTableItem` — Encapsulation of an entry in a Tcl Hash table as encapsulated in `CTCLHashTable`

Synopsis

```
#include <TCLHashTableItem.h>
...

template <class T>              // T Must have copy ctor & Assignment defined.
class CTCLHashTableItem
{
  CTCLHashTableItem (T am_Item );
  CTCLHashTableItem (const CTCLHashTableItem& aCTCLHashTableItem );
  virtual ~CTCLHashTableItem ( );

  CTCLHashTableItem operator= (const CTCLHashTableItem& aCTCLHashTableItem);
   int operator== (const CTCLHashTableItem& aCTCLHashTableItem);
  T getItem() const;
  T* operator-> ();
}
```

Provides an object oriented interface to elements of a hash table. See CTCLHashTable(3), and the first REFERENCE below for more information about hash tables.

Note that this is a templated class. The template parameter is the type of the item to be stored in the hash table. Suppose, for example, we have a bunch of named calibration parameters (floating point) that are stored in a hash table. A calibration The following code creates the calibration table, and inserts an element named george in it with the initial value of 0.0:

```
        CTCLHashTable<float>  calibrationTable;
        CTCLHashTableItem<float> entry(0.0);
        calibrationTable.Enter(std::string("george"), entry);
```

METHODS

```
CTCLHashTableItem (T Item );
CTCLHashTableItem (const CTCLHashTableItem& hashItem );
```

Constructs a `CTCLHashTableItem` either from the underlying type (*`Item`*) or from an existing `CTCLHashTableItem` (*`hashItem`*).

```
CTCLHashTableItem operator=(const CTCLHashTableItem& rhs);
int operator== (const CTCLHashTableItem& rhs);
```

`operator=` allows you to assign the value of one `CTCLHashTableItem` to another. The underlying templated type must be capable of assignment.

`operator==` allows you to compare two `CTCLHashTableItem`s to each other. The underlying templated type must be capable of equality comparison.

```
T getItem() const;
```

Returns the value of the item wrapped by the `CTCLHashTableItem` The underlying type must be capable of copy construction.

```
T* operator-> ();
```

Returns the address of the contents of the `CTCLHashTableItem`. This is most useful if `T` is a structure or class as it can be used to dereference member (data or functions) of the structure or class. For example:

```
struct complex {
                 double real;
                 double imaginary;
                 complex(double r, double i) :
                    real(r), imaginary(i) {}
                } complex;
CTCLHashTableItem<complex> v(1.0, 2.0);
double real = v->real;             // real = 1.0
double imag = v->imaginary;        // imag = 2.0
```

SEE ALSO

CTCLHashTable(3), CTCLHashTableIterator(3)

REFERENCES

Niklaus Wirth *Algorithms + Data Structures = Programs*
Prentice Hall Series in Automatic Computation 1976 See section 4.6

CTCLHashTableIterator

Name

`CTCLHashTableIterator` — Iterator for visiting all elements of a `CTCLHashTable`

Synopsis

```
#include >TCLHashTableIterator.h>
...
template <class T>
class CTCLHashTableIterator
{

public:
  CTCLHashTableIterator (Tcl_HashTable*   pTable);
  CTCLHashTableIterator (const CTCLHashTableIterator& aCTCLHashTableIterator );
  virtual ~ CTCLHashTableIterator ( );

  CTCLHashTableIterator operator=
                     (const CTCLHashTableIterator& aCTCLHashTableIterator);
  int operator== (const CTCLHashTableIterator& aCTCLHashTableIterator);
  CTCLHashTableItem<T>* getCurrentEntry() const;
  Tcl_HashTable* getHashTable() const;

  CTCLHashTableIterator& operator++ ();
  CTCLHashTableIterator operator++ (int i);
  CTCLHashTableItem<T>& operator* ();
  CTCLHashTableItem<T>* operator->();

};
```

DESCRIPTION

`CTCLHashTableIterator` objects are created and returned by `CTCLHashTableIterator::begin` and `CTCLHashTableIterator::end`. These objects are pointer like objects to `CTCLHashTableItem` objects within the hash table.

If you imagine that all containers can have an ordering defined on them, iterators are like pointers to elements of this ordering. Dereference operators yield an element of the container, and increment operators make the iterator ’point’ to the next element in the container according to the ordering.

For more information on both hash tables and iterators see the REFERENCES. For information about the classes that are related to this, consult manpages pointed to by the SEE ALSO section.

METHODS

```
  CTCLHashTableIterator (Tcl_HashTable* pTable);
  CTCLHashTableIterator(const CTCLHashTableIterator& rhs);
```

Construct a hash table iterator. Normally you will not need to use these constructors directly. They will be created, instead by `CTCLHashTable::begin()` or `CTCLHashTable::end()`. *`pTable`* is a pointer to an existing Tcl_HashTable created via `Tcl_InitHashTable`. *`rhs`* is an existing `CTCLHashTableIterator` object whose state will be used to initialize the object under construction.

```
  CTCLHashTableIterator operator=
                   (const CTCLHashTableIterator& rhs);
  int operator==(const CTCLHashTableIterator& rhs);
```

`operator=` allows you to assign the state of one `CTCLHashTableIterator` to another. When the assignment is complete, the left hand side object will be ’pointing’ to the same object as the right hand side object but be separately incrementable.

`operator==` allows you to compare two iterators for equality. equality is defined as the two iterators being defined on the same underlying hash table, pointing to the same element, and having the same increment context (e.g. an increment of both iterators will leave them both pointing to the same hash table item (different from the one prior to the increment).

```
CTCLHashTableItem<T>* getCurrentEntry() const;
Tcl_HashTable* getHashTable() const;
```

These two functions get at the information the iterator is encapsulating. `getCurrentEntry` returns a pointer to the entry that the iterator si currently ’pointing’ at. This is identical to the `operator->` function. `getHashTable` returns a pointer to the underlying Tcl_HashTable created by `Tcl_InitHashTable`.

```
CTCLHashTableIterator& operator++ ();
CTCLHashTableIterator operator++ (int i);
```

These two function support both pre and post increment operations on an iterator. There are slight differences in semantics between these operators best illustrated with a sample code fragment. In the fragment below, i is an `CTCLHashTableIterator`

```
CTCLHashTableItem item1 = *i++;   // item 1 is the item pointed to prior to increment
CTCLHashTableItem item2 = *++i;   // item 2 is the item pointed to after increment.
```

```
CTCLHashTableItem<T>& operator* ();
CTCLHashTableItem<T>* operator->();
```

These operators allow `CTCLHashTableIterator` objects to be treated like pointers to `CTCLHashTableItem` objects. `operator*` provides ’pointer’ dereferencing that allows code like:

```
                (*i).getItem();
```

`operator->` provides a pointer to struct like semantics allowing code like:

```
                i->getItem();
```

SEE ALSO

CTCLHashTable(3), CTCLHashTableItem(3), Tcl_InitHashTable(3tcl), Tcl_FirstHashEntry(3tcl), Tcl_NextHashEntry(3tcl)

REFERENCES

```
Niklaus Wirth *Algorithms + Data Structures = Programs*
Prentice Hall Series in Automatic Computation 1976 See section 4.6

Musser, Derge, Saini: *STL Tutorial and Reference Guide*
Addison-Wesley Professional Computing Series; 2001 ISBN 0-201-37923-6
```

CTCLIdleProcess

Name

`CTCLIdleProcess` — Allows the establishment of an executable object that can be scheduled to be invoked when the Tcl/Tk intperpreter has no events that require processing.

Synopsis

```
#include <TCLIdleProcess.h>

class CTCLIdleProcess : protected CTCLTimer
{
public:
  CTCLIdleProcess(CTCLInterpreterObject* pObject);
  CTCLIdleProcess(CTCLInterpreter* pInterp);
  virtual ~CTCLIdleProcess();

  void Set();
  void Clear();
  virtual void operator()() = 0;
};
```

DESCRIPTION

While Tcl provides a mechanism for scheduling the execution of a function when the interpreter main loop is idle (no pending events), this is not suitable for processes that may need to be rescheduled. Therefore, `CTCLIdleProcess` is actually based on a timer dispatch where the delay interval is 0ms.

`CTCLIdleProcess` provides an abstract base class for creating function like classes that are 'called' to run interleaved with the interpreter. A function like class is one that implements `operator()` (see REFERENCES) below. You can create an idle processor by creating a subclass of `CTCLIdleProcess` overriding `operator()`, creating an instance of that new class, and invoking the `Set()` function to schedule the execution of the `operator()`. Note that It is possible for the code in your `operator()` to reschedule itself by calling `Set()`.

METHODS

```
CTCLIdleProcess(CTCLInterpreterObject* pObject);
CTCLIdleProcess(CTCLInterpreter* pInterp);
```

Creates a `CTCLIdleProcess` and initializes the timer on which this is based. *`pInterp`* is the interpreter that will schedule the object's `operator()`. *`pObject`* points to an interpreter object who's interpreter will schedule the `operator()` to run.

```
void Set();
void Clear();
```

These function control the scheduling of the `operator()` call. `Set` schedules the function to be called pretty much the next time the interpreter loop is intered, while `Clear` cancels a pending schedule.

```
virtual void operator()() = 0;
```

This pure virtual function is overridden by your idle processor to provide the behavior of the idle processor.

SEE ALSO

CTCLTimer(3), Tcl_CreateTimerHandler(3tcl), Tcl_DoWhenIdle(3tcl),

REFERENCES

```
Musser, Derge, Saini: STL Tutorial and Reference Guide
Addison-Wesley Professional Computing Series; 2001 ISBN 0-201-37923-6
See section 2.4 for a description and discussion of function objects.
```

CTCLPackagedCommand

Name

`CTCLPackagedCommand` — Base class for a command that lives in a `CTCLCommandPackage`

Synopsis

```
#include <TCLPackagedCommand.h>
...
class CTCLPackagedCommand   : public CTCLProcessor
{

  CTCLPackagedCommand (const std::string& sCommand, CTCLInterpreter* pInterp,
                       CTCLCommandPackage& rPackage);
  CTCLPackagedCommand (const char* pCommand, CTCLInterpreter* pInterp,
                       CTCLCommandPackage& rPackage);
   ~ CTCLPackagedCommand ( );

  CTCLCommandPackage& getMyPackage();

  void setMyPackage (CTCLCommandPackage& am_rMyPackage);

};
```

DESCRIPTION

Command packages (see CTCLCommandPackage(3)), provide a way to organize a set of related Tcl command processors around a set of shared services. Objects derived from `CTCLPackagedCommand` are added to an object derived from `CTCLCommandPackage`. The `CTCLCommandPackage` manages bulk registration of all of the commands added to it. Construcint a `CTCLPackagedCommand` object provides it a reference to its package so that public members of the package can be invoked when the package commands are executing.

Note that since `CTCLPackagedCommand` is derived from `CTCLProcessor`, and does not supply a `operator()` You must derive concrete classes from this class implementing `operator()` to provide the desired command functionality.

METHODS

```
  CTCLPackagedCommand (const std::string& sCommand,
                       CTCLInterpreter* pInterp,
                       CTCLCommandPackage& rPackage);
  CTCLPackagedCommand> (const char* pCommand,
                       CTCLInterpreter* pInterp,
                       CTCLCommandPackage& rPackage);
```

Constructs a packaged command. *`sCommand`* or *`pCommand`* provide the command name. *`pInterp`* is a pointer to the interpreter on which the command will be registered. *`rPackage`* is a reference to the package this object will be a member of.

```
CTCLCommandPackage& getMyPackage();
```

Returns a reference to the object's package. This can be cast to the actual type of the package at which point package public members can be accessed.

```
  void setMyPackage (CTCLCommandPackage& rMyPackage);
```

Provides a new package for the command.

SEE ALSO

CTCLCommandPackage(3), CTCLProcessor(3)

CTCLResult

Name

`CTCLResult` — Provide an object oriented interace to the Tcl interpreter result.

Synopsis

```
#include <TCLResult.h>
...
class CTCLResult  : public CTCLObject
{
  CTCLResult (CTCLInterpreter* pInterp, bool reset=true );
  CTCLResult (const CTCLResult& aCTCLResult );
  virtual ~CTCLResult ( );

  CTCLResult& operator= (const CTCLResult& aCTCLResult);
  CTCLResult& operator= (const char* rhs);
  CTCLResult& operator=(std::string    rhs);

  int operator== (const CTCLResult& aCTCLResult) ;
  int operator!= (const CTCLResult& rhs);

  CTCLResult& operator+= (const char* pString);
  CTCLResult& operator+= (const std::string& rString);

  void Clear ()  ;
  void AppendElement (const char* pString)  ;
  void AppendElement (const std::string& rString);
  void commit() const;
  std::string getString();
};
```

DESCRIPTION

Each Tcl command can return *result string* the result string can be used by subsequent commands in the event the command operated successfully, or by **catch** commands if the command failed. `CTCLResult` provides an extension of the `CTCLObject` class that builds up a string which can then be comitted to the result.

METHODS

```
CTCLResult(CTCLInterpreter* pInterp,
           bool reset=true );
CTCLResult(const CTCLResult& aCTCLResult);
```

Constructs a Tcl interpreter result string. *`pInterp`* is the interpeter that will be associated with this result. *`reset`* controls whether or not the result string is reset when constructed, or if it is loaded with the current value of the result string. In the case of copy construction, the interpreter associated with *`aCTCLResult`* is used. *`aCTCLResult`* is committed to the interpreter result, and the object under construction is then loaded from that interpreter's result.

```
  CTCLResult& operator= (const CTCLResult& rhs);
  CTCLResult& operator= (const char* rhs);
  CTCLResult& operator=(std::string    rhs);
```

Assigns a value to the result from *`rhs`*. If the *`rhs`* is a `CTCLResult`, then the *`rhs`* is first committed to its interpreter result, the left hand object is then bound to the same interpreter as *`rhs`* and loaded with the result string of that interpreter.

```
  int operator== (const CTCLResult& rhs) ;
  int operator!= (const CTCLResult& rhs);
```

These functions suport comparison. Equality comparison is true (`operator==`) if the interpreters match as the assumption is that the user is working to maintain coherency if several `CTCLResult` objects are simultaneously live on a single interpreter. Inequality (`operator!=`) is defined as true when `operator==` is false.

```
CTCLResult& operator+= (const char* rhs);
CTCLResult& operator+=(const std::string& rhs);
```

`rhs` is textually appended to the result string being built up. Note that the semantics of this are different than for the base class where `operator+=` is a list append.

```
void Clear()  ;
```

Clears the result string being built up as well as clearing the underlying interpreter's result.

```
void AppendElement(const char* item)  ;
void AppendElement(const std::string& item);
```

Appends *`item`* to the result string being built up as a list element. This means that under some circumstances extra quoting may be done to ensure that the result will be maintained as a valid list.

```
void commit() const;
std::string getString();
```

`commit` sets the interpreter result string equal to the string being built up in the object. `getString` does a commit and then returns the string.

SEE ALSO

CTCLObject(3)

CTCLString

Name

`CTCLString` — Provide a wrapper for the Tcl_DString data type and its API

Synopsis

```
#include <TCLString.h>
...
class CTCLString
{
public:
  CTCLString ();
  CTCLString (const char* pString  ) ;
  CTCLString(const std::string& rString);
  CTCLString(const Tcl_DString& rString);
  CTCLString (const CTCLString& aCTCLString );
  ~ CTCLString ();

  CTCLString& operator= (const CTCLString& aCTCLString);
  int operator== (const CTCLString& aCTCLString);
  int operator!= (const CTCLString& aCTCLString);
  int operator> (const CTCLString& aCTCLString);
  int operator< (const CTCLString& aCTCLString);
  int operator>=(const CTCLString& aCTCLString);
  int operator<=(const CTCLString& aCTCLString);

  Tcl_DString& getString();
  CTCLString& Append (const std::string& rString, Int_t nLength=-1);
  CTCLString& Append (const CTCLString&  rString, Int_t nLength=-1);
  CTCLString& Append (Tcl_DString&       pString, Int_t nLength=-1);
  CTCLString& Append (const char*        pString, Int_t nLength=-1);
  CTCLString& AppendElement (const Tcl_DString*      pRhs);
  CTCLString& AppendElement (const CTCLString&       rRhs);
  CTCLString& AppendElement (const std::string&      rRhs);
  CTCLString& AppendElement (const char*             pRhs);
  CTCLString& AppendElement(DFloat_t value, const char* pFormat = "%f");
  CTCLString& AppendElement(long value, const char* pFormat = "%i");

  CTCLString& StartSublist ()  ;
  CTCLString& EndSublist ()  ;
  UInt_t  Length () const ;
  CTCLString& Truncate (UInt_t nNewLength)  ;
  Bool_t isCommand () const  ;

  Bool_t Match (const char*       pPattern) const;
  Bool_t Match (std::string&      rPattern) const;
  Bool_t Match (const CTCLString& rPattern) const;
  operator const char* () const;
  operator std::string () const;
  operator Tcl_DString* ();
};
```

DESCRIPTION

The Tcl API provides a dynamic string type Tcl_DString. For many purposes, the C++ std::string is sufficient, however the Tcl_DString list building functions are unmatched in std::string. `CTCLString` is an object oriented wrapping of a Tcl_DString

METHODS

```
CTCLString ();
CTCLString (const char* pString) ;
CTCLString(const std::string& rString);
CTCLString(const Tcl_DString& rString);
CTCLString (const CTCLString& aCTCLString );
```

Constructs a `CTCLString` object. With the exception of the first constructor, which produces an empty string, all of these constructors initialize the contents of the underlying `Tcl_DString` with the string representation of their parameter.

```
CTCLString& operator= (const CTCLString& rhs);
```

Supports assignment to a `CTCLString` from another; *`rhs`*.

```
int operator== (const CTCLString& rhs);
int operator!= (const CTCLString& rhs);
int operator> (const CTCLString& rhs);
int operator< (const CTCLString& rhs);
int operator>=(const CTCLString& rhs);
int operator<=(const CTCLString& rhs);
```

Relational operators provide for lexicographic copmarisons between the object and *`rhs`* which is another `CTCLString`.

```
Tcl_DString& getString();
```

Returns a reference tothe underlying Tcl_DString of the object.

```
CTCLString& Append (const std::string& String,
                  Int_t nLength=-1);
CTCLString& Append (const CTCLString& String,
                  Int_t nLength=-1);
CTCLString& Append (Tcl_DString&       String,
                  Int_t nLength=-1);
CTCLString& Append (const char*            String,
                  Int_t nLength=-1);
```

Appends a section of *`String`* to the `CTCLString` that is being built up. The first *`nLength`* characters are appended. if *`nLength`* is `-1` then all *`String`* is appended.

```
CTCLString& AppendElement (const Tcl_DString*  item);
CTCLString& AppendElement (const CTCLString&   item);
CTCLString& AppendElement (const std::string&  item);
CTCLString& AppendElement (const char*         item);
CTCLString& AppendElement(DFloat_t item,
                          const char* pFormat = "%f");
CTCLString& AppendElement(long     item,
                          const char* pFormat = "%i");
```

Appends *`item`* as a list element to the end of the string. If necessary quotation is performed to ensure the item is treated as a single list element. The *`pFormat`* parameter controls the conversion of non string data types to a string and is of the form of any control sequence used by sprintf. For example `"i = %d"` could be used to convert an integer to a label and its value which would be appended to the string as e.g. `{i = 1234}`

```
CTCLString& StartSublist ()  ;
CTCLString& EndSublist ()  ;
```

Used in conjuntion with `AppendElement` these start and end sublists which are list elements that consist of lists. Sublists can be nested to any depth. For example:

```
    CTCLString s;
    s.AppendElement("a");
    s.StartSublist();
    s.AppendElement("b");
    s.AppendElement("c");
    s.StartSublist();
    s.AppendElement("d");
    s.AppendElement("e");
    s.EndSublist();
    s.AppendElement("f");
    s.EndSublist();
    s.AppendElement("g");
```

Would make the `s` contain the string `"a {b c {d e} f} g"`

```
UInt_t  Length() const ;
```

Returns the number of characters in the string.

```
CTCLString& Truncate (UInt_t nNewLength)  ;
```

Truncates the string to the first *`nNewLength`* characters.

```
Bool_t isCommand () const  ;
```

Analyzes the string and returns `kfTRUE` if the string is a ’well formed command’. Note that a well formed command may still have syntax and execution errors. This just ensures that a string has a balanced set of quoting characters.

```
Bool_t Match (const char*           Pattern) const;
Bool_t Match (std::string&      Pattern) const;
Bool_t Match (const CTCLString& Pattern) const;
```

Returns `kfTRUE` if the contents of the string matches the *`Pattern`* parameter. The *`Pattern`* parameter can contain all of the wildcards in *glob* style pattern matching. See REFERENCES below for moer information about glob style matching.

```
operator const char* () const;
operator std::string () const;
operator Tcl_DString* ();
```

These operators are implicit and explicit type conversion operators that allow a `CTCLString` object to be treated as a char* pointing to a null terminated string, a std::string object, or a Tcl_DString pointer.

SEE ALSO

Tcl_DStringAppend(3tcl), Tcl_DStringAppendElement(3tcl), Tcl_DStringEndSublist(3tcl), Tcl_DStringFree(3tcl), Tcl_DStringGetResult(3tcl), Tcl_DStringInit(3tcl), Tcl_DStringLength(3tcl), Tcl_DStringResult(3tcl), Tcl_DStringSetLength(3tcl), Tcl_DStringStartSublist(3tcl)

REFERENCES

```
J.K. Ousterhout Tcl and the Tk Toolkit
Addison-Wesley Professional Computing Series 1994 see section 9.2
```

CTCLTimer

Name

`CTCLTimer` — Abstract base class for C++ objects attached to timer events.

Synopsis

```
#include <TCLTimer.h>
...
class CTCLTimer  : public CTCLInterpreterObject
{
public:
  CTCLTimer ();
  CTCLTimer(CTCLInterpreter* pInterp, UInt_t nMsec = 0);
  virtual ~CTCLTimer ( );


  Tk_TimerToken getToken() const;
  UInt_t getMsec() const;
  Bool_t IsSet() const;

  virtual   void operator() ()   = 0;

  void Set ()  ;
  void Set(UInt_t nms);
  void Clear ()  ;
};
```

DESCRIPTION

Tcl/Tk provide a mechanism for scheduling functions to be executed after a time delay specified in milliseconds. The `CTCLTimer` class is an abstract base class that provides an interface into the API for that facility. To use `CTCLTimer` you must create a class derived from `CTCLTimer` that overrides and implement the `operator()` function. Create an object from the resulting function class. Use the object's `Set` and `Clear` members to schedule or cancel a scheduled execution. The code fragment example below shows how to do this to create a class that periodically emits the text "Tick" to stderr. Many #include directives are missing for brevity.

```
// Interface to Ticker normally goes in a header.
class Ticker : public CTCLTimer
{
public:
    Ticker(CTCLInterpreter* pInterp, int seconds);
    virtual ~Ticker();

    virtual void operator()();
};
...
// Implementation of Ticker normally goes in a .cpp

// Constructor of Ticker:

Ticker::Ticker(int seconds) :
    CTCLTimer(pInterp, seconds*1000)
{
    Set();                   // Schedule first one.
}
// Destructor.. chain to base class.
Ticker::~Ticker() {}

// called when timer goes off:
void
Ticker::operator()() {
    cerr << "Tick\n";
    Set();                 // Schedule next one.
}
...


Ticker Tick(pInterp, 1);  // Tick every second.
```

METHODS

```
CTCLTimer ();
CTCLTimer(CTCLInterpreter* pInterp,
          UInt_t nMsec = 0);
```

Construct timer objects. The first form of the constructor creates a timer object that must be later bound into an interpreter via a call to `CTCLInterpreterObject::Bind`. The seconf form of the contructor creates a timer object that is already bound to *`pInterp`* and has an initial schedule delay of *`nMsec`*.

```
  Tk_TimerToken getToken() const;
  UInt_t getMsec() const;
```

These two members access internal state of the object. `getToken` returns the Tk_TimerToken associated with the timer object. This is the Tcl/Tk token that identifies the timer request to the interpreter. `getMsec` retrieves the current value of the delay parameter in milliseconds.

```
virtual   void operator() ()   = 0;
```

This function must be overidden and implemented in concrete timer classes. See the example in DESCRIPTION above.

```
void Set ()  ;
void Set(UInt_t nms);
Bool_t IsSet() const;
```

`Set` schedules the object for execution. If *`nms`* is provided it is saved as the scheduling parameter and determines the delay in milliseconds before `operator()` is next called. If not provided, the most recently used delay will be used again.

`IsSet` returns `kfTRUE` if the timer is currently pending, or `kfFALSE` if no pending timer request is active.

```
void Clear ()  ;
```

If a Timer request is pending, cancels it. If no timer request is pending, this function does nothing, and does not report an error.

SEE ALSO

CTCLInterpreterObject(3)

REFERENCES

```
Musser, Derge, Saini: STL Tutorial and Reference Guide
Addison-Wesley Professional Computing Series; 2001 ISBN 0-201-37923-6
See section 2.4 for a description and discussion of function objects.
```

Thread

Name

`Thread` — Abstract base class for thread objects.

Synopsis

```
#include <Thread.h>


 class Thread {
  Thread()();
  Thread(std::string name);
  virtual ~Thread();
  int detach();
  unsigned long getId();
  void start();
  void setName(std::string name);
  void join();
  const std::string getName();
  virtual = 0 void run();
}
```

Description

`Thread` is an abstract base class that can be used to create threads. To make a thread you will normally derive a class from `Thread` and write the `run` method to implement the thread's code.

A thread can then be created like any other object. The `start` method schedules the thread's `run` member for execution

Threads can block on the completion of a thread via the `join` method.

Public member functions

Constructors. Constructors create a thread. The thread is not scheduled for execution until the `start` method is called. Threads can have an optional thread name.

`Thread()`(void);

Constructs an anonymous thread.

`Thread`std::string*`name`*

Constructs a named thread.

Thread identification. Threads are identified by an optional name, which need not be unique, and a unique identifier (thread id), that is assigned by the underlying operating systemn.

```
unsigned long getId();
```

Returns the thread id. If the thread has not yet been started, `-1` is returned regardless of the thread. A thread only gets an id when it has been started.

```
void start();
```

Starts a thread. A thread id is allocated and the `run` method is scheduled for execution.

```
void setName(std::string name);
```

Modifies the name of the thread.

```
int detach();
```

Detaches an executing thread. If the thread has not started, this returns `-1`. Detaching a thread indicates that any operating system storage (not object storage) associated with a thread can be released when the thread exits. If the thread does not detach, it is necessary to `join` the thread to fully release its storage.

On success, `0` should be returned, otherwise a value that is one of the values in `errno.h` is returned to describe why the call failed.

```
  void join();
```

`join` blocks until the thread exits. When the thread does exit, thread specific storage is reclaimed. Note that if a thread has calledis `detach` it is not clear clear wht clear what this member will do.

```
  const std::string getName();
  virtual = 0 void run();
```

You must implement this function to provide a concrete `Thread` run-time behavior.

Synchronizable

Name

`Synchronizable` — Wait queue for threads

Synopsis

```
#include <Synchronizable.h>



 class Synchronizable {
  Synchronizable();
  virtual ~Synchronizable()();
  void waitFor();
  void waitFor(long timeout);
  void waitFor(long seconds, int nanoseconds);
  void notify();
  void notifyAll();
}
```

Description

This class implements a wait queue for threads. A thread can place itself in a wait queue and then be awakened by another thread, or by a timeout on the wait itself.

Public member functions

`Synchronizable(void);`

Creates a thread wait queue.

```
virtual ~Synchronizable()();
```

Destroys a thread wait queue.

```
void waitFor();
```

Waits for an unbounded length of time. The method completes when either the calling thread is at the head of the wait queue and another thread invokes the `notify`, member or another thread invokes the `notifyAll` member function.

```
void waitFor(long timeout);
```

Waits as above, but for at most *`timeout`* seconds. In the current implementation, it is not possible to distinguish between a wait that times out and one that completes due to notification.

```
void waitFor(long seconds, int nanoseconds);
```

Waits as above, but for *`seconds`* seconds and at least an additional *`nanoseconds`* nanoseconds, or until notification awakens the thread.

```
void notify();
```

Wakes up the thread that least recently entered one of the wait functions and has not yet timed out.

```
void notifyAll();
```

Awakens all of the threads that are currently blocked on this object.

SyncGuard

Name

`SyncGuard` — Provide Critical Regions, Monitors

Synopsis

```
#include <SyncGuard.h>
```

```
 class SyncGuard {
  SyncGuard()();
  SyncGuard(Synchronizable& syncer);
  SyncGuard(Synchronizable& syncer, bool tryonly);
  virtual ~SyncGuard();
  dshwrapthread_t getOwner();
}

Macros:
#define sync_self
#define sync_begin(s)
#define sync_begin2(t,s)
#define sync_end


#define sync_trybegin(s)
#define sync_trybegin2(t,s)
#define sync_tryend

#define sync_global_begin(t)
#define sync_global_end
```

Description

The `SyncGuard` class uses a `Synchronizable` object to create a critical region, or monitor. Both of these synchronization primitives are mechanisms that ensure that only one thread can execute a guarded code segment at a time.

Creating a `SynchGuard` on a synchronizable object locks the guard. Destroying the object releases the guard. You can create more than one `SyncGuard` on a single `Synchronizable` object, and typically will.

The simplest use of `SynchGuard` is via its macros.

The NSCLDAQ threading system creates a global `Synchronizable` object that is used by all of the locking macros and constructors that don't take an explicit parameter.

Public member functions

`SyncGuard()`

Constructs a synchronization guard that uses the global `Syncrhonizable` to perform the synchronization.

`SyncGuard`Synchronizable&*`syncer`*

Creates a `SyncGuard` that uses *`syncer`* as its synchronization object.

`SyncGuard`Synchronizable&*`syncer`*bool*`tryonly`*

Same as above, however if *`tryonly`* is true, and *`syncer`* is already locked, the thread continues execution.

```
virtual ~SyncGuard();
```

Destroys the guard, releasing the synchronization object.

```
dshwrapthread_t getOwner();
```

Returns the id of the thread that owns the synchronization object used by a synchronization guard. One use of this is to do a tryonly creation and then check to see if the running thread owns the object (indicating the try succeeded).

Macros. The macros below simplify the creation of critical segments of code.

`void` **`sync_self`**`(void);`

This macro is intended for use within an object that is derived from a `Synchronizable`. It locks the current object.

`void` **`sync_begin`**`(` *`Synchronizable& s`* `);`

Starts a a critical region that uses *`s`* to synchronize access.

`void`**`sync_begin2`**`(` *`t`* `,` *`Synchronizable& s`* `);`

Same as `sync_begin` but *`t`* is appended to the name of the temporary `Syncrhonizable` the macro creates.

`void` **`sync_end`**`(void);`

Marks the end of a critical section that was begun with one of the sync macros above.

`void` **`sync_trybegin`**`(`*`Synchronizable& s`*`);`

This macro creates a temporary `SynchGuard` object constructed on *`s`*. Non-blocking access is to the guard is attempted. If successful, the code that follows the macro up until the next `sync_tryend` macro invocation is executed. If access to the guard could not be gotten without blocking, the code will not be executed.

`void` **`sync_trybegin2`**`(` *`t`* `,` *`Synchronizable& s`* `);`

This macro behaves the same as `sync_trybegin2`, however the parameter *`t`* is used to construct the name of the `SyncGuard` object used by the macro.

`void` **`sync_tryend`**`(void);`

Marks the end of a block of code that was synchronized using the `sync_trybegin*` macros.

`void` **`sync_global_begin`**`(` *`t`* `);`

The NSCLDAQ thread synchronization library creates a global `Synchronizable` object. This can do process wide synchronization on a coarse grained level. `sync_global_begin` constructs a temporary `SyncGuard` on that object and locks it. The *`t`* paramter is used to construct the name of the guard object.

`void sync_global_end(void);`

Destroys the `SyncGuard` created by lexically most recent `sync_global_begin` macro. This releases the synchronization object for other threads.

URL

Name

URL — Parse Uniform Resource Identifiers (URI)

Synopsis

```
#include <URL.h>



 class URL {
  URL(std::string uri);
  URL(const URL& rhs);
  URL& operator=(const URL& url);
   const bool operator==(const URL& rhs);
   const bool operator!=(const URL& rhs);
  const std::string getProto();
  const std::string getHostName();
  int getPort();
  const std::string getPath();
  const std::string operator std::string();
}
```

Description

Objects of this class parse Universal Resource Identifiers (URIs). Methods of this class return the components of a URI.

Public member functions

Constructors.

`URL`std::string*`uri`*

Constructs a `URL` object by parsing *`uri`*. The constructor may throw a `CURIFormatException` exception. see Exceptions below for more information.

`URL`constURL&*`rhs`*

Constructs a `URL` object that is an exact duplicate of the *`rhs`* object.

Other canonical functions. The `URL` class implements assignment, and comparison for equality and inequality. Equality holds if all the components of the parsed URI are identical. Inequality holds if equality does not hold.

Other methods.

```
const std::string getProto();
```

Returns the protocol component of the URI. The protocol component describes the mechanism used to access the resource.

```
const std::string getHostName();
```

Returns the hostname component of the URI. The hostname describes where in the network the resource described by the URI is located.

```
int getPort();
```

Returns the port number part of the URI. While port numbers are optional on real URI’s they are not optional for NSCL URIs. The port determines where the server for the resource is listening for connections.

```
const std::string getPath();
```

Returns the path component of the URI. The path component tells the client and server where within the namespaces for the protocol the component is located. The path component is optional. If not provided, it defaults to `/`.

```
const std::string operator std::string();
```

Re-constructs and returns the stringified URL. This should be very close to the string that was used to construct this object, or the object from which the object was copied.

Exceptions

Not all strings are valid URIs. If a `URL` object is constructed with a string that is not a valid URI, the constructor will throw a `CURIFormatException`. `CURIFormatException` is derived from the `CExeption` common exception base class.

The NSCL Exception class library chapter describes the exception class hierarchy, how to use it and its common set of interfaces. The `CException` reference page describes the `CException` class. The `CURIFormatException` reference page describes the `CUIRFormatException` class.

CPortManager

Name

`CPortManager` — Provide a C++ interface to the server port manager daemon.

Synopsis

```
#include <config.h>
#include <CPortManager.h>
#include <CPortManagerException.h>



 class CPortManager {
  CPortManager(std::string host = string("localhost"));
  CPortManager(std::string host, int Port);
  int allocatePort(std::string application);
  std::vector<portInfo> getPortUsage();
}
```

Description

`CPortManager` provides a C++ application programming interface into the NSCL port manager server. This class serves as a proxy for communication with port management servers. Once you have created a `CPortManager` object, you can use it to allocate ports (if the server is local), and to list port usage of any server on the network.

Note that members of this class can throw exceptions of type `CPortManagerException`. See the CPortManagerException(3) for more informationa bout that.

To include the headers required you, or the Makefile skeleton you are using must have added the `include` subdirectory ofthe NSCLDAQ installation directory tree to the include paths for the compile e.g.: `-I/usr/opt/daq/current/include`.

To link you must include the NSCL DAQ lib subdirectory in the library search path, link in the library and ensure that the shared library can be found at load time. For example:

```
-L/usr/opt/daq/current/lib \
-lPortManager \
-Wl,"-rpath=/usr/opt/daq/current/lib"
```

Public member functions

```
  CPortManager(std::string host = string("localhost"));
```

Constructs a port manager interface object. The object connects to the port manager running on the host *`host`*, using the default port (`30000`) to form the connection. If the *`host`* parameter is omitted, it defaults to `localhost`

```
CPortManager(std::string host, int Port);
```

Constructs a port manager interfaceobject. The object connects to the host manager running on `host` via the connection port `Port`.

```
int allocatePort(std::string application);
```

Allocates a port from the port manager. On success the return value is the port number allocated in local machine byte order. Note that when interacting with the network functions, in general, this will have to be converted to network byte ordering.

Once the port is allocated, it remains allocated until the program exits. This is because a connection will be held with the port manager and cannot be closed by the user.

If a port could not be allocated (e.g. all are in use or the port manager could not be contacted, a `CPortManagerException` described in "Types and public data" below and completely in CPortManagerException.

```
std::vector<portInfo> getPortUsage();
```

Returns a vector of portInfo structs that describes the current port allocations of the port allocator. The most common use case for this is to locate a specific service offered by a remote port manager.

If the interaction with the server failed, a `CPortManagerException` described in "Types and public data" below and completely in CPortManagerException.

Types and public data

The set of port allocations is described by an STL vector of portInfo structs. The `std::vector` is described in any C++ reference that contains a description of the Standard Template Library (STL).

portInfo structs have the following fields:

Type: int
Name: `s_Port`
Description: The port number that this object describes. The port number is provided in the byte order of the local system. Note that when used to provide a port number to Unix/Linux network functions in general this number will have to be converted to network byte order.

Type: std::string
Name: `s_Application`

Description: The name of the application that requested this port. The application name represents some service that is being offered. Usually (but not always), the application name and the `s_User` fields together are unique system wide.

Type: std::string
Name: `s_User`
Description: The name of the user that was running the application that reserved this port. In the event that several users have run the same application on a single system, this identifies which user ran the application that requested this port.

`CPortManagerException` objects are thrown in most cases the library encounters an exception.

Exceptions

The `CPortManagerException` is thrown to inform the program of most errors dected by this API. For more information about this exception, see: CPortManagerException(3).

EXAMPLES

Allocates a port to MyApplication

Example 1. Allocating a port with the port manager

```
...
CPortManager pm;
int port;
try {
 port = pm.allocatePort("MyApplication");
 ListenOnPort(port);                      // Not shown for brevity
}
catch (CPortManagerException& error) {
 cerr << "Could not allocate a port: " << error << endl;
 exit(-1);
}
...
```

Lists to cout the port allocations on the system: ahost.nscl.msuedu

Example 2. Listing the port allocatiosn on a system.

```
     ...
   CPortManager pm("ahost.nscl.msu.edu");
   vector<CPortManager::portInfo> info = pm.getPortUsage();
   for(int i =0; i < info.size(); i++) {
      cout << "Port " << info.s_Port
           << " allocated to " << info.s_Application
           << " run by " << info.s_User << endl;
   }
    ...
```

SEE ALSO

portAllocator(3tcl), CPortManagerException(3daq), DaqPortManager(1tcl)

CPortManagerException

Name

`CPortManagerException` — Report errors conditions in port manager transactions

Synopsis

```
#include <config.h<
#include <histotypes.h<
#include <CPortManagerException.h<




 class CPortManagerException {
  CPortManagerException(std::string host, Reason why, std::string doing);
  virtual const const char* ReasonText();
  virtual const Int_t ReasonCode();
  static string ReasonCodeToText(int code);
}
```

`ostream&` **`operator<<`** `( ostream&` *`f`* `,  const CPortManagerException&` *`e`* `);`

Description

`CPortManagerException` is an exception that can be thrown by the port manager class `CPortManager` see the man page for that class). Since it is derived from `CException`, it can be caught as a generic error. In addition to the base class members, the class supports insertion into an output stream.

Public member functions

```
  CPortManagerException(std::string host, Reason why, std::string doing);
```

Constructs the exception. *`host`* is the host that the object was connected to, or attempted to connect to when the error was detected. *`why`*Is the reason for the exception. See "Types and public data" below for more information about the possible values the Reason type can take. *`doing`* provides context information that describes what the object was attempting to do when the error was detected.

```
  virtual const const char* ReasonText();
```

Returns comprehensive human readable text that describes the reason this operation failed.

```
  virtual const Int_t ReasonCode();
```

Returns the reason code. The reason code is just the Reason cast to an integer.

```
  static string ReasonCodeToText(int code);
```

Converts a reason code into a text string that describes the reason.

```
ostream& operator<< ( ostream& f ,  const CPortManagerException& e );
```

Formats the exception object *`e`* and writes it to the output stream *`f`*. A reference to the output stream is returned allowing the normal sorts of cascading of the `operator<<` function.

Types and public data

Reason is an enumerated type that describes the actual error condition that was detected:

NoPorts

When attempting to allocate a service port, thee daemon reported that all available service ports in the block it is managing are in use.

NotLocal

> You are attempting to allocate a port on a remote system. Service ports can only be allocated by programs running on the same system as the port manager daemon.

ConnectionFailed

> An attempt to connect to the daemon failed.

EXAMPLES

The example below catches an exception. If the exception was thrown because of a Connection failure. it is just printed to cerr. Otherwise, The reason text is printed along with a message idicating that a port allocation failed.

Example 1. Catching a CPortManagerException

```
...
try {
...
}
catch (CPortManagerException& e) {
  int why = e.ReasonCode();
  if((CPortManagerException::Reason)why ==
                  CPortManagerException::ConnectionFailed) {
     cerr << e << endl;
  }
  else {
     cerr << "Port Allocation Failed: " << e.ReasonText() << endl;
  }
}

...
```

SEE ALSO

portAllocator(3tcl), CPortManager(3daq), DaqPortManager(1tcl) CException(3tcl)

V. 3tcl

portAllocator

Name

`portAllocator` — Tcl API for the DaqPortManager daemon.

Synopsis

package require portAllocator

::portAllocator create name ***`?-hostname host? ?-port port`***

`::name` **listPorts**

`::name` **allocatePort** ***`application`***

`::name` **destroy**

DESCRIPTION

The **portAllocator** package provides access to the NSCL TCP/IP port management server. It can be used by TCP/IP server applications t obtain a server listen port. The package follows an object oriented model. The application creates a `portAllocator` object, which stands as a proxy between the application and a port manager server. Using this object the application can allocated and deallocate ports, as well as request port allocation information.

Once your application has finished interacting with a `portAllocator` object, it can destroy it. If the application must hold a connection to the server in order to maintain one or more allocated ports, destruction of the requesting object will not result in that connection being closed, ensuring that the port will remain allocated.

COMMANDS

::portAllocator create *`name ?-hostname host? ?-port port`*

Creates a port allocator with the specified name. The optional `-hostname` option allows you to specify with which host you want the allocator to communicate. If not supplied, this defaults to localhost. The optional `-port` switch allows you to specify a port on which to connect. If not specified,the allocator will first attempt to read the port number from the file `/var/tmp/daqportmgr/listen.port` before falling back to port number 30000.

The command returns the fully qualified name of the allocator. This name can be stored in a variable for later use (see EXAMPLES).

***::name* listPorts**

Returns a TCL formatted list that describes the ports that are cur? rently allocated by the server. Each element of the list is a three element sublist containing in order, the allocated port, the name of the application holding the port, and the name of the user that is run? ning the application.

::name* allocatePort *application

Attempts to allocate a port from the server. Note that the protocol only allows you to allocate ports from a server running on localhost. application is the name of the application under which you would like to register port ownership.

***::name* destroy**

Destroys a portAllocator. If the portAllocator is holding open a connection to the server because the application has allocated a port, this connection will remain open.

EXAMPLES

The example below allocates a port from the localhost, and starts listening for connections.

Example 1. Allocating a service port in Tcl

```
package require portAllocator
set p [::portAllocator create local]
set port [$p allocatePort]
socket -server handleConnections $port
```

The example below requests that the host somehost.nscl.msu.edu return a list of the ports in use. The port usage is then printed at stdout:

Example 2. Listing allocated ports in Tcl

```
package require portAllocator
set p [::portAllocator create remote -hostname somehost.nscl.msu.edu]
set usage [$p listPorts]
foreach allocation $usage {
  set port        [lindex $allocation 0]
  set application [lindex $allocation 1]
  set user        [lindex $allocation 2]
  puts "Port $port allocated to $application run by $user"
}
```

SEE ALSO

CPortManager(3daq), CPortManagerException(3daq), DaqPortManager(1tcl)

DvdBurner

Name

`DvdBurner package` — Burn NSCL Data to DVD

Synopsis

package require DvdBurner

DvdBurner::CreateDvds ?range?

DvdBurner::BurnDVD iso

DESCRIPTION

The DvdBurner package provides the capability to burn an arbitary ISO image or set of NSCL data acquisition systsem event data and associated data to DVD.

DvdBurner::CreateDvds ?range?

creates DVDs from a set of recorded data taking runs. The data and run meta-data are recorded to DVD. If necessary, several DVDs are created. Data are split so that all of the data associated with a run is on a single DVD (not split across more than one DVD. See, however "Dependencies and Restrictions" below.

DvdBurner::BurnDVD isofile

Burns the pre-built ISO image in the file *`isofile`* to DVD.

Dependencies and Restrictions

- growisofs must be isntalled and in the user’s path.

- The DVD must either be named `/dev/cdrom` or the environment variable `DVDROM` must be defined to be the name of the DVD burner to use.
- The variable DvdBurner::DVD holds the path to the DVD burner device and can be modified at run time any time after the package has been **require**ed.
- The DVD device must be writable by growisofs.
- The variable DvdBurner::DVDSize holds the number of megabytes a DVD can hold It can be modified from its default value of `4000` if necessary.
- The `DvdBurner::CreateDvds` command will fail if there are runs that won't fit on single DVDS. Errors in the NSCL file structure can also cause failures.
- The variable `DvdBurner::ISOFSroot` holds the directory in which the run data will be marshalled and the ISO images built. The initial value is `/scratch/$tcl_platform(user)/isos` where `tcl_platform(user)` is your username.

sequencer

Name

`sequencer` — Provide a ReadoutGui plugin for nscldaq 8.1 and later that can automate several data taking runs.

Synopsis

package require runSequencer

DESCRIPTION

The **runSequencer** package provides support for automating data taking runs of fixed duration. **runSequencer** is a ReadoutGui plugin that is compatible with nscldaq-8.1 and later.

To operate, you must use the ReadoutGUI from nscldaq-8.1 or later. At the NSCL, you can do this by using **/usr/opt/daq/8.1/bin/ReadoutShell**. See USING for more information on how to use this plugin.

For each run you can define a set of run parameters that are set through custom actions just prior to the start of the run. See CUSTOMIZING below for more information about how to setup these actions and how to specify the parameters.

Run sequences are called *Run plans*. Run plans are files that can be loaded into the sequencer, edited, saved, and executed. A run plan provides values for all of the parameters for each run in the sequence.

USING

This section desribes using in two senses of the word: Incorporating the sequencerGui into ReadoutGUI, and operating the sequencer once it is installed. Setting up the parameter definitions and actions used by the sequencer are described in CUSTOMIZING below

Incorporating SequencerGui into ReadoutGUI

To incorporate the sequencer into ReadoutGUI you must use a `ReadoutCallouts.tcl` file to extend ReadoutGui. This file must contain the line:

```
package require runSequencer
```

Furthermore you must run a Readout GUI from nscldaq version 8.1 or later. At the NSCL, you can run the 8.1 version of the ReadoutGUI as follows: **/usr/opt/daq/8.1/bin/ReadoutShell** Note that when you run the Readout GUI, the `sequencer.config` and `sequencerActions.tcl` files are expected to exist. See CUSTOMIZING below for information about the contents of these files. see "FILES and ENVIRONMENT" for where they belong.

Operating the sequencer GUI

The sequencer creates a second window associated with the Readout GUI. This window consists mainly of a table with column labels defined by the `sequencer.config` file. Each row of the table represents a run in the sequence of runs in a *run plan*

You can edit the table to provide values for each of the parameters (columns) for each step in the run sequence. Navigate in the table by using the arrow keys, the mouse or the tab and shift tab keys. Once you are happy with your run plan you can save it using the File⟶Save... menu command.

The File⟶Open... menu command allows you to read in a plan from file. This plan can be edited, and saved. File⟶Clear... clears the table.

To execute a run plan, use the ReadoutGUIPanel to set the length of each run. (The sequencer will turn on timed runs when you start the sequence). Then click the Execute Plan button to start the sequence. The button is relabeled Abort Plan while the plan is running. Clicking it while the plan is running ends the active run and stops the sequence. If event recording was enabled for an aborted run plan you are given the option to discard all the event data that was taken for the sequence.

CUSTOMIZING

The sequencer is completely customizable. The label for each column and actions to take to set each column are defined by the `sequencer.config` file. This file is a text file. Each line in the file describes a column. Blank lines are ignored as are lines for which the first non-blank character is a hash (#) character. Fields in a line are separated by whitespace. Whitespace can be incorporated into a field either by quoting the field with double quotes (") or curly brackets (for example {this has spaces}).

Each column definition line of `sequencer.config` has fields in the following order.

Column Name

> The text that labels the column in the run plan.

set action

> A tcl command that is executed when it is time to change the value of the item in the column. The column name and the value of that cell of the table are appended to the command. If the set action is blank, no action is taken. The action function must return 0 on success or else the run plan will be aborted prior to starting the run for the failing step. Note that if an action can return and error it should indicate to the user what the error was.

initialize action

> A tcl command that is executed when the sequencer is set up, before executing any run plan. This is intended to be used for any one-time intialization required to access the item controlled by the table. The column name is appended to this command when it is called.

See EXAMPLES for examples of the sequencer.config file. See "FILES and ENVIRONMENT" for information about where this file belongs.

The actions defined in the `sequencer.config` file must come from somewhere. During initialization, the sequencer sources the file `sequencerActions.tcl` this file is a Tcl script that can include procedure definitions, data definitions, etc. Normally it provides the tcl commands that are specified as actions in the `sequencer.config` file. For a sample `sequencerActions.tcl` file see EXAMPLES. For information about how the sequender finds this file, see "FILES and ENVIRONMENT"

FILES and ENVIRONMENT

The sequencer must locate a column configuration file and a script that defines the actions used by that file. By default, these files are:

`./sequencer.config`

> The column configuration file. By default this is located in the current working directory of the sequencer when it is started.

`./sequencerActions.tcl`

A script that defines actions used by the column configuration file. By default, this is sourced from the current working directory of the sequencer when it is started.

Environment variables can alter the location and names of these files:

SEQCONFIGDIR

If defined, this environment variable is the directory from which the two configuration files are loaded for example: **export SEQCONFIGDIR=~/config**

SEQCONFIGFILE

If defined, this environment variable is the name of the file used to configure the columns. If, for example, you: **export SEQCONFIGDIR=~/config** and **export SEQCONFIGFILE=columns.def** The column configuration file loaded will be `~/config/columns.def`.

SEQACTIONFILE

If defined, this environment variable is the name of the action script file that defines the actions referenced in the column configurationfile. If, you have **export SEQCONFIGDIR=~/config** and **export SEQACTIONFILE=actions.tcl**, The action script file will be sourced from `~/config/actions.tcl`.

EXAMPLES

Example 1. Action script example

This example shows part of an action script that defines the actions `epicsAccess` and `epicsSet` that provide actions to access epics channels.

```
package require epics
proc epicsSet {name value} {
  set status [catch  {$name set $value}] msg
  if {$status != 0} {
     tk_msgBox -message $msg -icon error -title {Epics channel set error}
  }
  return $status

}
proc epicsAccess name {
    epicschannel $name
}
```

Example 2. Sequencer column configuration file

The sequencer file below defines a set of columns that are all epics channels:

```
# Column          set_action  init_action

P222ET_TARGET    epicsSet    epicsAccess
P222ER           epicsSet    epicsAccess
FLTCHAN73        epicsSet    epicsAccess
P#5:406353       epicsSet    epicsAccess
```

Not the use of blank lines and comment lines.

The directory `examples/sequencer` under the installation directory of the nscl daq includes several sample files.