LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY
- LIGO -
CALIFORNIA INSTITUTE OF TECHNOLOGY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Document Type **LIGO-T980094-03 - E** 11/17/1998

The Generic API's baseline specifications

James Kent Blackburn
Philip Ehrens, David Farnham

Distribution of this document:

LIGO LDAS Group

This is an internal working document
of the LIGO Project.

California Institute of Technology
LIGO Project - MS 51-33
Pasadena CA 91125
Phone (818) 395-2129
Fax (818) 304-9834
E-mail: info@ligo.caltech.edu

Massachusetts Institute of Technology
LIGO Project - MS 20B-145
Cambridge, MA 01239
Phone (617) 253-4824
Fax (617) 253-7014
E-mail: info@ligo.mit.edu

WWW: http://www.ligo.caltech.edu/

The Generic API's
baseline specification

James Kent Blackburn
Philip Ehrens, David Farnham

California Institute of Technology
LIGO Data Analysis Group
November 17, 1998

I. Introduction

A. General Description:

1. The genericAPI provides the base set of functionality in the form of an interpreted command language that exist in all LIGO Data Analysis System (LDAS) distributed computing API components.
   a) The interpreted command language to be used is TCL/TK, which provides a command line, scripting and a graphical interface.
   b) The TCL/TK commands are extended to support low level system interfaces and greater computational performance using C++ code that utilizes the standard TCL/TK C code API library in the form of a TCL/TK package.
2. The genericAPI TCL/TK script accesses a genericAPI.rsc file containing needed information and resources to extend the command set of the TCL/TK language using the genericAPI package, which exists in the form of a shared object.
3. The genericAPI will provide setup and configuration for all socket and file based communications used in all other LDAS distributed computing API components.

B. The genericAPI.tcl Script's Specification:

1. The following is a list of TCL/TK procedures (**proc**'s) which are implemented in the genericAPI.tcl script.
   a) These commands are used as part of the genericAPI's on-line help facility, but may be used in other contexts as needed.
      (1) **renderHTML:** Parses and renders *HTML* content from a variety of possible sources which are identified by the content of the first parameter. It will accept a filename, a URL, or arbitrary HTML. The logging system (*described later*) uses this method to produce reports on demand. Extensions allow arbitrary data to be embedded in *HTML* content which can be handled by call-backs associated with tag names and attributes.
      Usage: **renderHTML $var .text**
      where **$var** is a TCL/TK variable containing a filename, a URL, or

HTML text and **.text** is the name of a text widget and. The **$var** is required and **.text** will be initialized if necessary
Return object: TCL/TK exception on error.

(2) **showHelp:** Compound command which collects and indexes LDAS help files and generates an *HTML* text widget with appropriate functionality for browsing the dynamic hyper-text help documentation. All files in a resource defined directory hierarchy which match a regular expression search for file names are included automatically, making the help system more flexible and extensible. The *HTML 2.0* subset is the base language for the help system.
Usage: **showHelp .text {topic} {path}**
where **.text** is the name of an LDAS *HTML* text widget and is a required parameter. Both **topic** and **path** are optional. The **topic** parameter has a default value of "help" and the default **path** is given in the genericAPI's resource file.
Return object: TCL/TK exception on complete absence of all help files or if **.text** is an incompatible widget.

b) These commands are used as part of the genericAPI's message logging facility but may be used in other contexts as needed.

(1) **openLog:** This is an internal function used by other log functions to open a log file for appending. It is not typically called directly by the user. If the file does not exist a non-interupting dialog will appear which will not go away until you either ask it to open an existing log or request that a new log file be created; looping back to the dialog each time an illegal log file is used. All entries in all log files are time-stamped and become available for automatic report generation.
Usage: **set fileid [openLog logname]**
where **fileid** is the file ID returned from the command, logname is the name of the log file. The default location for the log files are given in the genericAPI's resource file.
Return object: The normal return value is the file ID of the opened file. A TCL/TK exception will be when

(a) file already open,

(b) or group permissions restrict log file creation.

(2) **closeLog:** Closes a log file that has been opened with **openLog**.
Usage: **closeLog logname**
where **logname** is the name of the log file to which the entry is to be added.
Return object: TCL/TK exception when

(a) file is not open,

(b) or group permissions restrict log file closure.

(3) **addLogEntry:** Adds a log entry to a specified log automatically adding a time-stamp and user ID. Added log entries may consist of any text. However, only those entries which are tagged properly can become part of generated reports. Proper tags must be *HTML* or internal LDAS light-weight format. This function calls **openLog** if the log file is not already open for adding entries, then add the entry and finally closes the log file with **closeLog**. If the log file is already open, then this command leaves it open after adding the entry.
Usage: **addLogEntry logname message {tag}**
where **message** is the string to be added to the log file, and **logname** is the name of the log file, **tag** is an *HTML* or *Internal LDAS Light-Weight Format* tag (*defaults to <message>*).
Return object: TCL/TK exception when

(a) an attempt is made to write to an illegal log file,

(b) group permissions restrict writing,

(c) or log file does not exist.

(4) **watchLogs:** Opens a list of log files for *hot* reading of log data. Repeated calls to **watchLogs** at intervals results in *real-time* monitoring of log data. Logs can be dynamically added or removed from the active list. The files will be opened and closed as necessary.
Usage: **set data [watchLogs .text $logs]**
where **.text** is the name of an text widget or an un-initialized variable which is used internally by **watchLogs** and **$logs** is a TCL/TK list of log files to be watched.
Return object: Returns a fresh set of log entries after initialization in HTML format. TCL/TK exception for

(a) group permission restrictions on log file(s).

(5) **queryLogs:** Generates a Log content reports. Report contents are determined by optional start and end times or menu selection, and organized hierarchically. Times can be specified in local time, *GPS* time and time past (*last minute, hour, etc.*).
Usage: **set data [queryLogs .text {$start} {$end} {$logs}]**
where **$start** is a TCL/TK variable storing the start time and **$end** is a TCL/TK variable storing the end time in one of the allowed time formats given above and **$logs** is an optional TCL/TK list of log files to be watched which defaults to all available logs.
Return object: Returns a log report as HTML format. TCL/TK exception for

(a) illegally named **.text** widget,

(b) wrong widget type when using pre-existing widget,

(c) or group permission restrictions on log file(s).

(6) **ArchiveLog:** Closes current log file if open then moves the log file to a standard archive directory (*possibly NFS mounted*). The next call to **addLogEntry** will automatically create a new log file for the API. This command is meant to prevent log files from becoming manageably large.
Usage: **archiveLog {$log}**
where **$log** is an optional TCL/TK variable storing the name of log files to be archived and defaults to the API's log file. The path to the archive directory is specified in the resource file.
Return object: TCL/TK exception for

(a) log files specified that do not exist,

(b) directory path that does not exist

(c) or group permission restrictions on log files.

c) These commands are used as part of the genericAPI's socket command communications facility but may be used in other contexts as needed.

(1) **initSock:** Initializes a TCL/TK level socket connection in a TCL/TK interpreter on either a local or remote machine for communicating commands for services provided by individual API's. The genericAPI's resource file will provide aliases for common ports and services.
This command also initializes a TCL/TK associated array element of the form *${hostname}($port)* which contains the local socket ID. The resulting arrays are globally available and together represent a table of all sockets on all machines which are in use by the LDAS system.
Usage: **set services [initSock hostname port]**
where **hostname** is the remote (*or local is service is local*) host machine's name or IP number and **port** is the alias for a port defined in the genericAPI's resource file associated with the service.
Return object: Returns the socket ID for the services. TCL/TK exception

(a) on failure to connect.

(2) **closeSock:** Closes a socket initialized with initSock. The associated entry from the array of the form ${hostname}($port) containing the socket ID is removed.
Usage: **closeSock hostname port**
where **hostname** is the host machine's name or IP number and **port** is the alias for a port defined in the genericAPI's resource file asso-

ciated with the service.
Return object: TCL/TK exception

(a) if socket not currently open,

(b) or group permission restrictions prevent closing.

(3) **openListenSock:** Opens a socket using an attendant TCL/TK interpreter. A hash table entry of the form *${hostname}($alias)* will contain the socket ID. The port number and its alias are defined in the genericAPI's resource file. The interpreter listening to the socket can be a fully functional master or slave TCL/TK interpreter or a *safe* master or slave TCL/TK interpreter, possessing a strictly defined available command set depending on the context in which **openListenSock** is used. The command set available at the interpreter is a part of the API that *owns* the socket. A *key* may be required to evaluate any or all commands as specified in the resource file.
Usage: **set intID [openListenSock alias {safe}]**
where **alias** is the socket's port number or alias as defined in the genericAPI's resource file and **safe** is an option used to specify a safe interpreter.
Return Object: Normally returns the interpreter ID, or a TCL/TK exception if

(a) socket already open,

(b) port alias not specified in resource file,

(c) socket open fails,

(d) insufficient privilege.

(4) **closeListenSock:** Safe close for listening socket which will finish pending communications before closing the socket. The associated TCL/TK interpreter will be explicitly terminated when the socket is closed. The associated socket array entry will be modified to remove its socket ID.
Usage: **closeListenSock alias**
where **alias** is the socket's port number or alias as defined in the genericAPI's resource file.
Return Object: TCL/TK exception if

(a) socket not already open,

(b) interpreter terminates improperly,

(c) port alias not specified in resource file,

(d) insufficient privilege.

(5) **operatorCmd:** Sends a command to an API to be processed by the remote TCL/TK interpreter. Commands consist of TCL/TK code

which may be interpreted by a safe interpreter. Certain exceptions will cause a help window to be generated describing the commands available at the remote API and a cross reference to API's which know about the command that generated the exception if any. Commands received at the remote API's interpreter are queued and served to the interpreter on a FIFO basis.
Usage: **operatorCmd api command {key}**
where **api** is the name of the remote LDAS API that will interpret the extended TCL/TK code represented in **command**, and **key** is an optional security key used to authenticate privilege to execute commands on a remote API with a default value determined by the genericAPI's resource file.
Return object: TCL/TK exception if

(a) the socket for communication is not open,

(b) the remote socket is not listening and times out.

(6) **emergencyCmd:** High priority command used to communicate commands which need to be evaluated immediately. This command should have restricted usage.
Usage: **emergencyCmd api command {key}**
where **api** is the name of the remote LDAS API that will interpret the extended TCL/TK code represented in **command**, and **key** is an optional security key used to authenticate privilege to execute commands on a remote API with a default value determined by the genericAPI's resource file.
Return object: TCL/TK exception if

(a) the socket for communication is not open,

(b) the remote socket is not listening and times out.

(7) **pingAPI:** Check to see that an API's ports are alive. Sends a ping to both the *operator* and *emergency* ports of an API and then generates a log entry. If the ping does not return, an appropriate exception is generated. The ping consists of the TCL/TK string *ping*.
Usage: **set pingtime [pingAPI api {timeout}]**
where **api** is the name of an LDAS API, the resource file will identify the hostname and ports for each named API, and **timeout** is the number of milliseconds to wait for a live response (*defaults to 5000*).
Return object: Returns the ping round-trip time and local host and remote host clock times. A TCL/TK exception if

(a) named API does not exist in resource file.

d) These are miscellaneous commands used as part of the genericAPI but may be used in other context as needed.

(1) **popMsg:** Pops up an undecorated message widget in a prominent location which automatically disappears after a set time. Used for low priority exception handling where it is only necessary to inform the user that he is not going to get the expected action. Messages will persist a minimum of 1 second and a maximum of 5 seconds. If **popMsg** determines that the TK toolkit is not available, it defaults to sending the messages to STDERR. If the variable LOCALLOG is set to a path/filename in the resource file, all messages will be time-stamped and copied to the file location specified by LOCALLOG.
Usage: **popMsg message {time}**
where **message** is the string to appear in the message widget and **time** is the duration in milliseconds that the message is to appear, ranging no less than 1000 millisecond and no more than 5000 milliseconds (*default value of 2500 milliseconds*).
Return object: TCL/TK exception if unable to execute message.

e) These commands are used as part of the genericAPI's resource file facility but may be used in other context as needed.

(1) **sourceRsc:** Initialization function which can be called from a TCL/TK interpreter to cause modifications to the resource file to be inherited.
Usage: **sourceRsc api**
where **api** is the name of the LDAS API whose resource file is to be interpreted.
Return object: TCL/TK exception when

(a) named API doesn't exist,

(b) error occurs while sourcing resource file.

(2) **validateRsc:** Opens a resource file and verifies the contents, prompting the user to set missing environment variables and verifying that the settings are valid. A GUI is launched if anything vital is missing. The GUI explains the problem and it's solution and points to system specific help when applicable. This command is called by **sourceRsc** immediately before the resource file is actually interpreted.
Usage: **validateRsc api**
where **api** is the name of the LDAS API whose resource file is to be interpreted.
Return object: TCL/TK exception when

(a) named API doesn't exist,

(b) error occurs while sourcing resource file.

2. The genericAPI.rsc Resource File Specification:

a) The genericAPI.rsc resource file, in common with other resource files associated with different API's and interfaces in the LDAS, consists primarily of individual lines of TCL/TK code with no interdependencies, allowing essentially arbitrary ordering of resource information. The resource file is *sourced* when an API is started up, and provides information to the interpreter which is site specific or in the nature of a user interface.

b) Typical resource information would include aliases for machine ports, host names and API names, encrypted system and user keys for access and authentication, local system defaults and environment variables, and user preferences such as colors and fonts.

c) Users who write their own API's based on the genericAPI are encouraged to source their own resource file, based on the genericAPI.rsc, to establish unique preferences and avoid conflicts.

d) A default resource file is included with the genericAPI with verbose comments and a help file explaining it's use in detail.

e) If a required resource file is not found an exception is thrown with helpful instructions for configuring the API (see **sourceRsc**).

f) Required parameters which would generate exceptions without specified values include the default location for resource files, help files and LDAS libraries, as well as the name of the local host computer.

g) The values of encrypted keys will be calculated by a standard message digest or hashing algorithm.

3. Specification of the LDAS HTML text widget:

   a) The *LDAS HTML text widget* is a top-level window containing a plain TK text widget, with associated menus, entries, and buttons as defined by the **showHelp** command supporting a subset of *HTML 2.0* excluding tables, forms, multi-columns, *JPEG* images, and image maps (*these may be made available in the future*). It will support additional tags defined for the *Internal LDAS Light-Weight Data Format*. The **renderHTML** and the **renderURL** commands call the **showHelp** command with the name of the widget to be created. If the name provided for the widget is that of an existing widget which is not consistent with an *LDAS HTML text widget*, the **showHelp** command will throw an exception.

4. Specification of TCL/TK exceptions:

   a) Each TCL/TK command returns with a TCL/TK exception when the circumstances associated with the call warrant throwing an exception. These exceptions may be generated by more than one anomalous condition. To address the particulars of the condition that generated an exception, each TCL/TK exception will have associated with it unique integer ID's and descriptive messages allowing the exact cause of the exception

to be traced.

C. The genericAPI.so Package's Specification:

1. The following is a list of C/C++ language based extension to the TCL/TK command language which are implemented in the genericAPI.so shared object package. These extended commands are added to TCL/TK through the TCL/TK interface library and made available to the TCL/TK interpreter using the *loadable package* mechanism in TCL/TK.

   a) These extended commands are used to manage a C++ socket class library which allows binary data to be communicated between LDAS API's in either streams or as C++ objects.

      (1) **createDataSocket:** Creates a data socket at the specified port and address. The port and address are optional. The default address is the IP address of the machine returned by *gethostname*. The default port is zero, which cause the system to choose an unused port using a call to the **getSocketPort** command.
      Usage: **set ptSok [createDataSocket {port} {address}]**
      where **port** is an optional (but highly recommended) port number. The **address** is the optional IP number associated with the local host, but may be used to specify particular IP addresses when used on a gateway machine. ***Note:*** *In order to specify an address, the port must also be specified!*
      Return object: This command returns a pointer to a socket which can be stored in a TCL/TK variable as in the case of **ptSok** above.
      This command throws a TCL/TK exception under the following conditions:

         (a) bad_alloc - memory allocation fails,

         (b) bind_failure - unable to bind socket to the specified address & port,

         (c) invalid_address - the address is not a valid IP address,

         (d) invalid_host - the host to which address refers can't be found.

      (2) **createServerSocket:** Creates a server socket at the specified port and address and listens for connections. The port and address are optional. The default address is the IP address of the machine returned by *gethostname*. The default port is zero, which causes the system to choose an unused port using a call to the **getServerPort** command.
      Usage: **set ptSrv [createServerSocket {port} {address}]**
      where **port** is an optional (but highly recommended) port number and **address** is the optional IP number associated with the local host, but may be used to specify particular IP addresses when used on a gateway machine. ***Note:*** *In order to specify an address, the*

port must also be specified!
Return object: This command returns a pointer to a socket which can be stored in a TCL/TK variable as in the case of **ptSrv** above.
This command throws a TCL/TK exception under the following conditions:

(a) bad_alloc - memory allocation failed,

(b) bind_failure - unable to bind server to the specified address & port,

(c) invalid_address - the address is not a valid IP address,

(d) invalid_host - the host to which address refers can't be found.

(3) **connectDataSocket:** Connects the given socket to a server located at the specified address and port.
Usage: **connectDataSocket $ptSok address port**
where **ptSok** is a socket pointer that has been returned by **create-DataSocket**, **address** is the IP address or hostname of the server to connect to and **port** is the port number on which the server is listening.
Return object: none.
This command throws a TCL/TK exception under the following conditions:

(a) invalid_socket - the socket doesn't exist,

(b) connect_failure - the socket was unable to connect to the designatedaddress (for example, a server may not be listening at thataddress),

(c) invalid_address - the address is not a valid IP address,

(d) invalid_host - the host to which address refers can't be found.

(4) **acceptDataSocket:** Extracts the first pending connection request and associates it with a socket. If an address and / or port is provided, then the connection is only accepted from the specified address and / or port. The default port is zero, corresponding to any port and the default address is (*an empty string*) corresponding to any address.
Usage: **set ptSok [acceptDataSocket $ptSrv {address} {port}]**
where **ptSrv** is a server pointer that has been returned by **create-ServerSocket**, **address** is the optional IP address or hostname of the client from which to accept communications and has a default of *any client address* when not specified. The **port** is the optional port number from which to accept communications and has a default of *any client port*. In order to specify a port, the address must also be specified or the address can be specified using an empty string, allowing all addresses at a specific port.

Return object: This command returns a pointer to a socket which can be stored in a TCL/TK variable as in the case of **ptSok** above.
This command throws a TCL/TK exception under the following conditions:

(a) invalid_server - the server does not exist,

(b) invalid_address - the address is not a valid IP address,

(c) invalid_host - the host to which address refers can't be found,

(d) bad_alloc - memory allocation failed,

(e) accept_failure - the unix accept command failed,

(f) identify_failure - the source socket's identity was unable to be determined,

(g) illegal_connection - connection attempted from an unauthorized socket.

(5) **closeDataSocket:** Closes and destructs a data socket, freeing any memory allocated for it.
Usage: **closeDataSocket $ptSok**
where **ptSok** is a socket pointer variable which has previously been set by a call to **createDataSocket**.
Return object: none.
This command throws a TCL/TK exception under the following conditions:

(a) invalid_socket - the socket doesn't exist.

(6) **closeServerSocket:** Closes and destructs a data server, freeing any memory allocated for it.
Usage: **closeServerSocket $ptSrv**
where **ptSrv** is a server pointer variable which has previously been set by a call to **createServerSocket**.
Return object: none.
This command throws a TCL/TK exception under the following conditions:

(a) invalid_server - the server doesn't exist.

b) These extended commands are used to obtain information from the C++ socket class library associated with current instances of socket objects.

(1) **getSocketIpAddress:** Returns the socket's local IP address.
Usage: **set ptVar [getSocketIpAddress $ptSok]**
where **ptSok** is a socket pointer variable which has previously been set by a call to **createDataSocket**.
Return object: This command returns a string containing the local IP address of the socket which can be stored in a TCL/TK variable as in the case of **ptVar** above.

This command throws a TCL/TK exception under the following conditions:

(a) invalid_socket - the socket doesn't exist.

(2) **getSocketPort:** Returns the socket's local port.
Usage: **set ptVar [getSocketPort $ptSok]**
where **ptSok** is a socket pointer variable which has previously been set by a call to **createDataSocket**.
Return object: This command returns a string containing the local port number of the socket which can be stored in a TCL/TK variable as in the case of **ptVar** above.
This command throws a TCL/TK exception under the following conditions:

(a) invalid_socket - the socket doesn't exist.

(3) **getServerIpAddress:** Returns the server's local IP address.
Usage: **set ptVar [getServerIpAddress $ptSrv]**
where **ptSrv** is a socket pointer variable which has previously been set by a call to **createServerSocket**.
Return object: This command returns a string containing the local IP address of the socket which can be stored in a TCL/TK variable as in the case of **ptVar** above.
This command throws a TCL/TK exception under the following conditions:

(a) invalid_server - the server doesn't exist.

(4) **getServerPort:** Returns the server's local port.
Usage: **set ptVar [getServerPort $ptSok]**
where **ptSok** is a socket pointer variable which has previously been set by a call to **createServerSocket**.
Return object: This command returns a string containing the local port number of the socket which can be stored in a TCL/TK variable as in the case of **ptVar** above.
This command throws a TCL/TK exception under the following conditions:

(a) invalid_server - the server doesn't exist.

(5) **getSocketPeerIpAddress:** Returns the peer's IP address to which this socket is connected.
Usage: **set ptVar [getSocketPeerIpAddress $ptSok]**
where **ptSok** is a socket pointer variable which has previously been set by a call to **createDataSocket**.
Return object: This command returns a string containing the peer's IP address for the socket which can be stored in a TCL/TK variable as in the case of **ptVar** above.

This command throws a TCL/TK exception under the following conditions:

(a) invalid_socket - the socket doesn't exist,

(b) unconnected_socket - the socket is not connected.

(6) **getSocketPeerPort:** Returns the peer's port to which this socket is connected.
Usage: **set ptVar [getSocketPeerPort $ptSok]**
where **ptSok** is a socket pointer variable which has previously been set by a call to **createServerSocket**.
Return object: This command returns a string containing the peer's port number for the socket which can be stored in a TCL/TK variable as in the case of **ptVar** above.
This command throws a TCL/TK exception under the following conditions:

(a) invalid_socket - the socket doesn't exist,

(b) unconnected_socket - the socket is not connected.

c) These extended commands are used to store, restore and reset the information contained in C++ socket objects currently instantiated in the genericAPI.

(1) **save:** Closes all sockets, writing information about their connections to the given file (*which is overwritten if it already exists*).
Usage: **save filename**
where **filename** is the name of the file where all socket objects are to be stored.
Return object: none.
This command throws a TCL/TK exception under the following conditions:

(a) file_creation_failed - the file could not be created.

(2) **restore:** Restores socket connections as written in the given file (*which must have been written by the save command*).
Usage: **restore filename**
where **filename** is the name of the file where the socket objects were previously stored with the **save** command.
Return object: none.
This command throws a TCL/TK exception under the following conditions:

(a) file_not_found - the file could not be located on system,

(b) bad_alloc - insufficient memory available.

(3) **reset:** Reset is used to clean up all C++ objects that have been instantiated in the genericAPI.so package. This command is prima-

rily meant as a full reset to initial state for the package, freeing up all dynamic memory that has been allocated.
Usage: **reset**
Return object: none.

d) These extended commands are used to communicate elements of the *Internal LDAS Light-Weight Data* and *Raw Binary Data* between API's using the *Data Sockets*.

(1) **sendElementAscii:** This command sends an *Internal LDAS Light-Weight Data* set, called an *Element* because of its relationship to *XML* elements, through a *Data Socket* in *ASCII* form *(this includes **base64** formatted data)*. This method of sending *Elements* is not expected to be used often and is provided for completeness.
Usage: **sendElementAscii ptSok ptElem**
where **ptSok** is a pointer to a *Data Socket* which has previously been opened with the **createDataSocket** command, and **ptElem** is a pointer to an *Element* object that has previously been instantiated in the C++ layer.
Return object: none.
This command throws a TCL/TK exception under the following conditions:

(a) invalid_socket - the socket doesn't exist,

(b) unconnected_socket - the socket isn't connected,

(c) invalid_element - the element doesn't exist.

(2) **sendElementObject:** This command sends an *Internal LDAS Light-Weight Data* set, called an *Element* because of its relationship to *XML* elements, through a *Data Socket* as a C++ *Object*. This method of sending *Elements* be used often because of efficiency.
Usage: **sendElementObject ptSok ptElem**
where **ptSok** is a pointer to a *Data Socket* which has previously been opened with the **createDataSocket** command, and **ptElem** is a pointer to an *Element* object that has previously been instantiated in the C++ layer.
Return object: none
This command throws a TCL/TK exception under the following conditions:

(a) invalid_socket - the socket doesn't exist,

(b) unconnected_socket - the socket isn't connected,

(c) invalid_element - the element doesn't exist.

(3) **sendRawBinary:** This command sends a raw stream of binary data through a *Data Socket*. The binary data must have been previously been instantiated in the C++ binary storage class which includes an

attribute for the number of bytes. This command is used to send raw unstructured or arbitrarily structured data through the socket. No attempt is made to understand the content of the data.
Usage: **sendRawBinary ptSok ptBin**
where **ptSok** is a pointer to a *Data Socket* which has previously been opened with the **createDataSocket** command, and **ptBin** is a pointer to an *Binary* object that has previously been instantiated in the C++ layer.
Return object: none.
This command throws a TCL/TK exception under the following conditions:

(a) invalid_socket - the socket doesn't exist,

(b) unconnected_socket - the socket isn't connected,

(c) invalid_element - the element doesn't exist.

(4) **recvElementAscii:** This command receives an *Internal LDAS Light-Weight Data* set from a *Data Socket* in *ASCII* form *(this includes **base64** formatted data)*.
Usage: **set ptElem [recvElementAscii ptSok]**
where **ptSok** is a pointer to a *Data Socket* which has previously been connected with the **acceptDataSocket** command, and **ptElem** is a pointer to the *Element* object which is instantiated in the C++ layer by the receiver.
Return object: This command returns a pointer to an *Element* object that has been received through the *Data Socket*. The format attribute for the *Element* is guaranteed to be *ASCII* after this command is called.
This command throws a TCL/TK exception under the following conditions:

(a) invalid_socket - the socket doesn't exist,

(b) unconnected_socket - the socket isn't connected,

(c) bad_alloc - insufficient memory for element.

(5) **recvElementObject:** This command receives an *Internal LDAS Light-Weight Data* set from a *Data Socket* in C++ object form.
Usage: **set ptElem [recvElementObject ptSok]**
where **ptSok** is a pointer to a *Data Socket* which has previously been connected with the **acceptDataSocket** command, and **ptElem** is a pointer to the *Element* object which is instantiated in the C++ layer by the receiver.
Return object: This command returns a pointer to an *Element* object that has been received through the *Data Socket*. The incoming format attribute for the *Element* is unaltered by this command.

This command throws a TCL/TK exception under the following conditions:

(a) invalid_socket - the socket doesn't exist,

(b) unconnected_socket - the socket isn't connected,

(c) bad_alloc - insufficient memory for element.

(6) **recvRawBinary:** This command receive a raw stream of binary data through from *Data Socket*. This command is used to receive raw unstructured or arbitrarily structured data from the socket. No attempt is made to understand the content of the data.
Usage: **set ptBin [recvRawBinary ptSok]**
where **ptSok** is a pointer to a *Data Socket* which has previously been connected with the **acceptDataSocket** command, and **ptBin** is a pointer to the *Binary* object which is instantiated in the C++ layer by the receiver.
Return object: This command returns a pointer to a *Binary* object that has been received through the *Data Socket*.
This command throws a TCL/TK exception under the following conditions:

(a) invalid_socket - the socket doesn't exist,

(b) unconnected_socket - the socket isn't connected,

(c) bad_alloc - insufficient memory for raw binary.

e) These extended commands are used to communicate elements of the *Internal LDAS Light-Weight Data* and *Raw Binary Data* to and from the TCL/TK layer and the underlying C++ layer.

(1) **putElement:** This command puts an *Internal LDAS Light-Weight Data* set, called an *Element* object, into the C++ layer from a binary string variable (*the variable doesn't necessarily contain non-printable character*) in the TCL/TK layer. The command returns a pointer to the Element object which has been instantiated in the C++ layer by this command. The data stored in the binary string variable can be translated into a new format in the Element object using the format and compress options.
Usage: **set ptElem [putElement $bstring {format} {compress}]**
where **bstring** is a TCL/TK variable containing either an ASCII or Binary string for an *Internal LDAS Light-Weight Data Element*. The optional **format** and **compress** parameter is used to translate the data within the **bstring** (along with the associated attributed) into a different format within the instantiated *Element* object in the C++ layer, and **ptElem** is a pointer to the *Element* object which is instantiated in the C++ layer. The **format** can be one of {**ascii** | **binary** | **base64**}. The **compress** can be a single integer value from {**0** - **9**}

where **0** is no compression and **9** is maximum compression. The **compress** parameter is ignored when the **format** is **ascii**. The default for **format** is "*attribute"* causing the *Internal LDAS Light-Weight Data Format* attribute value to be used.
Return object: This command returns a pointer to an *Element* object.
This command throws a TCL/TK exception under the following conditions:

(a) bad_alloc - insufficient memory for element,

(b) illegal_element - TCL/TK bstring format is illegal,

(c) illegal_format - translation format unrecognized,

(2) **getElement:** This command gets an *Internal LDAS Light-Weight Data* set which has previously been instantiated as an Element object in the C++ layer and stores it in a binary string variable in the TCL/TK layer. The format of the data in the bstring representation of the Element object is optional and defaults to the attribute's value and if the format is binary or base64, then an optional compression factor can be specified.
Usage: **set bstring [getElement ptElem {format} {compress}]**
where **ptElem** is a pointer to the *Element* object which is instantiated in the C++ layer. The optional **format** can be one of {**ascii** | **binary** | **base64**}. The **compress** option is a integer value between 0 and 9, including no string meaning to leave the data at its current compression level as assigned by the attribute. The **compress** parameter is ignored when the **format** is **ascii**. The **bstring** is the TCL/TK variable to contain either an ASCII or Binary string for an *Internal LDAS Light-Weight Data Element.*The default for **format** is "*attribute"* causing the *Internal LDAS Light-Weight Data Format* attribute value to be used.

Return object: This command returns an *Internal LDAS Light-Weight Data Element* as a TCL/TK binary string.
This command throws a TCL/TK exception under the following conditions:

(a) bad_alloc - insufficient memory for element,

(b) invalid_element - the element does not exist,

(c) illegal_format - translation format unrecognized.

(3) **putRawBinary:** This command puts a raw binary data set into the C++ layer in the form of a binary object containing the raw binary data and the number of bytes associated with the raw binary data. It returns a pointer to the binary object. No attempt is made to parse

the raw binary data.
Usage: **set ptBin [putRawBinary $bstring $nbytes]**
where **bstring** is a TCL/TK variable containing the raw binary data, **nbytes** is a TCL/TK variable containing the number of bytes associated with the raw binary data, and **ptBin** is a pointer to the binary object instantiated in the C++ layer by this command.
Return object: This command returns a pointer to the binary object instantiated in the C++ layer when called.
This command throws a TCL/TK exception under the following conditions:

(a) bad_alloc - insufficient memory for binary object.

(4) **getRawBinary:** This command gets a raw binary data set from a binary object in the C++ layer, returning a binary string containing the raw binary data and modifies the value of the second parameter to be the number of bytes associated with the raw binary data. No attempt is made to parse the raw binary data.
Usage: **set bstring [getRawBinary ptBin nbytes]**
where **ptBin** is a pointer to a binary object in the C++ layer which has previously been instantiated, **nbytes** is a TCL/TK variable which will be updated to hold the number of bytes associated with the raw binary data, and **bstring** is the TCL/TK variable to contain the binary data after the call is made.
Return object: This command returns a TCL/TK binary string.
This command throws a TCL/TK exception under the following conditions:

(a) bad_alloc - insufficient memory for binary string,

(b) invalid_binary - binary object doesn't exist.

(5) **destructElement:** This command deallocates memory for an element object, removing it from the C++ layer. It takes one argument which is a pointer to the element to be removed.
Usage: **destructElement ptElem**
where **ptElem** is a pointer to an element object previously instantiated in the C++ layer.
Return object: None. This command returns a TCL/TK exception under the following conditions:

(a) invalid_element - specified element doesn't exist.

(6) **destructRawBinary:** This command deallocates memory for a raw binary object, removing it from the C++ layer. It takes one argument which is a pointer to the raw binary object to be removed.
Usage: **destructRawElement ptElem**
where ptElem is a pointer to a raw binary object previously instanti-

ated in the C++ layer.
Return object: None. This command returns a TCL/TK exception under the following conditions:

(a) invalid_binary - specified raw binary object doesn't exist.

D. The Internal LDAS Light-Weight Data Format Specification:

1. The *Internal LDAS Light-Weight Data Format* is a subset of the *LIGO Light-Weight Data Format*. Both are based on *XML*, the likely successor to *HTML*. However, the *Internal LDAS Light-Weight Data Format* is designed to be the minimal set of elements needed to move data through sockets and between the TCL/TK layer and the extended commands found in the C/C++ layer. It is primarily meant to be a machine oriented data format, and as such, relies heavily on attributes over nested elements. The LDAS system will support the full implementation of the *LIGO Light-Weight Data Format* using a specialized API, the *Light-Weight Data Format API*, which will be specified in a forthcoming set of documents.
2. Each element of the *Internal LDAS Light-Weight Data Format* is of the form:
<tag attribute1="value" attribute2="value" ...>rawdata</tag>
where the element begins with a < character and is followed by the **tag**-name which identifies the base data type. Then the **attributes** are listed, each set equal to a **value** enclosed in quotes and provide descriptions about the data. The opening **tag** is closed with a > character. The **rawdata** then follows. The element is terminated by the closing **tag** which is just the < character followed by the / character followed by the **tag**-name and finally the > character.
3. The tag-names are case-insensitive and can be any of the following for the *Internal LDAS Light-Weight Data Format*:
   a) **CHAR_S** - signed byte,
   b) **CHAR_U** unsigned byte,
   c) **INT_2S** - 2 byte signed integer,
   d) **INT_2U** - 2 byte unsigned integer,
   e) **INT_4S** - 4 byte signed integer,
   f) **INT_4U** - 4 byte unsigned integer,
   g) **REAL_4** - 4 byte IEEE 754 floating point number,
   h) **REAL_8** - 8 byte IEEE 754 floating point number,
   i) **COMPLEX_8** - pair of REAL_4's ordered as (*real, imaginary*),
   j) **COMPLEX_16** - pair of REAL_8's ordered as (*real, imaginary*).
4. The understood attributed for the *Internal LDAS Light-Weight Data Format* are (*all other attributes are ignored by the genericAPI*):
   a) **name = "name:attr1:attr2:..."** - the name or names to be associated

with this data. A set of naming attributes can appear after the name separated with colons. This would be useful for describing generic objects like graphs or tables where title, axis labels, etc. might be associated. Defaults to an empty string if not present;

b) **ndim = "integer"** - the number of dimension in the rawdata and has a default value of 1 (*vector*) if not present;

c) **dims = "integer,integer,..."** - ndim comma delimited integers telling the number of elements in each dimension of the rawdata; if ndim=0 (*scalar*) then dims is ignored and defaults to 1;

d) **units = "unit1,unit2,..."** - ndim comma delimited unit names specifying the units, if any, for each dimension of the rawdata. Defaults to an empty string if not present;

e) **mdorder = "f77 | c"** - indicates whether a multidimensional data set is to be incremented fastest on the first index "f77" or last index "c"; the default is "c" if not present;

f) **format = "ascii | base64 | binary"** - the encoded format of the rawdata; if "ascii" then compression is not allowed and each number is whitespace *(spaces, tabs, returns)* separated; the default is binary if not present;

g) **compression = "0 - 9"** - an integer between 0 and 9 specifying the level of compression used for binary or base64 formats; the default is 0 (*no compression*) if not present;

h) **byteorder = "little | big"** - whether integers are stored in little endian or big endian order; the default is little endian if not present;

i) **bytes = "integer"** - number of bytes of rawdata between the element's tags-names; required if format is binary and compression not zero;

j) **comment = "arbitrary text string"** - optional and defaults to an empty string if not present.