U.S. DEPARTMENT OF
ENERGY

PNNL-22589

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

The System Event Logger

DES-0007

Revision 4

Charlie Hubbard

March 2013

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by **Battelle** *Since 1965*

The System Event Logger

Charlie Hubbard

DES-0007
Revision 4
March 2013

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor Battelle Memorial Institute, nor any of their employees, *makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights.* Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or Battelle Memorial Institute. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under
Contract DE-AC05-76RL01830

Printed in the United States of America

Available to DOE and DOE contractors from the Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831-0062
ph: (865) 576-8401
fax: (865) 576-5728
email: reports@adonis.osti.gov

Available to the public from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161
ph: (800) 553-6847
fax: (703) 605-6900
email: orders@ntis.fedworld.gov
online ordering: http://www.ntis.gov/ordering.htm

Contents

1 Introduction

The system event logger is a standard system server that provides general message logging services to the control system. Like all system servers, the event logger is implemented as a C++ class called `LoggerServer`, which is derived from the `BaseServer` class (see *cliserv.h* and *cliserv.cpp*).

As is standard for system servers, there is also an accompanying client interface class, called `LoggerClient`, which is derived from the `BaseClient` class (see *cliserv.h* and *cliserv.cpp*). The `LoggerClient` class provides API functions needed by clients to communicate with the server.

The logger server and its client interface rely heavily on the the inter-process communication framework, and message handling mechanisms inherited from their parent classes. For that reason, to achieve a detailed understanding of the event logger, the reader should first be thoroughly familiar with the `BaseServer` and `BaseClient` classes. These are fully described in design document ***DES-0005, The Client/Server Architecture***.

In the code, the definition and implementation for the event logger can be found in the file *loggerServer.cpp*. The client interface library for the server is defined and implemented in *loggerClientLib.h* and *loggerClientLib.cpp*. Also, a simple text-based menu client for the server exists, and it is implemented in *loggerMenu.cpp*.

The primary documentation for the event log server source code is the Doxygen-generated HTML documentation associated with each of the above source files. That documentation set is automatically built based on the source code itself. It provides the most detailed, most up-to-date descriptions of the code. The document you are reading now is supplemental, and is intended to provide deeper background for the server, and its client API. If contradictions between this document and the Doxygen-generated documentation are found, the Doxygen-generated documentation should be considered correct.

2 Basic Operation

During the course of a system run, it is desirable to maintain a file-based log of events that occurred during the run. Such a capability provides a permanent record of how a particular run progressed, and serves as a valuable diagnostic for determining the cause of any problems that were encountered during the run. The system event logger provides this capability.

The event logger maintains a file on disk called the *event log* file[1]. At any time, clients can send messages to the event logger and the event logger will add them to the log file in the order received.

[1]The situation is actually slightly more complicated than this. Please see section 3 on *file channels* for details.

Optionally, the logger may also send its log messages to an IRC server for convenient, real-time viewing. This ability is enormously useful during project development and testing, and can also be useful during normal operation (see section 8 for details).

The event logger is a *pure server*, meaning it is available to respond to requests from external clients, but it never generates requests of its own to other servers. As a pure server, the event logger sits at the lowest level in the system server hierarchy.[2] All other servers in the system (and most/all client-only applications as well) maintain client connections to the event logger.

In addition to the text of the log messages themselves, each record written to the event log file carries with it several other bits of information such as the name of the client that sent the message, the time the message was sent, the location in the code where the message was generated, and so on (these will be discussed in detail later on). Finally, each message is assigned a priority value by the sending client. The priority value tags a message with an appropriate level of importance. The event logger is configured with a minimum threshold of importance a message must meet before it is included in the event log file. Messages that do not meet the threshold are silently discarded without being logged. This makes it possible (for instance) to record very detailed run logs during development, but scale back to less storage-intensive logs on deployed systems, perhaps only recording errors and warnings. More will be said about priorities later on in section 6.

3 File Channels

The event log server supports the concept of *file channels*. On startup, the server allocates one or more of these file channels (the exact number to allocate is specified on the command line). Individual file channels are referred to by number, beginning with zero and increasing sequentially.

The way this works is simple. When clients tell the server to begin logging to a new file, they also specify a *file channel* ID to associate with the file. Conversely, when clients tell the logger to stop logging, they specify a particular file channel to shut down. When new log messages are sent to the server, the server writes them to every file channel currently associated with an open file.

At first glance, file channels appear to do nothing but create a lot of redundant files. After all, when new data is available, that same data is written to all open file channels. To what end?

The need for file channels springs from several past and present projects in which a sample is processed and then moved into a nuclear counting cell, where it it sits for a very long time being analyzed. These systems typically support multiple nuclear counting cells. As soon as one counting cell is loaded with sample, it is possible to start processing another sample with the intent of loading it into a different counting cell. The result is that at any given

[2] See design document ***DES-0005, The Client/Server Architecture*** for a description of server hierarchy.

time, multiple samples can be in various states of staggered but overlapping processing or analysis. Because data is typically reported on a per-sample basis, it is convenient to have an event log record that covers a particular sample from its initial processing to its final analysis. File channels handle this situation. When a new sample begins processing, a new log is opened for it on an unused file channel. When final analysis of that sample is complete, the associated file channel can be closed. Other samples that are in other states of processing/analysis continue to log messages on their associated file channels.

Of course, not all projects will have a situation like described above in which multiple, simultaneous (but staggered) event logs are desirable. Initially, support for file channels was thought to be highly project-specific and unneeded in what is supposed to be, a general-purpose event logger. In fact, the initial implementation of file channels was a modification of an earlier, single file event logger that was made specifically to support a staggered sample project similar to the one described above. However, support for file channels has been implemented in such a way as to be invisible to clients that do not require it. In particular, the client API messages for starting and stopping a log file default their file channel fields to channel 0. Therefore, if clients neglect to fill in the field, the logger always acts on channel 0, the default file channel. Because the addition of file channels does not complicate the client interface to the server in any way for clients that don't require file channel support, it was decided to tolerate the feature in this general-purpose server, even though it may often go unused.

4 Client Request Messages

In addition to the standard set of client request messages that all servers support[3], the logger server implements six additional messages. Those are briefly described in this section. For full details on the client API functions that generate these messages, please refer to the Doxygen documentation for the `LoggerClient` class.

4.1 LOG_MESSAGE

This message allows clients of the logger server to record a new message to the log file.

4.2 LOG_GET_PRIORITY

This requests the current logging priority set on the server. This is the minimum priority level a new message must have in order to be included in the log file. Messages sent to the logger tagged with priorities lower than this value will be silently discarded.

[3]See design document ***DES-0005, The Client/Server Architecture*** for more details on the standard set of client request messages.

4.3 LOG_SET_PRIORITY

This message allows the client to set the event log server's minimum logging priority to any valid priority level.

4.4 LOG_START

This message instructs the event log server to open a new log file with the specified name and associate it with the specified file channel ID. Keep in mind that the API function that implements this message defaults the file channel ID to zero. In this way, clients that do not care about multi-file support (file channels), do not even need to know that the server supports the feature.

4.5 LOG_STOP

This message instructs the event log server to close the log file associated with the specified file channel ID. Keep in mind that the API function that implements this message defaults the file channel ID to zero. In this way, clients that do not care about multi-file support (file channels), don't even need to know that the server supports the feature.

4.6 LOG_STOP_ALL

This message instructs the event log server to close the log files associated with every file channel. It is not an error if some or all file channels are not currently logging.

4.7 LOG_GET_FILE_NAMES

This message returns a list of files that are currently open for logging. On systems that are not taking advantage of multi-file support, the list will never contain more than one entry.

5 Message Fields

As mentioned briefly in section 2, in addition to a free-format text message, each record sent to the event logger and written to file contains several additional fields. There are eight fields in all.

- *timestamp* — This is a ten-digit time stamp representing the real-world time when the record was written to the log file. The time stamp is in standard POSIX `time_t` format, which is to say it represents the number of seconds since midnight January 1st, 1970 GMT. POSIX compliant operating systems, like Linux and QNX, provide

many utility functions for manipulating `time_t` style timestamps and rendering them in various, more human-readable, forms.

- *priority* — This is the priority level the client set on the message when he sent it to the log server. Much more is said about log priority levels in section 6 below.
- *process name* — In our standard client/server architecture, all clients and all servers are identified by short text names that are unique across the project. This field contains the name of the client that sent the message to the event log server.
- *client name* — Many messages sent to the log server are sent by other system servers, and many of those log messages are generated as the sending server works to fulfill a request from one of its clients. This field contains the name of that server's client. The intention is to record not just what the server is doing, but also what client asked it to do it. Do not confuse this field with the *process name* field!

  The *client name* field is confusing. Perhaps an example will help to make this point more clear.

  Assume a system has two servers called `valves` and `control`. The `valves` server maintains a list of valves on the system and allows its clients to open or close them. The `control` server is a higher level server that is responsible for sequencing valves in a particular order to perform a specific task. The `control` server accomplishes this by sending messages to the `valves` server. Any time the `valves` server changes the state of a valve, it also sends a message to the event log server to note the change. In those log messages, the *process name* field will contain the string "valves". That is, it is the `valves` server that has sent the message to the logger. The *client name* field, on the other hand, will contain the string "control". That is, the change to the valve that the `valves` server made, was at the request of the `control` server.
- *line number* — This is the line number of the line in the source code where this message was sent to the logger. Now normally, no programmer sending a message to the logger server is going to know the line number of the line from which he makes that call, and, even if he did, the line number is likely to change as code is added or deleted above that line. Fortunately, the compiler provides a special macro, called `__LINE__`, that automatically computes the line number at compile-time, so the programmer doesn't have to worry about it. Also, the client API to the event log server provides its own macro, called `LOG_MSG()`, that automatically inserts the current line number value into the message. Programmers that use the `LOG_MSG()` macro, never have to worry about this field at all.
- *file name* — This is the name of the source code file that generated this particular message to the log. Like the *line number* field, this field is automatically filled in when the programmer generates the log message using the `LOG_MSG()` macro, so he doesn't need to worry about keeping this field explicitly up to date.
- *function name* — This is the name of the function that generated this particular message to the log. Like the *line number* and *file name* fields, this field is automatically

filled in when the programmer generates the log message using the `LOG_MSG()` macro, so he doesn't need to worry about keeping this field explicitly up to date.

- *message* — This is the actual free-form text message the client wants added to the event log.

6 Message Priority Levels

The event logger supports seven priority levels. These are defined by an anonymous `enum` located in *loggerClientLib.h*. The `enum` definition is reproduced below with descriptions to follow.

```
enum {   // logger priority levels
   LOG_PRI_HIGHEST,
   LOG_PRI_FATAL,
   LOG_PRI_ERROR,
   LOG_PRI_WARNING,
   LOG_PRI_MESSAGE,
   LOG_PRI_CHANGE,
   LOG_PRI_STATUS
};
```

Ultimately, authors of event logger client applications are free to use any priority they want for any message. However, when the event logger was written, each priority was intended to have a specific meaning. Consider the following list as set of suggested guidelines.

- `LOG_PRI_HIGHEST` — This is the highest level priority defined. It is unique in that messages of this priority can not be blocked by the event logger. They are *always* logged. By convention, in the code base, `LOG_PRI_HIGHEST` is used to report all application startups and shutdowns.
- `LOG_PRI_FATAL` — This priority level is reserved for reporting error conditions that are so severe that the reporting application could not recover and had to exit.
- `LOG_PRI_ERROR` — This priority level is reserved for reporting error conditions that, although serious, are not fatal.
- `LOG_PRI_WARNING` — This priority level is used to report unusual circumstances that are not necessarily errors, but should still be brought to the attention of the system operator.
- `LOG_PRI_MESSAGE` — This priority level is used for arbitrary messages that report events without an error connotation.
- `LOG_PRI_CHANGE` — This priority level is used to report changes to an application's state. For instance, a server that manages valves may report changes of valve state at this priority level. By convention, real projects typically run their event loggers set such that `LOG_PRI_CHANGE` messages are the lowest priority messages to actually get saved. Of course this can be set to whatever makes sense for a particular project.

- `LOG_PRI_STATUS` — This seldom used priority level was intended to be used to report client requests for information that would not result in a change of state on the server. For example, asking whether or not a specific valve is open or closed is a request for status. A command to open or close a valve represents a change of state.

7 Log Data File Format

In this section we describe the format of the event log files the server writes.

The server's log files are text format files, organized such that each new log record takes one new line in the file. The first line is always a *header* line. All additional lines are log records. All lines, including the header line, are ASCII strings encoded in DSP format[4]. All lines are terminated with a line-feed character.

7.1 The Header Line

Every event log file begins with a header line that describes something about the system that created the file. The header line format is shown below. As can be seen, this is a DSP format string, therefore it is most easily parsed with the `MessageParser` utility class.

format

```
(header,<version>,(<projectID>,<systemID>,<location>))
```

- `header` — This is the string literal "header". The sole purpose of this field is to identify the line as the header line.
- `<version>` — This field contains the file format version. As of this writing, version 1 is the most current. There are no changes anticipated, but if changes are made, this field provides a way for a viewer program to recognize different file versions and parse/display them accordingly.
- `<projectID>` — This field identifies the *type* of system that generated the log file (i.e. a DAS system, a MRVL system, and so on).
- `<systemID>` — This field identifies the *specific* system that generated the log file (this could be something like a site identifier, USX01 say).
- `<location>` — This identifies the system's location. During development, this is typically a room number.

The last three items in the above list are populated from environment variables that are assumed to already be defined on the control computer at the time a new log file is started. The `<projectID>` field corresponds to the `LOG_PROJECT_ID` environment variable, `<systemID>`

[4]See design document ***DES-0002, Data Serialization Protocol*** fore complete details on DSP encoding.

corresponds to LOG_SYSTEM_ID, and <location> corresponds to LOG_LOCATION. If one or more of these environment variables is not defined when the log file is started, the string "Unknown" is used instead.

7.2 Log Record Lines

All remaining lines in the file are log records. The format for a log record line is shown below.

```
(<timestamp>,<priority>,<processName>,<clientName>,<lineNumber>,<fileName>,<functionName>,<message>)
```

7.3 Log File Example

The following is a short example of a log file

```
(header,1,(Unknown,Unknown,Unknown))
(1271721811,5,logger,,554,loggerServer.cpp,_Start,Started new log file ~(event.log~))
(1271721811,0,logger,,249,loggerServer.cpp,LoggerServer,Server startup complete)
(1271721816,0,anadig,,242,optoServer.cpp,OPTOServer,Server startup complete)
(1271721817,4,pid,,4360,pidServer.cpp,Initialize,Configuration complete)
(1271721817,0,pid,,2745,pidServer.cpp,PIDServer,Server startup complete)
(1271721818,0,control,,716,controlServer.cpp,ControlServer,Server startup complete)
(1271721819,0,mks,,803,mks910Server.cpp,MKS910Server,Server startup complete)
...
(1271721866,5,anadig,WebGUI,689,optoServer.cpp,SetDigital,Output 'v503' changed to ON)
(1271721961,0,WebGUI,,266,wtWebGUI.cpp,main,Application shutting down)
(1271721964,0,sohMenu,,154,sohMenu.cpp,main,Application startup)
(1271721987,5,soh,sohMenu,512,sohServer.cpp,Stop,Closing SOH log file ~(~))
...
(1271958667,0,logger,,261,loggerServer.cpp,~LoggerServer,Server shutting down)
(1271958667,5,logger,,573,loggerServer.cpp,_Stop,Closing log file ~(event.log~))
```

Note that most of these lines have blank values for the <client> field (the fourth field). That means that the application that wrote these lines was *not* performing an action on behalf of a client when the message was submitted to the event logger. The line with time stamp 1271721866 is an exception. Here the "anadig" server reported that it turned on digital output channel "v503". The forth field ("WebGUI" in this case) indicates that this change was requested by a client calling itself "WebGUI". Another example of this is the line with time stamp 1271721987. Here a client calling itself "sohMenu" told the "soh" server to close the state-of-health log file. The "soh" server is the one that wrote this line into the event log.

8 IRC Support

The event logger has the optional capability of reporting its messages to a chat channel on an IRC server, in addition to sending them to the current log file(s). This capability is enabled or disabled at *compile time*, by setting the preprocessor macro `IRC` to either a one (IRC capability enabled) or a zero (IRC capability disabled).[5]

Sprinkled throughout the *loggerServer.cpp* source code, you will find code snippets that are conditionally compiled into the final executable based on the value of `IRC`. One such code snippet is shown below.

```
#if IRC > 0
   if (argc != 7) {
      // all IRC command-line parameters are REQUIRED if IRC support is
      // compiled in.  Otherwise, they are allowed to be present on the
      // command line, but they are ignored.
      Usage();
      exit(1);  //--------------------
   }

   ircServer  = argv[3];
   ircPort    = atoi(argv[4]);
   ircChannel = argv[5];
   ircBotName = argv[6];
#endif
```

8.1 Using the `IRCConnection` Utility Class

The event logger uses the `IRCConnection` utility class (defined in *ircLib.h* and implemented in *ircLib.cpp*) for its IRC support. The class allows the event logger to connect to a user-specified IRC server, join a user-specified chat channel, and send log messages to that channel as they become available. The class transparently manages the connection to the IRC server, responding to those IRC server messages it needs to respond to, and automatically reconnecting to the IRC server if the connection is lost for some reason. The `IRCConnection` class is completely described in design document ***DES-0009*, *The IRCConnection Utility Class***.

Like all users of the `IRCConnection` class, the event logger derives a custom class from it to implement the specific capability it needs. The event logger's class definition is shown below.

```
#if IRC > 0
class LoggerIRC: public IRCConnection {
```

[5] The project makefile is set up to expect a value for the `IRC` preprocessor macro to be passed in on the command line to the *make* utility. Compiling the project using "make IRC=1" will automatically define `IRC` as one and pass the definition along to all source modules that require it. By default, IRC support is enabled (i.e. if the value for `IRC` is not explicitly given to *make*, then a value of one will be used). Users need to explicitly disable it ("make IRC=0") to remove IRC support from the compiled version.

```
public:
   LoggerIRC(string botName, string uname, string rname, string chan);
   string channel;

protected:
   void Msg376Handler(PrefixPart &prefix, CommandPart &command, ParamsPart &params);
};
#endif
```

The class is simple, doing nothing other than defining one new IRC server message handler for message type "376" (end of the "message of the day" message). This message gets sent by the IRC server as the last line of its message-of-the day paragraph. The event logger uses receipt of this message to know when it has successfully connected to the server. All the message handler does in response is to send a "JOIN" message back to the server asking to join the chat channel specified by the user when the event logger was started (see section 9).

Once the connection to the IRC server has been established and the appropriate chat channel joined, the event logger uses the base class' `Message()` method to post new log messages to the channel.

8.2 Text Attribute and Color Codes

In the IRC world, there is an unofficial, but very popular and widely adopted, standard for setting the foreground and background colors of the characters that make up channel messages. This is the *mIRC* color scheme, named after the Windows IRC client application that first implemented it.

The *mIRC* standard defines attribute codes and color codes. Attribute codes always appear in pairs, and they surround the text they affect. The attribute codes are unusual in that, although they are added to text strings, the actual numeric representation of the code value is used rather than its text representation. For example, the attribute code for color is 03. When added to the text string, rather than using the text string "03" for the code, its actual one-byte numeric representation (hexadecimal 0x03) is used instead. Color codes, on the other hand, are represented using their text string representation (so the code for "red", which is 04, is actually represented by the text string "04").

The *mIRC* attribute codes are listed in the following table (code values are in decimal).

Table 1: *mIRC* Attribute Codes

Code	**Affect**	**Comments**
02	bold	The enclosed text will be displayed in bold.
03	color	The enclosed text will be displayed using the specified foreground and background colors.
22	italics	The enclosed text will be italicized (some clients respond to this code by swapping the current foreground and background colors instead; so called "reverse video").
31	underlined	The enclosed text will be underlined.

Foreground and background color codes range from 00 to 15 and are defined in the following table.

Table 2: *mIRC* Color Codes

Code	Color	Code	Color
“00”	white	”08”	yellow
“01”	black	”09”	light green
“02”	blue	”10”	teal
“03”	green	”11”	light cyan
“04”	red	”12”	light blue
“05”	brown	”13”	pink
“06”	purple	”14”	grey
“07”	orange	”15”	light grey

Most of the attribute codes take no additional parameters. For example, the following string would display the word “bold” in boldface (items in angle brackets denote non-printable ASCII code characters, which is what all *mIRC* attribute codes are).

```
This is a <0x02>bold<0x02> example!
```

The attribute code for color is slightly different. For the color attribute, a foreground color code and a background color code, separated by a comma, must immediately follow the first color attribute code of the pair. In the following example, the text “red on yellow” would be displayed with red letters on a yellow background.

```
This example contains some <0x03>04,08red on yellow<0x03> text.
```

The event logger uses these *mIRC* codes to color code the message priority tag of the messages it reports to the IRC channel. The code to do this can be found in the `_LogMsg()` method.

8.3 IRC Channel Output Format

Not all of the fields of a log record are sent to the IRC channel, but most of them are. The IRC output consists of one record per line, and each line displays the message priority, the name of the client that is responsible for the request that caused the message to get logged, the name of the server that logged the message, the file, function and line number of the piece of source code that generated the message and the message itself. The priority string is color coded based on priority level. An example snippet of the event logger’s IRC output appears below (without the color coding).

```
<logger> CHANGE  (WebGUI ==> anadig) (optoServer.cpp/SetDigital 711) Digital output
         channel 'v502' changed to OFF
<logger> CHANGE  (WebGUI ==> anadig) (optoServer.cpp/SetDigital 711) Digital output
         channel 'v302' changed to ON
<logger> CHANGE  (WebGUI ==> anadig) (optoServer.cpp/SetDigital 711) Digital output
         channel 'v302' changed to OFF
<logger> HIGHEST ( ==> controlMenu) (controlMenu.cpp/main 412) Application startup
<logger> ERROR   ( ==> mks) (mks910Server.cpp/DataUpdateThread 688) Error reading
         response from 'dps101' while trying to read pirani pressure (Read timed out
         with no data received)
<logger> HIGHEST ( ==> controlMenu) (controlMenu.cpp/main 647) Application shutdown
```

9 Starting the Server

The event log server can be started from the command line directly, but, most often, it is started from the project's *start.sh* script. The server takes either two or six command line parameters, depending on whether or not IRC support has been compiled into the server. Even when IRC support is not enabled, the long version of the command line is still allowed, but the IRC specific parameters are ignored.

```
loggerServer priority numChannels [irc-server irc-port irc-channel irc-name]
```

- `priority` — The initial minimum logging priority (can be changed on the fly by clients). Valid numbers are 0=highest, 1=fatal, 2=error, 3=warning, 4=message, 5=change, and 6=status. Any other value will default to priority 5=change.
- `numChannels` — The number of file channels the server should support. On projects where multi-file support is not needed (see section 3), this should be set to one.
- `irc-server` — Tells the server which IRC server to connect to. This parameter is optional and then only meaningful if IRC support has been compiled into the server.
- `irc-port` — Tells the server which port to connect on when connecting to the IRC server. This parameter is optional and then only meaningful if IRC support has been compiled into the server.
- `irc-channel` — Tells the server which IRC channel to join. This parameter is optional and then only meaningful if IRC support has been compiled into the server.
- `irc-name` — This is the nickname the logger server will use on the IRC server. This parameter is optional and then only meaningful if IRC support has been compiled into the server.

10 The Test Menu Program

For development and testing purposes, a small text-mode menu client application called *loggerMenu* (see *loggerMenu.cpp*), has been developed for the event logger. As expected, the program uses the `LoggerClient` class as its interface to the server.

The menu program is meant to be started from within a terminal window. It requires no command line arguments. When the program runs, it presents the user with the following text menu.

```
General Server Items:
 -1 - ping server
 -2 - get server statistics
 -3 - get server message response interval histogram
-99 - shutdown server

Logger Server Specific Items:
 1 - Send message to log
 2 - Get current logging priority
 3 - Set current logging priority
 4 - Start new log file
 5 - Stop logging
 5 - Stop logging on ALL file channels
 7 - Get file names

0 - Exit Program

Enter Selection >
```

The first three menu items correspond to standard server messages all servers can respond to (see the file *cliserv.h* and design document ***DES-0005, The Client/Server Architecture***). The six options under the "Logger Server Specific Items" heading correspond to the six server-specific messages implemented by the event logger. Users choose the number of the message they want to send, and they are prompted for additional parameters as needed. The selected message is sent to the server and the response is formatted and displayed to the terminal.

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by **Battelle** *Since 1965*

U.S. DEPARTMENT OF
ENERGY

902 Battelle Boulevard
P.O. Box 999
Richland, WA 99352
1-888-375-PNNL (7665)
www.pnnl.gov