Status Server Overview

Tom Vermeulen

26 May 2004

This document is available on the Web at: http://software.cfht.hawaii.edu/statserv/StatusServerOverview/

Contents

1 Purpose

The purpose of this document is to provide a general overview of the Status Server and highlight the utilities available to communicate with the Status Server.

2 Overview and Current Use

The Status Server serves as an open repository of status and state information available to any client within the CFHT network. Clients are able to view, create, update, remove or monitor information within the Status Server. In some respects, the Status Server could be thought of as a shared memory pool for multiple clients.

Some examples of how the Status Server is used include:

- A central staging area for the building of FITS files replacing template files. This allows FITS files to be built in a more parallel fashion without the need for explicit client-side synchronization.
- A replacement for the state information contained within passive text file databases ("par" files. Information contained within "par" files require a relatively inefficient file scan from NFS mounted files.
- A source of information used in GUI status displayed and early warning systems.
- A source of instrument and session related information.
- A source of plant monitoring information.

Figure 1 shows a diagram of the software components which currently provide or use information contained within the Status Server. The low level software components can be broken down by category.

2.1 Agents and Instrument Control

Agents running under director as well as other software components involved in instrument control push information into the Status Server. For the most part, these components don't extract information from the Status Server since they are considered the "master system of record" for this information. The following list contains some of the agents or software components which populate the Status Server. All updates happen within the /i Status Server hierarchy.

- **MegaCam fipcom agent** - This agent communicates with the electronics controlling the filter jukebox, shutter, and cryovessel for MegaPrime. The agent updates both raw data values and state information in the /i/megacam/fipvar location of the Status Server.
- **WIRCam filter wheel agent** - This agent controls the two WIRCam filter wheels via a Galil and updates status in the /i/wircam/status location.
- **WIRCam vacuum agent** - This agent monitors the vacuum pressure of the cryostat via a Granville-Phillips vacuum gauge and updates status in the /i/wircam/status location.
- **WIRCam temperature sensing agent** - This agent monitors temperature sensors located in and around the cryostat for WIRCam. The agent communicates with a Sensoray DIN2419 16 channel temperature monitoring system and updates status in the /i/wircam/status location.
- **WIRCam temperature control agent** - This agent monitors and controls temperatures in the cryostat for WIRCam. Agent communicates with a LakeShore 332 temperature controller and updates status in the /i/wircam/status location.

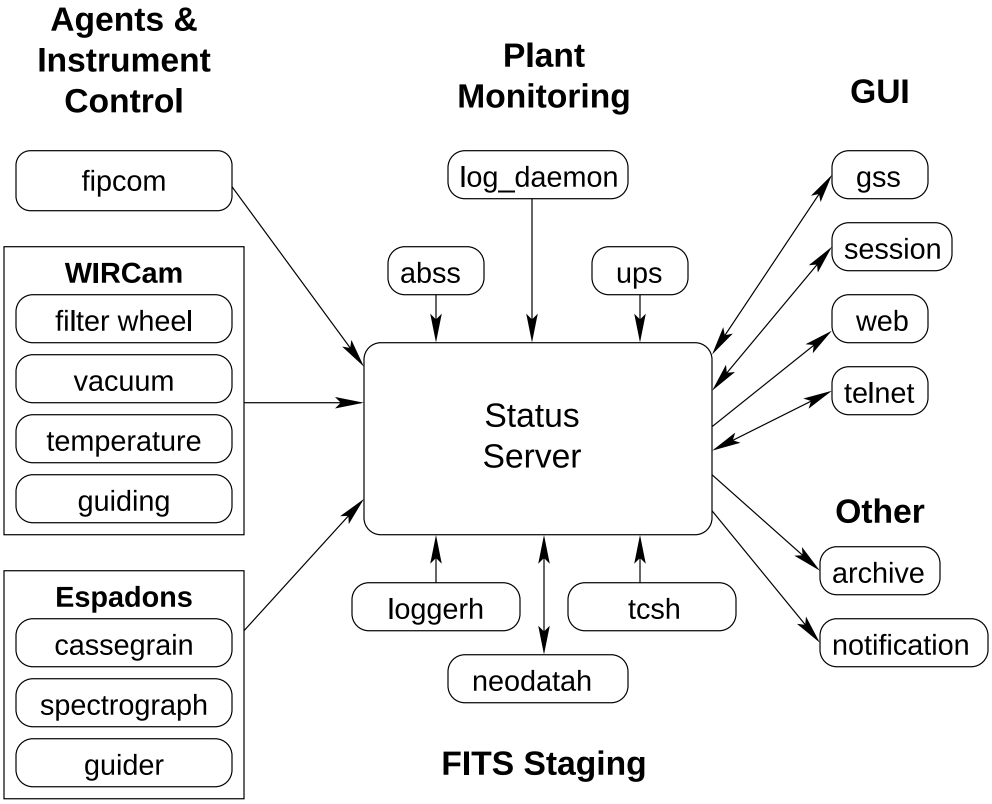

Figure 1: Status Server and Software Component Interactions

- **WIRCam guiding and acquisition system** - How and where information will be updated in the Status Server for the guiding and acquisition system has yet to be defined. It is likely that the aquisition system will update acquisition state information in the Status Server for informational purposes. It may be useful to have the guider update state information in the Status Server as well (not 50Hz data).
- **Espadons cassegrain agent** - This agent monitors and manages the state of the hardware connected to the Espadons Cassegrain unit. This consists primarily of Galil controlled motors, calibration lamps, and temperature sensors. Information is stored in the /i/espadons/cassegrain subdirectory of the Status Server.
- **Espadons spectrograph agent** - This agent monitors and manages the state of the hardware connected to the Espadons Spectrograph unit. This consists primarily of Galil controlled motors and temperature sensors. Information is stored in the /i/espadons/spectrograph subdirectory of Status Server.
- **Espadons guider agent** - This agent monitors and manages the state of guiding for Espadons. Information is stored in the /i/espadons/guider subdirectory of the Status Server.

2.2 Plant Monitoring Information

There are several pieces of plant monitoring information which are populated in the Status Server including probe data from the datalogger, Allen-Bradley PLC information, and UPS status. All updates happen within the /p Status Server hierarchy.

- **abss** - This is a software program which utilizes the Allen-Bradley PLC communication library to extract data from network connected Allen-Bradley PLCs. At this point, the primary use is to monitor data for the

MegaPrime cooling upper end, which is stored in the /p/plc/megaPrimeCoolingUpperEnd subdirectory of the Status Server.

- **log_daemon** - This is a daemon which extracts datalogger probe information and stores it every 10 seconds in the /p/logger subdirectory of the Status Server.
- **ups daemon** - There are two daemons, one at the summit and one in Waimea, which monitor the UPS systems. Data is updated in the /p/ups subdirectory of the Status Server.

2.3 FITS Staging Area

As previously mentioned, the Status Server is used as a central staging area for the consolidation of FITS header information. At this point, there are several Pegasus Session style handlers which have been updated to enable headers to be populated in the staging area. With Espadons and WIRCam, agents will directly update the staging area when called with "beginfits" and "endfits" commands. The staging area for FITS data is contained within the /f Status Server hierarchy.

- **loggerh** - This is the handler which gathers the datalogger information and populates the corresponding headers in the staging area of the Status Server.
- **tcsh** - This is the handler which gathers TCS information and populates the corresponding headers in the staging area of the Status Server.
- **neodatah** - This is a handler which is called before the exposure is taken, and before the other handlers are called, in order to set up the FITS staging area of the Status Server. It is also called after all the handlers are called at the end of an exposure to merge the FITS headers into the FITS file created by DetCom. Finally, it cleans up the FITS staging area associated with the exposure.

2.4 Graphical User Interfaces

Several interfaces have been developed to view and manipulate data within the Status Server.

- **gss** - This is a small gtk-based executable available on all our UNIX systems to view data within the Status Server in a fashion very similar to Windows Explorer. When called with the hidden "-e" option it also allows the ability to create, update, and remove objects and directories in the Status Server.
- **web** - There are several web pages available for viewing data within the Status Server. The RPM web server has the capability to easily set up dynamic forms to view Status Server information. An example of such a form can be found at http://rpm.cfht.hawaii.edu/ billy/megacamcooling.rpm. It is also possible to traverse the Status Server at http://www.cfht.hawaii.edu/cgi-bin/webss.
- **session** - Some instrument sessions have (MegaCam?) or will soon have (Espadons) GUIs which are tied very closely to the Status Server. For example, Espadons has many text fields and icons which are tied to Status Server data and are automatically updated whenever the instrument state data changes.
- **telnet** - It is possible to telnet into the Status Server, in order to view and manage data within it. You can telnet to the Status Server by typing "telnet statserv statserv". If you then type "help" you will see the following list of possible commands. Typing "help ¡command¿" provides detailed help on the underlying command.

2.5 Other Interfaces

Other interfaces are currently in the prototype and planning stages. Now that the Status Server is populated with various bits of status data, there is an interest to archive some of the data for trend analysis and diagnostic purposes. In addition, notification messages need to be generated based on business rules designed to identify fault or warning conditions.

- **archive** - While not complete, a prototype archiving program has been created which takes a set of Status Server objects and deadbands stored in /ss/archive/objects and places monitors on each of these objects. The data is then stored in the file defined at /ss/archive/filename. Before putting a full archiving agent into use, more discussion need to occur regarding requirements and design. The instrumentation group is very interested in a mechanism to view history and plots of Status Server data.
- **notification** - A notification program has been created to send notifications to individuals based on rules defined in the /ss/notication subdirectory of the Status Server. At this point, the rules have been defined and tested to work for the PLC data for the MegaCam cooling system provided by the abss component. This program has been written based on the requirements documented in the "Alert/Notification Requirements" document which can be found on the web at http://software.cfht.hawaii.edu/statserv/notification.

3 Overall Design

This section will focus on some of the overall design aspects of the Status Server which will hopefully help answer some questions as to how it works.

3.1 Directory Structure

Objects within the Status Server are grouped together in a "tree-like" fashion patterned after the UNIX file system. As a result, it is possible for a client to traverse and manipulate objects within the Status Server much like traversing a directory tree and manipulating files in a file system. Objects within the Status Server can be referenced either via a fully qualified directory path/object name combination, or a relative path-name combination. In order to manage relative path references, a current path is maintained for each client connection. Rules to determine whether a path-name combination is expressed as an absolute path or relative path are applied the same way they are in a UNIX file system. A visual example of the type of structure used to hold Status Server information is shown in figure 2. This example is a screen shot taken from gss.

3.2 Directory and Data Object Information

Each directory and data object in the Status Server consists of a series of attributes. These attributes include:

1. **Name** - Within the Status Server the object name, in combination with its associated directory path location, must be unique. The name and directory path must consist of a string of 7 bit ASCII printable characters.
2. **Value** - The value stored with the object. The value of an object stored in the Status Server is always enclosed within double quotes if it is valid. If the value of the object is not valid, it is not enclosed within double quotes. For example, the following values would be considered valid: "data", "0.0", or "sample data". If a value is not enclosed within double quotes, it must always be one of the following values.
   (a) **NONEXISTENT** - If a request is received by the Status Server to initiate a monitor on a data object which does exist, a data object will be created, and its value will be set to NONEXISTENT. For a client perspective, any request made to update or retrieve the value of this object will fail with an indication that the object does not exist, until the object is created with a "touch" command request.

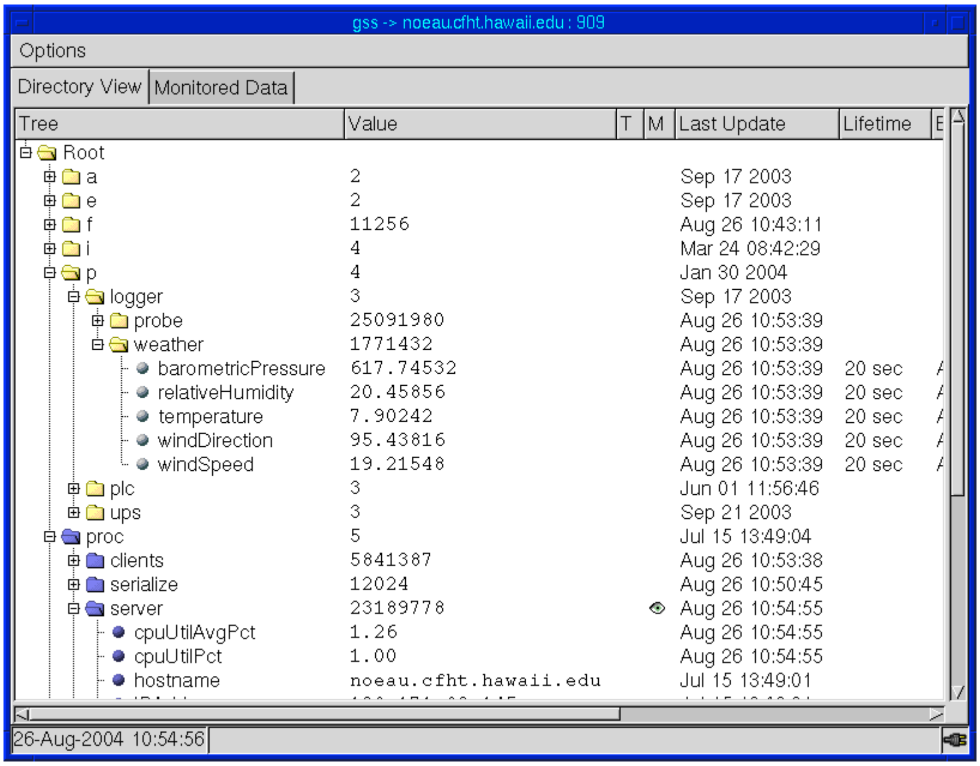

Figure 2: gss - Directory Hierarchy Screen shot

(b) **UNDEFINED** - If a client has created an object, but has not assigned a value. This would be the initial value of an object following a "touch" command request.

(c) **EXPIRED** - If the object has not been updated within a "lifetime" length of time since its last update.

3. **Comment** - An entry describing what the object is.

4. **Lifetime** - Indicates the maximum amount of time this object can be considered valid. As an example, the current seeing may only be defined to be valid for an hour.

5. **Auto-expire** - Indicates that an object will be considered valid as long as the client connection to the Status Server is available. If the connection is broken, the value of the object will change to EXPIRED.

If a data object has a value of NONEXISTENT, it is completely removed and deallocated whenever its use counts are zero. This means that a data object can not be completely removed if a client has performed a touch, monitor, or directory listing request on the object. This is a requirement to enforce pointer integrity within the Status Server.

3.3 Client ⇔ Server Communication

The Status Server listens over a socket interface to client requests. The server services each request and sends back an associated response. With the exception of a disconnect request, each client request will receive a response from the Status Server. In most cases, the client will receive a single line response to a request. The exception to the single response model is the case where a client has requested monitoring updates or the client has requested the contents of a directory. Multiple line response messages will always be terminated with an end-of-transaction (EOT) return message. The client must not send any new commands until it has fully processed the current command. If, for some reason, the server receives a new command request from a client before it has sent the client the last response, it may inform the client that a protocol error has occurred. At this point, the Status Server will expect the client to close the connection. If, however, the client sends another command, the Status Server will close the client connection.

In the case of monitored objects, it is possible for a client to receive an unsolicited message across the interface. This message is triggered the first time a client-monitored object is updated beyond the “deadband” restriction and the client has not already been informed of a monitor update. Once a client is informed that it has monitored information to retrieve, it must initiate a “poll” request to retrieve the information. This is an event-driven model which triggers the client to always initiate a retrieval of monitor information.

The Status Server utilizes the sockio library to handle the low-level socket details. The sockio library uses a single-threaded, non-blocking approach to handling client connections. You can review the the CFHT Socket I/O Library document for more information regarding the design of this library (see http://software.cfht.hawaii.edu/sockio). Figure 3 shows a system diagram illustrating how the sockio library is used as part of both the Status Server and the client C API library.

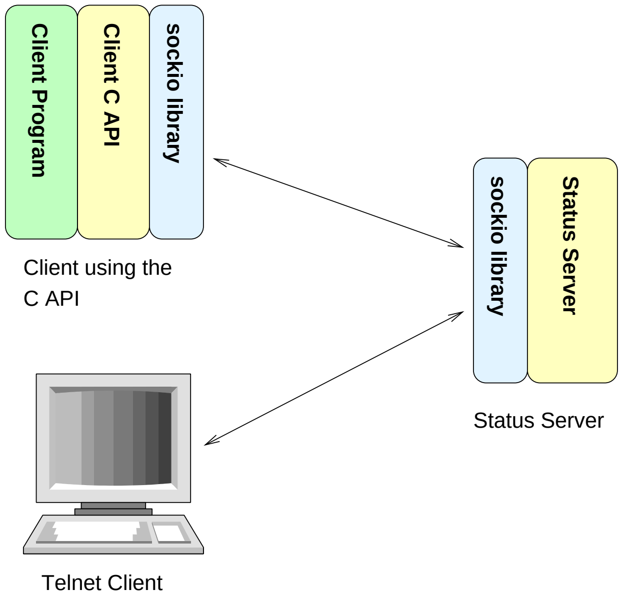

Figure 3: Status Server High Level System Diagram

Both the Status Server and sockio library are designed in such a way that any data sent across the socket can be gracefully handled. This includes receiving binary data or unusually long messages which may or may not be properly terminated with a newline character. If a client attempts to connect from outside the CFHT network, or a client violates the established message protocol, whenever possible its connection will be terminated.

Each request received by the Status Server will be checked to make sure it is both a valid command and does not contain any invalid characters. The Status Server will only process requests which contain URL encoded 7 bit ASCII printable characters terminated with a newline (CR/LF or LF). If a non-conforming request is received, it will be rejected with a “syntax error” response. In the Status Server encoding scheme, only printable characters with the exception of some special characters can be sent unencoded. Figure 4 shows the characters which must be explicitly encoded prior to being received by the Status Server.

URL encoding of a character consists of a “%” symbol, followed by the two-digit hexadecimal representation (case-insensitive) of the character value. For example, a tab character would be encoded as “%09”.

Since the Status Server does not perform any encoding or decoding functionality, functions will be available in the Client API to perform encoding and decoding of Strings from an 8 bit character format to URL encoded format. Clients which decide to access the Status Server via a telnet session or custom socket implementation, must be aware of the URL encoding requirements of the Status Server and perform the necessary encoding.

Character	ASCII Value (Hex)	Reason for Encoding
Percent Sign ('%')	25	Used to URL encode/escape other characters, so it should itself also be encoded.
Single Quote (''')	27	Used as a wrapper around distinct fields of data. The parser will treat data within a single quotes as one field.
Double Quote ('"')	22	Used as a wrapper around distinct fields of data. The parser will treat data within double quotes as one field.
Control Characters	< 20	Must be encoded to prevent unpredictable behavior.
Extended Characters	$> 7E$	Must be encoded to prevent display issues via an interactive telnet session.

Figure 4: Characters Which must be URL Encoded

It is important to note that any encoding schemes used to encapsulate data are completely hidden from clients using the Client API. A client using the Client API does not need to call any encoding or decoding functions.

All string data stored and manipulated within the Status Server is 7 bit only.

3.4 Client C API Primer

The full details of the client C API can be found at http://www.cfht.hawaii.edu/statserv/StatusServer API. If you wish to use the client C API to communicate with the Status Server, it is important to know the prerequisites and scope of the system. The Status Server was designed from the beginning to be a very simple system to meet a specific set of needs. With this in mind, the following list outlines a basic understanding required before the Client C API can be used effectively.

- First and foremost, before using the Status Server API the libsockio, libcli, libss, and libssapi libraries must be installed on the client system.
- The Status Server was designed to hold relatively small pieces of information. There are two important constants defined in ss/ss_define.h which outline the maximum length of both the object name (SS_MAX_NAME_SIZE) and value (SS_MAX_VALUE_SIZE). At the time this document was written, these were both set to 255. It is important to understand that the length of the name is based on a relative path name as opposed to the full absolute path for an object.
- The Status Server was designed to hold information which is updated relatively infrequently. Typically, this means information which updates at around a frequency of one hertz (1x per second or less frequent). While initial benchmarks have shown the ability for the Status Server to handle many thousands of transactions per second, this is a system which contains a large amount of data. As such, it is important common sense is used when evaluating the type of information to be stored in the Status Server.
- Data within the Status Server is kept in memory and serialized at a set time interval. The interval itself can be found at "/proc/serialize/interval" in the Status Server. Typically, this interval is around 5 minutes or so. As a result, it is important to determine how this would affect the information you are planning on putting in the Status Server. In many cases, the serialization interval is relatively unimportant if the information is constantly provided by a persistent client. An example of this is the CFHT weather tower information. This information is constantly updated every 10 seconds by a dedicated client. If the client supplying weather information should go away, the information will expire within 20 seconds. By keeping data in memory and only serializing information periodically, the Status Server can be extremely fast.
- The Status Server client connection is handled very much like NFS. Once a client has successfully logged on to the Status Server, if the Status Server itself goes away, or if a periodic loss of network connectivity occurs, an API function call will block until a connection to the Status Server can be re-established.

- For the most part, clients should not need to worry about where the Status Server is running. The CFHT environment will have one and only one Status Server running on a dedicated port (currently 909) in the CFHT network. If it should become necessary to shut down that Status Server (perhaps machine maintenance) and restart a Status Server on another machine, this can be done almost seamlessly. The Client C API is set up to cycle through a list of potential Status Servers. If connectivity is lost to a given Status Server it will block and continue to cycle through the list until a Status Server can be found. The host list is defined in ss_api.c as STATUS_SERVER_HOST_LIST. Rather than changing this definition, it is also possible to override the list by assigning a new host list to the STATSERV environment variable.
- If connectivity is lost and re-established with a Status Server, the Client C API will make sure that its interaction with the Status Server is unaffected. For example, if the Status Server is shut down on a specific host and a new Status Server is started on a different host, all previously initiated "touches" and "monitors" will be resent and any pending function call will be re-executed.
- Any "out-of-memory" situations detected within the Client C API will result in an abort of the client program utilizing the API. This may seem somewhat extreme. However, given that Linux is the platform of choice for most of the clients at CFHT, this is a situation which can never occur. Starting with the 2.4 Linux Kernel, malloc() will never fail. Instead, if the kernel detects that it is out of memory, it will start killing processes based on a predefined algorithm until memory is available. Given the way Linux is designed and the inherent difficulty in successfully recovering from memory allocation failures, this was a conscious design decision which was made.

3.4.1 Example - Retrieve the value of an object from the Status Server

This is the source code for 'ssGet', a command line utility which retrieves the value of an object from the Status Server. The following would retrieve the current temperature reading from the Status Server.

```
#!/bin/sh
ssGet /p/logger/weather/temperature
```

Here is the source for the ssGet.c program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include "cli/cli.h"
#include "ss/ss_define.h"
#include "ssapi/ss_api.h"
#include "ssapi/ss_error.h"


/* Print a message to stderr indicating what the proper usage syntax is */
static void usageSyntax(void) {
   fprintf(stderr, "usage: ssGet [NAME=]name\n");
}


/* Clean up routine to free up any used memory and exit with a failure */
static void exitWithFailure(command_opt_t* opts) {
   cli_opts_free(opts);
   exit(EXIT_FAILURE);
}
```

```
int
main(int argc, char* const argv[])
{
   char *name = NULL;       /* Location to store the name of the object */
   char *value = NULL;      /* Location to store the value */

   /* Set up the command parsing array for population */
   command_opt_t get_opts[] = {
      { "n*ame", &name, "The name of the object" },
      OPTIONLIST_END
   };

   /* Parse the argument list */
   if (cli_opts(argv + 1, get_opts) != argc - 1) {

      /* Error occured during option parsing */
      usageSyntax();
      exitWithFailure(get_opts);
   }

   /* Make sure that a valid object name was supplied */
   if (!name || !*name) {

      /* Error occured during option parsing */
      usageSyntax();
      exitWithFailure(get_opts);
   }

   /* Allocate space to hold the value returned from the Status Server */
   value = (char *)malloc(SS_MAX_VALUE_SIZE);
   if (value == NULL) {
      fprintf(stderr, "error: memory allocation failed.\n");
      exitWithFailure(get_opts);
   }

   /* Log on to the Status Server */
   if (ssLogon(argv[0]) == FAIL) {
      fprintf(stderr, "Connection failed: %s\n", ssGetStrError());
      free(value);
      exitWithFailure(get_opts);
   }

   /* Retrieve the contents from the Status Server */
   if (ssGetString(name, value, SS_MAX_VALUE_SIZE) == FAIL) {
      fprintf(stderr, "ssGet '%s' failed: %s\n", name, ssGetStrError());
      free(value);
      exitWithFailure(get_opts);
   }
   puts(value);
   free(value);
   cli_opts_free(get_opts);
   exit(EXIT_SUCCESS);
```

```
}
```

4 Document Change Log

Version	Date	Comments
1.0	Aug 26, 2004	First Release.