Status Server Client C API

Tom Vermeulen

6 February 2008

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

Contents

1 Purpose

The purpose of this document is to provide a general overview of the Status Server and outline the C API available to clients which wish to interact with the Status Server.

2 Status Server Overview

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.

2.1 Client $\Leftrightarrow$ Server Communication

The Status Server will listen over a socket interface to client requests. The server will service each request and send back an associated response. While it is possible to interact with the Status Server via a telnet session using its communications protocol, the Client C API is designed to abstract the complexity of the socket interface from the client program. The underlying communications mechanism between the client and server is based on TCP/IP socket communication.

2.2 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 1.

2.3 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.

```
# Status Server database
# ----------------------
#
# Format is /path/and/varname =  # Description in comment#
# Formatting of <sample value> indicates "native" type for this field.
# This is handled internally by the routine that serializes the database.
#
# TRUE/FALSE     - Booleans will have *un-quoted* TRUE or FALSE as the value.
# "string"       - Strings always saved with " as the first char in value field.
# 10.            - Float values will always show a decimal point (even if .0)
# 15             - Numeric values without a decimal indicate integers.
#
# Top-level "directories":
#
# /i/ with a subdirectory for each instrument (often same names as handlers)
# /t/ with subdirectories for each telescope subsystem
# /p/ for plant environment, weather, data-logger variables
# /f/ has subdirectories for each exposure where FITS headers are accumulated
#
/i/                                         # Instruments
/i/megacam/                                 # Megacam agent stuff
/i/megacam/etime        = 10.               # Current exposure time
/i/megacam/etype        = "BIAS"            # Current exposure type
/i/megacam/filter       = 0                 # Current filter position
#
# /i/cfh12k/ are all generated by 12kcom(detcom) and used to be in .,12kcom.par
#
/i/cfh12k/status        = "Idling"          # Camera status for GUI
/i/cfh12k/raster        = "FULL"            # Current raster setting
/i/cfh12k/etime         = 10.               # Current exposure time
/i/cfh12k/etype         = "FLAT"            # Current exposure type
/i/cfh12k/filter        = 0                 # Current filter position
/i/cfh12k/filter[0]     = "R"               # Desc. of filter in slot 0
/i/cfh12k/filter[1]     = "V"               # Desc. of filter in slot 1
/i/cfh12k/filter[2]     = "B"               # Desc. of filter in slot 2
/i/cfh12k/filter[3]     = "I"               # Desc. of filter in slot 3
/i/cfh12k/observer      = "Galileo"         # Current OBSERVER header
/i/cfh12k/object        = "TF dawn"         # Current OBJECT header
/i/cfh12k/comment       = "Twilight flats"  # Current CMMTOBS header
/i/cfh12k/piname        = "Mellier"         # Current PINAME header
/i/cfh12k/runid         = "99IIF142"        # Current RUNID header
```

Figure 1: Status Server Hierarchical Name-Value Pair Representation

2. **Value** - The value stored with the object. The value of an object stored in the Status Server will always be enclosed within double quotes if it is valid. If the value of the object is not valid, it will not be 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 (see section **??**).

   (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 (see section **??**).

   (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 will be 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 API Primer

Before beginning to use the Client C API, 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 inportant 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 serializaing 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 seemlessly. 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.

4 Detailed API

4.1 `ssLogon()` - Establish a connection to the Status Server

Before any other API calls are made to view, create, update, remove, or monitor information within the Status Server, this routine must be called to establish a connection. Once a connection is established to the Status Server, from the client perspective it remains persistent until the client chooses to disconnect from the Status Server.

If the Status Server itself is halted or the network connection between the client and Status Server is broken, the client API will enter into a blocking wait and retry process. As a result, it is not necessary for the client to check whether a connection failure occured following a successful connection. This approach is very similar to an NFS client-server interaction.

Call syntax

```
PASSFAIL ssLogon(const char *name)
```

Input parameters

- `name` - Name of the client program. This should correspond to argv[0] within the main loop of the client.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_SYNTAX_ERROR` - Either the name of the client program is set to NULL, or an invalid hostname:port string is specified in the STATSERV environment variable.
- `SS_NOT_CONNECTED_TO_SERVER` - If the sockio library was not able to establish a connection to the Status Server host.

4.2 `ssLogoff()` - Disconnect from the Status Server

The connection between the client and Status Server will remain persistent until the client calls the ssLogoff() function to disconnect.

```
void ssLogoff(void)
```

4.3 `ssGetError()` - Get the enumerated error value

Whenever an error occured on a Status Server API call, the enumerated value for the error code can be retrieved using this function call. This is very similar to what can be done at a system level with the external integer errno.

Call syntax

```
ss_errno_t ssGetError(void)
```

Return value

This function will return the value of the enumerated type ss_errno_t. The possible enumerated type possibilities are:

- SS_NO_ERROR -
- SS_SYNTAX_ERROR - A request was received by the Status Server which does not have the correct syntax.
- SS_INTERNAL_ERROR - An internal error was detected in the Status Server.
- SS_OBJ_DOES_NOT_EXIST - The object does not exist.
- SS_DIR_DOES_NOT_EXIST - The directory does not exist.
- SS_NODE_DOES_NOT_EXIST - Neither an object or directory exists in the Status Server at the requested location.
- SS_MON_DOES_NOT_EXIST - The requested monitor does not exist.
- SS_TOUCH_DOES_NOT_EXIST - A previous touch by this client does not exist.
- SS_OBJ_ALREADY_EXISTS - An object of the same name already exists.
- SS_DIR_ALREADY_EXISTS - A directory of the same name already exists.
- SS_NO_MONITORS_DEFINED - No monitors are defined.
- SS_DIR_CONTAINS_SUBDIR - The directory contains subdirectories.
- SS_DIR_CONTAINS_HIDDEN_OBJ - The directory contains hidden objects.
- SS_DIR_HAS_DEPENDENCIES - The directory has dependencies.
- SS_INT_CONVERT_ERROR - Unable to convert a string to an Integer.
- SS_DBL_CONVERT_ERROR - Unable to convert a string to a double precision floating point number.
- SS_BOOL_CONVERT_ERROR - Unable to convert a string to a boolean.
- SS_NOT_DEFINED_STATE - An object has not been initialized.
- SS_EXPIRED_STATE - The object has expired.
- SS_NOT_CONNECTED_TO_SERVER - If the sockio library was not able to establish a connection to the Status Server host.
- SS_UNKNOWN_ERROR - An unknown error has been encountered.
- SS_OBJ_HAS_DEPENDENCIES - An object has dependencies.
- SS_NODE_NOT_FOUND - The referenced object or directory can not be found.
- SS_READ_ONLY_MIRROR - Requesting an update to a read-only mirrored Status Server.

4.4 `ssGetStrError()` - Get the enumerated error value

This routine will return back a pointer to a string containing descriptive text of the last error which occured on a Status Server API call. Please see the previous section for more details regarding the possible errors. This is very similar to what can be done at a system level with the call strerror(errno).

Call syntax

```
const char *ssGetStrError(void)
```

Return value

Descriptive string containing error details.

4.5 `ssDisconnectCallback()` - Register a callback function in case of a disconnect

This routine allows a client the ability to register a callback function which will be called whenever the client is disconnected from the Status Server. If the user wishes to remove a previously defined callback routine, this routine can be called with the callback function set to NULL.

It is important to note that this callback function will not be called if a client invokes the ssLogoff() routine.

Call syntax

```
void ssDisconnectCallback(ss_callback_fn callback)
```

Input parameters

- `callback` - Client side callback function to be called in case of a disconnect or NULL to remove a previously defined callback routine. The callback function definition is as follows:

```
typedef void (*ss_callback_fn)(void);
```

4.6 `ssReconnectCallback()` - Register a callback function in case of a re-connect

This routine allows a client the ability to register a callback function which will be called whenever the client is re-connected to the Status Server. If the user wishes to remove a previously defined callback routine, this routine can be called with the callback function set to NULL.

It is important to note that this callback function will not be called if a client invokes the ssLogoff() routine.

Call syntax

```
void ssReconnectCallback(ss_callback_fn callback)
```

Input parameters

- `callback` - Client side callback function to be called in case of a re-connect or NULL to remove a previously defined callback routine. The callback function definition is as follows:

```
typedef void (*ss_callback_fn)(void);
```

4.7 Create an object or register the intent to modify or remove an object

Prior to either updating or removing an object within the Status Server, the client must perform a "touch" on the object. The touch enables a client to specify the intent to update or modify a value in the Status Server. If the data object does not exist before this call is made, it will be created with a state indicating that the object value is undefined.

This routine must also be used if a client wishes to change either the comment associated with an object or the lifetime.

Once a touch is placed on an object, it cannot be completely removed from the Status Server until the use counts for the object are zero. This means that no clients currently have an outstanding touch, monitor, or directory listing on the object. As a result, once a client performs a touch on an object, it is guaranteed to be available for updating.

There are four different API calls which can be used to perform a touch on an object. Each of the different API calls is described in the following sections.

4.7.1 `ssTouchObject()` - Touch object without specifying a lifetime

This function call can be used if you don't wish to modify modify the current lifetime associated with an object. If the object does not yet exist when this call is made, it will be created with an unlimited lifetime.

Call syntax

```
PASSFAIL ssTouchObject(const char *name,
                       const char *comment)
```

Input parameters

- `name` - Name of the data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than the maximum length of SS_MAX_NAME_SIZE defined in ss_define.h.
- `comment` - Description of the data object. If you do not wish to specify a comment, or don't want to modify the current comment, this must be set to NULL.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_DIR_ALREADY_EXISTS` - A directory of the same name already exists.
- `SS_PERMISSION_DENIED` - If an attempt was made to perform a touch on a system object. For example, it is not possible to perform a touch on any objects within the /proc directory path.
- `SS_SYNTAX_ERROR` - The name specified is either NULL, or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.7.2 `ssTouchObjectWithLifetime()` - Touch object with a specified lifetime

This routine performs the same functionality as the ssTouchObject() call with the additional benefit of setting the lifetime of the object.

Call syntax

```
PASSFAIL ssTouchObjectWithLifetime(const char *name,
                                   const char *comment,
                                   time_t lifetime)
```

Input parameters

- `name` - Name of the data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than the maximum length of SS_MAX_NAME_SIZE defined in ss_define.h.
- `comment` - Description of the data object. If you do not wish to specify a comment, or don't want to modify the current comment, this must be set to NULL.
- `lifetime` - An integer number, expressed in seconds, which identifies the amount of time the data object will be considered valid following a modification. If the lifetime is to be unlimited (i.e. no expiration date), this parameter must be set to zero.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_DIR_ALREADY_EXISTS` - A directory of the same name already exists.
- `SS_PERMISSION_DENIED` - If an attempt was made to perform a touch on a system object. For example, it is not possible to perform a touch on any objects within the /proc directory path.
- `SS_SYNTAX_ERROR` - The name specified is either NULL, or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.7.3 `ssTouchObjectWithAutoExpire()` - Touch object with lifetime based on duration of client connection

This function performs a touch on an object with the lifetime based on the duration of the client socket connection. For example, if you use this function and update an object in the Status Server, it will be considered valid until your client connection to the Status Server is broken at which point the object will change state to EXPIRED.

Call syntax

```
PASSFAIL ssTouchObjectWithAutoExpire(const char *name,
                                     const char *comment)
```

Input parameters

- `name` - Name of the data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than the maximum length of SS_MAX_NAME_SIZE defined in ss_define.h.
- `comment` - Description of the data object. If you do not wish to specify a comment, or don't want to modify the current comment, this must be set to NULL.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_DIR_ALREADY_EXISTS` - A directory of the same name already exists.
- `SS_PERMISSION_DENIED` - If an attempt was made to perform a touch on a system object. For example, it is not possible to perform a touch on any objects within the /proc directory path.
- `SS_SYNTAX_ERROR` - The name specified is either NULL, or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.7.4 `ssTouchObjectExpire()` - Touch object and set value to EXPIRED

This function will initiate a touch on an object in the Status Server and immediately change the value to be EXPIRED.

Call syntax

```
PASSFAIL ssTouchObjectExpire(const char *name,
                             const char *comment)
```

Input parameters

- `name` - Name of the data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than the maximum length of SS_MAX_NAME_SIZE defined in ss_define.h.
- `comment` - Description of the data object. If you do not wish to specify a comment, or don't want to modify the current comment, this must be set to NULL.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_DIR_ALREADY_EXISTS` - A directory of the same name already exists.
- `SS_PERMISSION_DENIED` - If an attempt was made to perform a touch on a system object. For example, it is not possible to perform a touch on any objects within the /proc directory path.
- `SS_SYNTAX_ERROR` - The name specified is either NULL, or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.8 `ssTouchdir()` - Create a new directory or register the intent to remove a directory

The call serves two main purposes. First, it allow for a directory to be created if it doesn't already exists. Secondly, it can allow a client to register the intent to remove a directory from the Status Server. If the directory does not exist before this call is made, it will be created with an update count value of "1".

This routine must also be used if a client wishes to change the comment associated with a directory.

Once a touch is placed on the directory, it cannot be completely removed from the Status Server until the use counts for the directory are zero. This means that no clients currently have an outstanding touch, monitor, or directory listing on the directory. As a result, once a client performs a touch on a directory, it is guaranteed to remain persistent until an "untouch" is performed.

Call syntax

```
PASSFAIL ssTouchDir(const char *path,
                    const char *comment)
```

Input parameters

- `path` - Name of the directory path. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.
- `comment` - Description of the directory. If you do not wish to specify a comment, or don't want to modify the current comment, this must be set to NULL.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- SS_OBJ_ALREADY_EXISTS - A data object of the same name already exists.
- SS_PERMISSION_DENIED - If an attempt was made to perform a touch on a system directory. For example, it is not possible to perform a touch on the "/proc" system directory.
- SS_SYNTAX_ERROR - The path specified is either NULL, or invalid.
- SS_NOT_CONNECTED_TO_SERVER - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.9 ssUntouch() - Remove the touch from a Status Server data object or directory

Once a touch is placed on an object or directory, it cannot be completely removed from the Status Server until the use counts for the object or directory are zero. This means that no clients currently have an outstanding touch, monitor, or directory listing on the object or directory. As a result, if a client creates an object or directory by performing a touch or touchdir and another client is responsible for removing the object or directory, the first client must "untouch" the object or directory before it can be completely removed by the second client.

Call syntax

```
PASSFAIL ssUntouch(const char *name)
```

Input parameters

- name - Name of the directory or data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- SS_NODE_DOES_NOT_EXIST - Directory or data object does not exist.
- SS_TOUCH_DOES_NOT_EXIST - A previous touch by this client does not exist.
- SS_SYNTAX_ERROR - The name specified is either NULL, or invalid.
- SS_NOT_CONNECTED_TO_SERVER - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.10 Update an object in the Status Server

Once a successful touch has been performed on an object, it is possible for the client to initiate an update request. The update request causes the value associated with the object to be modified. The client can update a Status Server object with either string, integer, double, or boolean data.

4.10.1 `ssPutString()` - Update with a string value

Call syntax

```
PASSFAIL ssPutString(const char *name,
                     const char *value)
```

Input parameters

- `name` - Name of the data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.
- `value` - New string for the data object. The string must be less than SS_MAX_VALUE_SIZE which is defined in ss_define.h. The string must be NULL terminated and can contain both printable and non-printable 8-bit characters.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_OBJ_DOES_NOT_EXIST` - Data object does not exist.
- `SS_PERMISSION_DENIED` - Either the client did not perform a previous touch on the object, or an attempt was made to update a system object.
- `SS_SYNTAX_ERROR` - Either the name or value specified is either NULL, or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.10.2 `ssPutInt()` - Update with an integer value

Call syntax

```
PASSFAIL ssPutInt(const char *name,
                  long value)
```

Input parameters

- `name` - Name of the data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.
- `value` - New integer value for the data object.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_OBJ_DOES_NOT_EXIST` - Data object does not exist.
- `SS_PERMISSION_DENIED` - Either the client did not perform a previous touch on the object, or an attempt was made to update a system object.
- `SS_SYNTAX_ERROR` - Either the name or value specified is either NULL, or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.10.3 `ssPutDouble()` - Update with a double-precision floating point value

Call syntax

```
PASSFAIL ssPutDouble(const char *name,
                     double value)
```

Input parameters

- `name` - Name of the data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.
- `value` - New double-precision floating point value for the data object.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_OBJ_DOES_NOT_EXIST` - Data object does not exist.
- `SS_PERMISSION_DENIED` - Either the client did not perform a previous touch on the object, or an attempt was made to update a system object.
- `SS_SYNTAX_ERROR` - Either the name or value specified is either NULL, or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.10.4 `ssPutBoolean()` - Update with a boolean value

Call syntax

```
PASSFAIL ssPutBoolean(const char *name,
                      BOOLEAN value)
```

Input parameters

- `name` - Name of the data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.
- `value` - New boolean value for the data object.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_OBJ_DOES_NOT_EXIST` - Data object does not exist.
- `SS_PERMISSION_DENIED` - Either the client did not perform a previous touch on the object, or an attempt was made to update a system object.
- `SS_SYNTAX_ERROR` - Either the name or value specified is either NULL, or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.10.5 `ssPutPrintf()` - Update with a printf style argument

Call syntax

```
PASSFAIL ssPutPrintf(const char *name,
                     const char *fmt, ...)
```

Input parameters

- `name` - Name of the data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.
- `fmt` - Character string that describes the format to be used.
- `...` - Variable number of arguments depending on the number of items being printed.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_OBJ_DOES_NOT_EXIST` - Data object does not exist.
- `SS_PERMISSION_DENIED` - Either the client did not perform a previous touch on the object, or an attempt was made to update a system object.
- `SS_SYNTAX_ERROR` - Either the name or value specified is either NULL, or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.11 Retrieve the value of an object from the Status Server

Since the Status Server is an open repository without permissions, any client has the ability to retrieve an object from the Status Server. Within the Status Server itself, values are stored as strings. The API provides the ability to extract information from the Status Server as either a string, integer, double or boolean value.

4.11.1 `ssGetString()` - Retrieve the value as a string

The information within the Status Server is stored in a modified URL encoded format. This routine will unencode the string and store it within a string specified by the user. The client will receive all the Status Server data up to the maximum length specified by the client. If length of the Status Server string is larger than the amount of room specified by the client, the string will be truncated.

Call syntax

```
PASSFAIL ssGetString(const char *name,
                     char *value,
                     size_t max_len)
```

Input parameters

- `name` - Name of the data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.
- `max_len` - Maximum size of the buffer pre-allocated by the client. This length must include space for the NULL terminating character. If the client specified string length is less than SS_MAX_VALUE_SIZE, it is possible for the client to receive truncated data.

Output parameters

- `value` - Buffer where the contents of the data object will be stored.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_OBJ_DOES_NOT_EXIST` - Data object does not exist.
- `SS_NOT_DEFINED_STATE` - Data object has been created, but has not been initialized with a valid value.
- `SS_EXPIRED_STATE` - Data object exceeded its specified lifetime and is expired.
- `SS_SYNTAX_ERROR` - Either the name or value specified is either NULL, or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.11.2 `ssGetInt()` - Retrieve the value as an integer

Call syntax

```
PASSFAIL ssGetInt(const char *name,
                  long *value)
```

Input parameters

- `name` - Name of the data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.

Output parameters

- `value` - Address where the integer value of the data object will be stored.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_OBJ_DOES_NOT_EXIST` - Data object does not exist.
- `SS_NOT_DEFINED_STATE` - Data object has been created, but has not been initialized with a valid value.
- `SS_EXPIRED_STATE` - Data object exceeded its specified lifetime and is expired.
- `SS_INT_CONVERT_ERROR` - Data object could not be converted to an integer.
- `SS_SYNTAX_ERROR` - Either the name or value specified is either NULL, or invalid.

- SS_NOT_CONNECTED_TO_SERVER - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.11.3 `ssGetDouble()` - Retrieve the value as a double-precision floating point number

Call syntax

```
PASSFAIL ssGetDouble(const char *name,
                     double *value)
```

Input parameters

- `name` - Name of the data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.

Output parameters

- `value` - Address where the double-precision floating point value of the data object will be stored.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- SS_OBJ_DOES_NOT_EXIST - Data object does not exist.
- SS_NOT_DEFINED_STATE - Data object has been created, but has not been initialized with a valid value.
- SS_EXPIRED_STATE - Data object exceeded its specified lifetime and is expired.
- SS_DBL_CONVERT_ERROR - Data object could not be converted to a double-precision floating point value.
- SS_SYNTAX_ERROR - Either the name or value specified is either NULL, or invalid.
- SS_NOT_CONNECTED_TO_SERVER - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.11.4 `ssGetBoolean()` - Retrieve the value as a boolean

Call syntax

```
PASSFAIL ssGetBoolean(const char *name,
                      BOOLEAN *value)
```

Input parameters

- `name` - Name of the data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.

Output parameters

- `value` - Address where the boolean value of the data object will be stored.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_OBJ_DOES_NOT_EXIST` - Data object does not exist.
- `SS_NOT_DEFINED_STATE` - Data object has been created, but has not been initialized with a valid value.
- `SS_EXPIRED_STATE` - Data object exceeded its specified lifetime and is expired.
- `SS_BOOL_CONVERT_ERROR` - Data object could not be converted to a boolean value.
- `SS_SYNTAX_ERROR` - Either the name or value specified is either NULL, or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.12 `ssRemove()` - Remove an object from the Status Server

Once a successful touch has been performed on an object, it is possible for the client to initiate a removal request of the object within the Status Server.

The data object cannot be completely removed from the Status Server until the use counts for the object are zero. This means that no clients currently have an outstanding touch, monitor, or directory listing on the object. If the use counts are greater than zero, the object will be put in a state of "NONEXISTENT" and will be removed completely whenever the use counts do go to zero. In the meantime, any request to "get" or "stat" the object will return an indication that the object does not exist as if it was completely removed from the Status Server.

Call syntax

```
PASSFAIL ssRemove(const char *name)
```

Input parameters

- `name` - Name of the data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_NODE_DOES_NOT_EXIST` - Data object does not exist.
- `SS_PERMISSION_DENIED` - Either the client did not perform a previous touch on the object, or an attempt was made to remove a system object.
- `SS_SYNTAX_ERROR` - The name specified is either NULL or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.13 `ssStat()` - Check for the existence and status of an object

This function will check whether a specific data object exists in the Status Server and if so, return its status.

Call syntax

```
PASSFAIL ssStat(const char *name,
                ss_stat_t *status)
```

Input parameters

- `name` - Name of the data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.

Output parameters

- `status` - Enumerated type indicating whether the data object is valid, undefined, expired, or nonexistent.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_SYNTAX_ERROR` - The name specified is either NULL or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.14 `ssGetAttributes()` - Retrieve the attributes of a Status Server object or directory

This routine will return all the important attributes associated with a data object or directory. The information return is the same information as what is returned as part of a directory listing.

Call syntax

```
const ss_dirent_t *ssGetAttributes(const char *name)
```

Input parameters

- `name` - Name of the directory or data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.

Return value

This function will either return a populated ss_dirent_t data structure, or NULL. If the function should return NULL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_NODE_DOES_NOT_EXIST` - Directory or data object does not exist.
- `SS_SYNTAX_ERROR` - The name specified is either NULL or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.15 Place a monitor on an object

The client has the option of placing a monitor on any object with an optional deadband value. If a deadband are not desired, it can be set to 0. This indicates the client would like to monitor each change to an object value.

Once a monitor is placed on an object, the client must use the ssPoll() or ssWait() routine to retrieve updated monitor information.

Monitor changes to object can then be handled one of two ways. The client can specify a callback function to be called whenever the object changes, or the client can specify memory locations where the object value and value status will be updated. The value status enables the client to know when the state of an object changes, since an object can be valid, expired, undefined, or deleted. When a callback function is used, the updated value is always returned as a string.

4.15.1 `ssMonitorCallback()` - Place a monitor on an object and be informed of the change via a callback

Following an ssPoll() or ssWait() API call, if the value of a data object changes on the Status Server beyond the deadband threshold, the new value and status will be passed to a user callback function, along with the arbitrary 'userptr' provided by the user.

Call syntax

```
PASSFAIL ssMonitorCallback(const char *name,
                           double deadband,
                           ssMonitorCallbackFunc userfunc,
                           void *userptr)
```

Input parameters

- `name` - Name of the data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.
- `deadband` - Amount the Status Server data object must change by before the client is informed of a change in value. If this is set to 0, the client will be informed of all possible changes.
- `userfunc` - This is the client side function which will be called whenever an update of the monitored object occurs. The callback function definition is as follows:

  ```
  typedef void (*ssMonitorCallbackFunc)(void *userptr,
                                        const char *new_value,
                                        ss_stat_t value_status)
  ```

  The value_status is an enumerated type which indicates the status of the value returned. Possible values include SS_VALID, SS_NONEXISTENT, SS_NOTDEFINED, and SS_EXPIRED and SS_TYPE_CAST_EXCEPTION. The new_value field should only be used if the value_status has a state of SS_VALID. If the new_value is considered valid, you must make a copy of the returned string value if you would like the value to remain persistent in the client. In this case, the client should never receive an SS_TYPE_CAST_EXCEPTION since the value is stored as a string in the Status Server, so a type cast is not being performed.
- `userptr` - A pointer provided by the user and passed back as a handle to the callback function. If this is not used by the client, it can be set to NULL.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_OBJ_DOES_NOT_EXIST` - Data object does not exist.
- `SS_SYNTAX_ERROR` - The name specified is either NULL or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.15.2 `ssMonitorString()` - Place a monitor on an object and retrieve updates as a string

Following an ssPoll() or ssWait() API call, if the value of a data object changes on the Status Server, a new string value will be stored in the string buffer specified by the user. The string buffer specified by the client will be populated with the Status Server data up to the maximum length specified by the client. If length of the Status Server string is larger than the amount of room specified by the client, the string will be truncated.

Call syntax

```
PASSFAIL ssMonitorString(const char *name,
                         size_t max_len,
                         char *value,
                         ss_stat_t *value_status)
```

Input parameters

- `name` - Name of the data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.
- `max_len` - Maximum size of the buffer pre-allocated by the client. This length must include space for the NULL terminating character. If the client specified string length is less than SS_MAX_VALUE_SIZE, it is possible for the client to receive truncated data.

Output parameters

- `value` - Buffer where the contents of the data object will be stored.
- `value_status` - Enumerated type which indicates the status of the value returned. Possible values include SS_VALID, SS_NONEXISTENT, SS_NOTDEFINED, SS_EXPIRED, and SS_TYPE_CAST_EXCEPTION. The value field should only be used if the value_status has a state of SS_VALID. In this case, the client should never receive an SS_TYPE_CAST_EXCEPTION since the value is stored as a string in the Status Server, so a type cast is not being performed.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_OBJ_DOES_NOT_EXIST` - Data object does not exist.
- `SS_SYNTAX_ERROR` - The name specified is either NULL or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.15.3 `ssMonitorInt()` - Place a monitor on an object and retrieve updates as an integer

Following an ssPoll() API call, if the value of a data object changes on the Status Server beyond the deadband threshold specified, a new integer value will be stored at the integer memory address specified by the user.

Call syntax

```
PASSFAIL ssMonitorInt(const char *name,
                      long deadband,
                      long *value,
                      ss_stat_t *value_status)
```

Input parameters

- `name` - Name of the data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.
- `deadband` - Amount the Status Server data object must change by before the client is informed of a change in value. If this is set to 0, the client will be informed of all possible changes.

Output parameters

- `value` - Address where the integer value of the data object will be stored.
- `value_status` - Enumerated type which indicates the status of the value returned. Possible values include SS_VALID, SS_NONEXISTENT, SS_NOTDEFINED, SS_EXPIRED, and SS_TYPE_CAST_EXCEPTION. The value field should only be used if the value_status has a state of SS_VALID. It is possible for the client to receive a status of SS_TYPE_CAST_EXCEPTION if it is not possible to convert the string stored on the Status Server to an integer.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_OBJ_DOES_NOT_EXIST` - Data object does not exist.
- `SS_SYNTAX_ERROR` - The name specified is either NULL or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.15.4 `ssMonitorDouble()` - Place a monitor on an object and retrieve updates as a double-precision floating point number

Following an ssPoll() API call, if the value of a data object changes on the Status Server beyond the deadband threshold specified, a new double-precision floating point number value will be stored at the memory address specified by the user.

Call syntax

```
PASSFAIL ssMonitorDouble(const char *name,
                         double deadband,
                         double *value,
                         ss_stat_t *value_status)
```

Input parameters

- `name` - Name of the data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.
- `deadband` - Amount the Status Server data object must change by before the client is informed of a change in value. If this is set to 0, the client will be informed of all possible changes.

Output parameters

- `value` - Address where the integer value of the data object will be stored.
- `value_status` - Enumerated type which indicates the status of the value returned. Possible values include SS_VALID, SS_NONEXISTENT, SS_NOTDEFINED, SS_EXPIRED, and SS_TYPE_CAST_EXCEPTION. The value field should only be used if the value_status has a state of SS_VALID. It is possible for the client to receive a status of SS_TYPE_CAST_EXCEPTION if it is not possible to convert the string stored on the Status Server to a double-precision floating point number.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_OBJ_DOES_NOT_EXIST` - Data object does not exist.
- `SS_SYNTAX_ERROR` - The name specified is either NULL or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.15.5 `ssMonitorBoolean()` - Place a monitor on an object and retrieve updates as a boolean

Following an ssPoll() API call, if the value of a data object changes on the Status Server, a new boolean value will be stored at the memory address specified by the user.

Call syntax

```
PASSFAIL ssMonitorBoolean(const char *name,
                          BOOLEAN *value,
                          ss_stat_t *value_status)
```

Input parameters

- `name` - Name of the data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.
- `deadband` - Amount the Status Server data object must change by before the client is informed of a change in value. If this is set to 0, the client will be informed of all possible changes.

Output parameters

- `value` - Address where the integer value of the data object will be stored.
- `value_status` - Enumerated type which indicates the status of the value returned. Possible values include SS_VALID, SS_NONEXISTENT, SS_NOTDEFINED, SS_EXPIRED, and SS_TYPE_CAST_EXCEPTION. The value field should only be used if the value_status has a state of SS_VALID. It is possible for the client to receive a status of SS_TYPE_CAST_EXCEPTION if it is not possible to convert the string stored on the Status Server to a boolean.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_OBJ_DOES_NOT_EXIST` - Data object does not exist.
- `SS_SYNTAX_ERROR` - The name specified is either NULL or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.16 Place a monitor on a directory

The client has the option of placing a monitor on any directory. Within the Status Server, an update counter is maintained with each directory. This counter is incremented any time the value of a data object within that directory is modified. As a result, a client could place a monitor on a directory and perform a directory listing to get the new values of a directory following a change. One potential application where this could be useful is if a directory is used to store the values within a GUI drop-down list. If a new data object is added or removed, the client could place a monitor on the directory and update the possible options within the drop-down list dynamically.

It is possible to specify a deadband in order to limit the notifications of directory updates. In most cases, it will probably be desirable to be notified of all directory updates, in which case the deadband value must be set to 0.

Once a monitor is placed on a directory, the client must use the ssPoll() or ssWait() routine to retrieve updated monitor information.

Monitor changes to a directory can be handled one of two ways. The client can specify a callback function to be called whenever the contents of a directory change, or the client can specify memory locations where the directory count and value status will be updated. The value status enables the client to know when the state of a directory changes, since a directory can be either be valid or deleted. When a callback function is used, the updated value is always returned as a string

4.16.1 `ssMonitorDirCallback()` - Place a monitor on a directory and be informed of the change via a callback

Following an ssPoll() or ssWait() API call, if the contents of a directory change, the new value and status will be passed to a user callback function, along with the arbitrary 'userptr' provided by the user.

Call syntax

```
PASSFAIL ssMonitorDirCallback(const char *path,
                              ssMonitorCallbackFunc userfunc,
                              void *userptr)
```

Input parameters

- `path` - Name of the directory path. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.
- `userfunc` - This is the client side function which will be called whenever an change to the monitored directory occurs. The callback function definition is as follows:

  ```
  typedef void (*ssMonitorCallbackFunc)(void *userptr,
                                        const char *new_value,
                                        ss_stat_t value_status)
  ```

  The value_status is an enumerated type which indicates the status of the value returned. Since it is not possible for a directory to become expired or undefined, the only possible values the client should see are SS_VALID or SS_NONEXISTENT. If the new_value is considered valid, you must make a copy of the returned string value if you would like the value to remain persistent in the client. This value can be converted to an integer, since the Status Server manages directory counts an incrementing integer value.
- `userptr` - A pointer provided by the user and passed back as a handle to the callback function. If this is not used by the client, it can be set to NULL.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_DIR_DOES_NOT_EXIST` - Directory does not exist.
- `SS_SYNTAX_ERROR` - The name specified is either NULL or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.16.2 `ssMonitorDir()` - Place a monitor on a directory

Following an ssPoll() or ssWait() API call, if the contents of a directory change, the client can specify a memory location to hold the new directory count and value status.

Call syntax

```
PASSFAIL ssMonitorDir(const char *path,
                      long *value,
                      ss_stat_t *status)
```

Input parameters

- `path` - Name of the directory path. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.

Output parameters

- `value` - Address where the update count value of the directory will be stored.
- `value_status` - Enumerated type which indicates the status of the value returned. Since it is not possible for a directory to become expired or undefined, the only possible values the client should see are SS_VALID or SS_NONEXISTENT.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_DIR_DOES_NOT_EXIST` - Directory does not exist.
- `SS_SYNTAX_ERROR` - The name specified is either NULL or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.17 `ssUnmonitor()` - Remove the monitor from a data object or directory

This routine will remove a monitor from a data object or directory currently monitored by the client.

Call syntax

```
PASSFAIL ssUnmonitor(const char *name)
```

Input parameters

- `name` - Name of the directory or data object. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_NODE_DOES_NOT_EXIST` - Directory or data object does not exist.
- `SS_MON_DOES_NOT_EXIST` - The client has not previously placed a monitor on this directory or data object.
- `SS_SYNTAX_ERROR` - The name specified is either NULL, or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.18 `ssPoll()` - Check for and, if available, retrieve monitor updates

This routine will initiate a check whether any monitored information is available for retrieval by the C API and, if so, retrieve the monitored data from the Status Server and either call the client registered callback function or store the information in the memory locations whhich were previously defined during the setup of the monitor.

If a monitor request was made to monitor as an Integer, floating point, or boolean data types, a conversion will be performed from the string format received over the interface to the monitor requested data type. Any errors detected, either during the conversion process or from the data response sent by the Status Server, will be stored in the previously allocated memory location for return code information.

Call syntax

```
PASSFAIL ssPoll(void)
```

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.19 `ssWait()` - Wait for a period and, if available, retrieve monitor updates

This routine is identical to the ssPoll() except it allow for a timeout to be supplied which indicates how long the timeout should be on a select() when waiting for activity to occur on a socket. If activity occurs on the socket, it will initiate a check whether any monitored information is available for retrieval by the C API and, if so, retrieve the monitored data from the Status Server and either call the client registered callback function or store the information in the memory locations whhich were previously defined during the setup of the monitor.

If a monitor request was made to monitor as an Integer, floating point, or boolean data types, a conversion will be performed from the string format received over the interface to the monitor requested data type. Any errors detected, either during the conversion process or from the data response sent by the Status Server, will be stored in the previously allocated memory location for return code information.

Call syntax

```
PASSFAIL ssWait(int timeout_hundredths)
```

Input parameters

- `timeout_hundredths` - Time in hundredths of a second to wait for activity on the socket.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.20 `ssSync()` - Force client to synchronize itself with the server

This function will force mailbox retrieval message to be sent to the Status Server and will wait for a response from the Status Server. It is useful to call this after setting up a set of new monitors. If monitored information is available for retrieval, it will retrieve the monitored data from the Status Server and either call the client registered callback function or store the information in the memory locations whhich were previously defined during the setup of the monitor.

If a monitor request was made to monitor as an Integer, floating point, or boolean data types, a conversion will be performed from the string format received over the interface to the monitor requested data type. Any errors detected, either during the conversion process or from the data response sent by the Status Server, will be stored in the previously allocated memory location for return code information.

Call syntax

```
PASSFAIL ssSync(void)
```

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.21 `ssPwd()` - Retrieve the current working directory path

This routine will retrieve the current working directory path for the client from the Status Server. The default path used by a client upon connection to the Status Server is "/". This will be the path returned by this routine unless the client has explicitly change the working path through a call to ssChdir().

Call syntax

```
const char *ssPwd(void)
```

Return value

This function will either return the current working directory path or NULL in case of an error. If the function should return NULL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.22 `ssChdir()` - Change the current working directory path

This routine will change the current working directory for the client on the Status Server. The default path used by a client upon connection to the Status Server is "/".

Call syntax

```
PASSFAIL ssChdir(const char *path)
```

- `path` - Name of the directory path. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- SS_DIR_DOES_NOT_EXIST - Directory does not exist.
- SS_SYNTAX_ERROR - The path specified is either NULL, or invalid.
- SS_NOT_CONNECTED_TO_SERVER - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.23 ssRmdir() - Remove a directory from the Status Server

Once a successful touch has been performed on a directory, it is possible for the client to request the removal of the directory from the Status Server.

The directory cannot be completely removed from the Status Server until the use counts for the directory are zero. This means that no clients currently have an outstanding touch, monitor, or directory listing on the directory. If the use counts are greater than zero, the object will be put in a state of "NONEXISTENT" and will be removed whenever the use counts become zero.

Call syntax

```
PASSFAIL ssRmdir(const char *path)
```

Input parameters

- path - Name of the directory path. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- SS_NODE_DOES_NOT_EXIST - Directory does not exist.
- SS_PERMISSION_DENIED - Either the client did not perform a previous touch on the directory, or an attempt was made to remove a system directory.
- SS_SYNTAX_ERROR - The path specified is either NULL or invalid.
- SS_NOT_CONNECTED_TO_SERVER - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.24 `ssOpenDir()` - Open a Status Server directory for reading

When performing a directory listing, a Status Server directory must first be opened for reading. If the directory can be successfully opened, all the underlying contents can be retrieved by making successive calls to ssReaddir. Information from multiple directories can be retrieved by using the same regular expression matching rules used by the UNIX "ls" command.

Call syntax

```
PASSFAIL ssOpenDir(const char *path)
```

Input parameters

- `path` - Name of the directory path. The name can be specified with a fully-qualified absolute directory path, or as a name containing a relative directory path. Absolute references must be prefixed by a leading '/'. Relative path references may be prefixed with either a "./" or "../". Please note that upon the initial connection to the Status Server, the current path for the client will be "/". The name must be less than SS_MAX_NAME_SIZE which is defined in ss_define.h. In addition, the same regular expression matching rules used by the UNIX "ls" command can be included in the directory path.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_NODE_DOES_NOT_EXIST` - Directory does not exist.
- `SS_SYNTAX_ERROR` - The path specified is either NULL or invalid.
- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.25 `ssReadDir()` - Retrieve the next directory listing object

This routine will retrieve the next directory line if a directory has previously been opened for reading. Upon successful completion a populated ss_dirent_t will be returned back to the client.

Call syntax

```
const ss_dirent_t *ssReadDir(void)
```

Return value

This function will either return a populated ss_dirent_t data structure, or NULL. If the function should return NULL, there are two reasons why this could occur. First, there are no more directory lines to be read. Secondly, an error occured. If the function should return NULL, the ssGetError() function must be called to determine if an error occured. If ssGetError() returns SS_NO_ERROR, the directory contents specified in the ssOpendir() function have all been retrieved. If ssGetError() is not SS_NO_ERROR, these are the possible error conditions why this routine may fail:

- `SS_DIR_DOES_NOT_EXIST` - This call was made before having called the ssOpendir() routine.

4.26 `ssGetFileDescriptor()` - Retrieve the socket file descriptor used to communicate with the Status Server

Call syntax

This routine will return back the file descriptor used for socket communication with the Status Server. As a result, the client could use the file descriptor within a select loop in order to be informed when the Status Server sends something to the client. This is particularly useful for a client which has placed monitors on Status Server variables.

Caution must be used when using the file descriptor. If it is used incorrectly, the socket communications between the client and Status Server may be adversely affected.

```
PASSFAIL ssGetFileDescriptor(int *fd)
```

Output parameters

- `value` - Address where the file descriptor value will be stored.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.27 `ssTraceOn()` - Turn on tracing within the Status Server

This routine turns on the tracing of all Status Server activity on the server itself. Please be `VERY CAREFUL` when using this option, since tracing will seriously affect the performance of the Status Server and could flood the CFHT_LOG.

Also, once tracing is turned off, it will stay on until it is turned off with the ssTraceOff() function call. It will not be turned off automatically when the client who requested it disconnects from the Status Server.

Call syntax

```
PASSFAIL ssTraceOn(void)
```

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- SS_NOT_CONNECTED_TO_SERVER - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.28 `ssTraceOff()` - Turn off tracing within the Status Server

This routine turns off the tracing of all Status Server activity on the server itself. The client which initiated the trace with the ssTraceOn function call does not need to be the client which requests the tracing to be turned off.

Call syntax

```
PASSFAIL ssTraceOff(void)
```

Return value

This routine will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- SS_NOT_CONNECTED_TO_SERVER - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.29 `ssAutosave()` - Initiate a serialization of the Status Server

This routine triggers a serialization of the Status Server data contents to a text file. The location of where the serialization text file is stored can be found by looking at the value of "/proc/serialize/filename". A serialization of the Status Server is also performed periodically at the interval specified by "/proc/serialize/interval".

Call syntax

```
PASSFAIL ssAutosave(void)
```

Return value

This routine will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- SS_NOT_CONNECTED_TO_SERVER - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.30 `ssWait()` - Wait for a period and, if available, retrieve mirror updates

This routine allows for a timeout to be supplied which indicates how long the timeout interval should be on a select() call when waiting for activity to occur on a socket. If activity occurs on the socket, it will initiate a check whether any mirroring information is available for retrieval by the C API. If so, the mirroring data will be retrieved and the client registered callback function will be invoked.

Call syntax

```
PASSFAIL ssMirrorWait(int timeout_hundredths)
```

- `timeout_hundredths` - Time in hundredths of a second to wait for activity on the socket.

Return value

This function will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

4.30.1 `ssMirrorCallback()` - Mirror a Status Server and be informed of any changes via a callback

Following an ssMirrorWait() API call, if an object or directory changes on the Status Server, a mirror update transaction will be passed to a user callback function.

Call syntax

```
PASSFAIL ssMirrorCallback(ssMonitorCallbackFunc userfunc),
```

Input parameters

- `userfunc` - This is the client side function which will be called whenever a mirror update transaction is received. The callback function definition is as follows:

  ```
  typedef void (*ssMirrorCallbackFunc)(const char *value)
  ```

  The value field contains a string with the mirror update transaction.

Return value

This function should always return PASS when called via the client API. If a failure should happen to be returned, this should be considered an internal error within the Status Server API library. At this point, the client API library will initiate an abort() causing a segmentation fault to occur.

4.31 `ssShutdown()` - Shutdown the Status Server

This routine will initiate a serialization of the Status Server data followed by a shutdown of the server itself. This is a disruptive operation and will affect all clients currently connected to the Status Server. As a result, the utmost discretion must be used to determine when the routine should be used.

Call syntax

```
PASSFAIL ssShutdown(void)
```

Return value

This routine will return either PASS or FAIL. If the function should return FAIL, the reason for the error will be an enumerated type returned by the ssGetError() function. The error types which are possible when calling this routine are:

- `SS_NOT_CONNECTED_TO_SERVER` - This error should only if the client has never called ssLogon() or if the initial ssLogon() failed. Otherwise, a retry mechanism is in place to handle a temporary loss of connection. As a result, if a client has successfully logged on to the Status Server, it should not need to check for this return value.

5 Examples

5.1 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);
}
```

5.2 Update the value of a Status Server object

This is the source code for 'ssPut', a command line utility which update the value of a Status Server object. The following would update the value of "/test/testval1" to be "TEST VALUE" with a lifetime of 5 seconds. In this case, if the object does not already exist, it would be created prior to being updated.

```
#!/bin/sh
ssPut /test/testval1 "TEST VALUE" comment="test update" lifetime=5
```

Here is the source for the ssPut.c program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <limits.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: ssPut [NAME=]name [VALUE=]value "
   "[COMMENT=comment] [LIFETIME=lifetime]\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 *comment = NULL;   /* Location to store the comment */
   char *lifetime = NULL;  /* Location to store the lifetime value */
   char *value = NULL;     /* Location to store the value */
   PASSFAIL rc;            /* Return code value */
```

```
   time_t lifetime_val = 0;      /* lifetime (0=unlimited) */

   /* Set up the command parsing array for population */
   command_opt_t put_opts[] = {
      { "n*ame",     &name,     "Name of the object"                },
      { "v*alue",    &value,    "Value of the object"               },
      OPTIONLIST_NONPOSITIONAL,
      { "c*omment",  &comment,   "Descriptive comment of the object" },
      { "l*ifetime", &lifetime,  "Time(seconds) this object is valid"},
      OPTIONLIST_END
   };

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

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

   /* Make sure that at a minimum, a valid object name, and object
    * value were supplied */
   if (!name || !*name || !value) {

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

   /* Check to make sure that the value specified by the client
    * not beyond what the Status Server can store */
   if (strlen(value) > SS_MAX_VALUE_SIZE) {
      fprintf(stderr, "error: value to large to store in the "
      "Status Server");
      exitWithFailure(put_opts);
   }

   /* Make sure if a lifetime parameter was supplied that it is valid */
   if (lifetime && *lifetime) {

      /* Parse the lifetime option */
      char *stop_at = NULL;     /* Location at which strtol may stop */

      lifetime_val = strtol(lifetime, &stop_at, 10);

      if (lifetime_val < 0 || *stop_at != '\0' ||
 ((lifetime_val == LONG_MIN || lifetime_val == LONG_MAX) &&
  (errno == ERANGE)) ||
 ((lifetime_val == 0) && (errno == EINVAL))) {

/* lifetime value is invalid */
fprintf(stderr, "error: lifetime parameter supplied is invalid");
exitWithFailure(put_opts);
```

```
      }
   }

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

   /* Perform a touch of the object in the Status Server */
   if (lifetime && *lifetime)
      rc = ssTouchObjectWithLifetime(name, comment, lifetime_val);
   else
      rc = ssTouchObject(name, comment);

   if (rc == FAIL) {
      fprintf(stderr, "Put failed: %s\n", ssGetStrError());
      exitWithFailure(put_opts);
   }

   /* Perform an update of the object in the Status Server */
   if (ssPutString(name, value) == FAIL) {
      fprintf(stderr, "ssPut failed: %s\n", ssGetStrError());
      exitWithFailure(put_opts);
   }
   cli_opts_free(put_opts);
   exit(EXIT_SUCCESS);
}
```

5.3 Monitor a Status Server object

This is the source code for 'ssMonitor', a command line utility which will place monitor on a Status Server object and exit with a new value once it is different than the value specified on the command line.

```
#!/bin/sh
ssMonitor /test/testval1 "OLD VALUE"
```

Here is the source for the ssMonitor.c program:

```
#ifdef VXWORKS
#include <selectLib.h>
#else
#include <sys/time.h>
#include <sys/types.h>
#endif
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.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: ssMonitor [NAME=]name [VALUE=]value\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 */
   char *value = NULL;           /* object value for comparison */
   char *ss_value = NULL;        /* value received from the Status Server */
   ss_stat_t status = SS_VALID;  /* object status */

   /* Set up the command parsing array for population */
   command_opt_t mon_opts[] = {
      { "n*ame", &name,   "Name of the object"        },
      { "v*alue", &value, "Value used for comparison" },
      OPTIONLIST_END
   };

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

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

   /* Make sure that a valid object name and comparison value were supplied */
   if (!name || !*name || !value) {

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

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

```
  }

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

  /* Place a monitor on the Status Server variable */
  if (ssMonitorString(name, SS_MAX_VALUE_SIZE,
      ss_value, &status) == FAIL) {
     fprintf(stderr, "ssMonitor failed: %s\n", ssGetStrError());
     free(ss_value);
     exitWithFailure(mon_opts);
  }

  /* Go in to a loop surrounding ssWait() and break out of the routine
   * when the retrieved value following a monitor update is different
   * than the specified value */
  for (;;) {

     /* Initiate a poll request to retrieve monitor information.  The
      * ssWait() routine with a parameter of -1 will cause the wait
      * to block indefinitely until data is received on the socket */
     if (ssWait(-1) == FAIL) {
fprintf(stderr, "Poll failed: %s\n", ssGetStrError());
free(ss_value);
exitWithFailure(mon_opts);
     }

     /* Check whether the return value is valid.  If so, echo the
      * new monitored value to stdout.  Otherwise, continue back in
      * the select loop. */
     if (status == SS_VALID) {
if (strcmp(ss_value, value) != 0) {
   puts(ss_value);
   free(ss_value);
   cli_opts_free(mon_opts);
   exit(EXIT_SUCCESS);
}
     }
     else {
if (status == SS_NONEXISTENT) {
   fprintf(stderr, "Monitored object does not exist\n");
} else if (status == SS_NOTDEFINED) {
   fprintf(stderr, "Monitored object is not defined\n");
} else if (status == SS_EXPIRED) {
   fprintf(stderr, "Monitored object is expired\n");
}
else {
   fprintf(stderr, "Invalid monitor object status\n");
}
```

```
 free(ss_value);
 exitWithFailure(mon_opts);
      }
   }
}
```

6 Document Change Log

Version	Date	Comments
0.1	Dec 6, 2002	Preliminary Release.
0.2	Jan 31, 2003	Added ssTouchObjectWithLifetime(). Minor modifications.
0.3	Mar 27, 2003	Added a brief API overview and some usage examples.
1.0	Sept 18, 2003	Several changes to the API for the 1.0 release.
1.1	May 19, 2004	Added the ability to monitor a Status Server. API modifications to monitoring functions.
1.2	February 6, 2007	Documented several missing functions.