Status Server Functional Specification

Tom Vermeulen

24 May 2002

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

Contents

List of Figures

1 Introduction

1.1 Purpose

The purpose of this document is to define the functional specification for the Status Server. The functional specification documented here is based on a set of previously defined requirements. The full set of requirements will not be repeated in this document, so please read the "Status Server Requirements" document.

The goal of the Status Server is to serve as an open repository of status and state information easily available to any client within the CFHT network. Clients will be able to view, update, and place monitors on data elements within the server. One key area where the Status Server will be used is as the staging area for the building of FITS files, replacing the current template files.

There are essentially two components which make up the system referred to as the "Status Server". The main component is the Server itself, which contains the information stored by clients. The second component is the client. The "Status Server" project requires that a robust API library be available for clients to interact with the server. It is the responsibility of the client API library to hide the details of the socket protocol used to create, update, and retrieve Status Server information. This document will outline a proposed client-side C API and discuss the underlying functionality the API library and Status Server will support in order to service the client-side requests.

The first draft of this document must be reviewed by the members of the software group and will be amended following review.

1.2 Scope

Unless otherwise noted, the functional specifications identified in this document are intended to be implemented in the first release of the Status Server. However, release requirements may dictate the priority and staging of functionality.

2 Functional Specifications

2.1 Client access to the Status Server

There are essentially three ways which a client can access the Status Server. They are as follows:

- **Via a C-API** - This is the preferred access method for clients to access the Status Server. Within this document, a proposed API is provided for each access function.
- **Via UNIX Shell Scripts** - Shell Scripts will be generated by placing a wrapper around the C-API calls, so in effect the functionality provided by the Shell Scripts will be facilitated via C-API calls.
- **Via a telnet session** - The underlying communication mechanism used by the C-API to communicate with the Status Server is via a socket protocol. Is is possible for clients to utilize the same protocol and communicate to the Status Server via a telnet session.

2.2 Status Server Object Organization

2.2.1 Directory Structure

Objects within the Status Server are grouped together in a tree-like fashion patterned after the UNIX file system. As a result, it will be 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 will be maintained for each client connection. Rules to determine whether a path-name combination is expressed as an absolute path or relative path will be 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.

```
# 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

2.2.2 Object Information

Each 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. In addition, the following series of characters will not be permitted in a directory path or object name (“, ’, =, space).

2. **Value** - The value stored with the object. This value will be in one of the following states:

(a) **Undefined** - The object was created, but the value has not been set.
(b) **Expired** - The object has expired, so the value is not longer valid.
(c) **Valid** - The object contains a valid value.

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.

It is important to note that within the Status Server the name, value, and comment associated with an object will be stored as strings. Each string will be a null terminated array of printable characters. While the Status Server will only handle strings of printable characters, the Client API will support null terminated 8 bit character strings. The Client API will be responsible for encoding strings from 8 bit character strings to 7 bit printable character strings prior to sending information across the socket interface. The Client API will also decode strings received from the Status Server. The Status Server will check that strings have a proper set of allowable characters in case an invalid string character was sent across a socket connection outside of the Client API.

2.3 Status Server ⇔ Client Interaction

The Status Server will listen over a socket interface to client requests. The server will service each request and send back an associated response. With the exception of a disconnect request, each client request will receive a response from the Status Server. In most cases, the client will receive a single line response to a request. The exception to the single response model is the case where the client has requested monitoring updates or the client has requested the contents of a directory. Multiple line response messages will always be terminated with either an end-of-response return message or indication of which message is the last response. In the case of a request, which may return multiple lines, the client must not send any new commands until it has fully processed the current command.

In the case of monitored objects, it is also possible for a client to receive an unsolicited message across the interface indicating that monitored information is available. This message is triggered the first time a client-monitored object is updated beyond age and "deadband" restrictions and the client has not already been informed of a monitor update. Once a client is informed that it has monitored information to retrieve, it must initiate a "poll" request to retrieve the information.

The Status Server will maintain a list of client connections and service one client request at a time. Client requests will be handled in a "round-robin" format, so each client will receive an equal opportunity to talk to the Status Server.

Each request received by the Status Server will be checked to make sure it is both a valid command and does not contain any invalid characters. The Status Server will only process requests which contain 7 bit ASCII printable characters terminated with a newline (CR/LF or LF). Figure 2 shows the basic checking performed by the Server for each request.

The specifications, documented in this section, describe the functions performed by both the Status Server and client for each request. There is essentially no added functionality performed by the client if it is connecting via a telnet session. As a result, the client-side functionality is described from the perspective of what is being performed by the C-API library and what a client using the library must be aware of. In addition, the C-API call is included for each command. Some of the descriptions and functionality start to cross the line in to what would typically be included in a detailed design document. However, this should be useful in helping to characterize how the overall system will function.

2.3.1 Access the Status Server

In order for a client to view, manipulate, or subscribe to any information within the Status Server, it must first access the information. The connection established between the client and Status Server will remain persistent until the client chooses to disconnect, or the network connection between client and server is broken. A flowchart illustrating the connection flow is shown in figure 3.

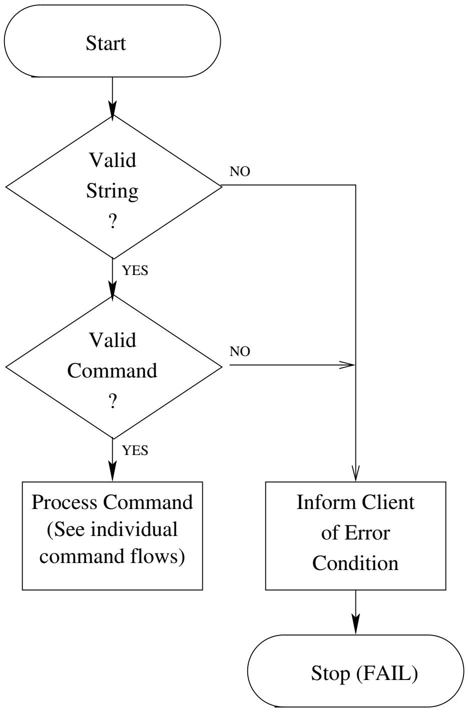

Figure 2: Basic Command Processing Performed by the Status Server

2.3.1.1 Status Server functionality

When the Status Server receives a connection request for a client, it will try to open a socket connection. As part of the connection process, the server will check to make sure that the client is connecting via a valid CFHT internal IP address or via the loopback network (127.x.x.x). If a connection request somehow came in for an invalid address, it will be refused. If a connection can not be opened, the client will receive an indication that the connection was refused. Once the connection is opened, the Status Server is ready to accept requests from the client.

In order to improve the ability to trace a poorly behaving client, the Client API will send a connection message, following the socket connection, supplying the UNIX Process ID (PID) of the process servicing the socket connection along with the program name. The Status Server will check the syntax of this connection message to make sure it is valid. If the syntax is valid, this information will be associated with the client connection and a positive response will be sent. Otherwise, the client will be informed of the error. As a summary, the connection request will result in one of the following conditions which the client must be able to handle.

- Success
- Failure - connection refused
- Failure - syntax error

2.3.1.2 Client-side functionality

The client will initiate a connection request to the Status Server. For traceability purposes, the Client API will send a subsequent message with the UNIX Process ID and program name. When a client chooses to establish a connection with the Status Server, it must decide whether the Client API should automatically try to re-establish a connection to the Status Server if the connection is broken and whether a socket timeout threshold is desired.

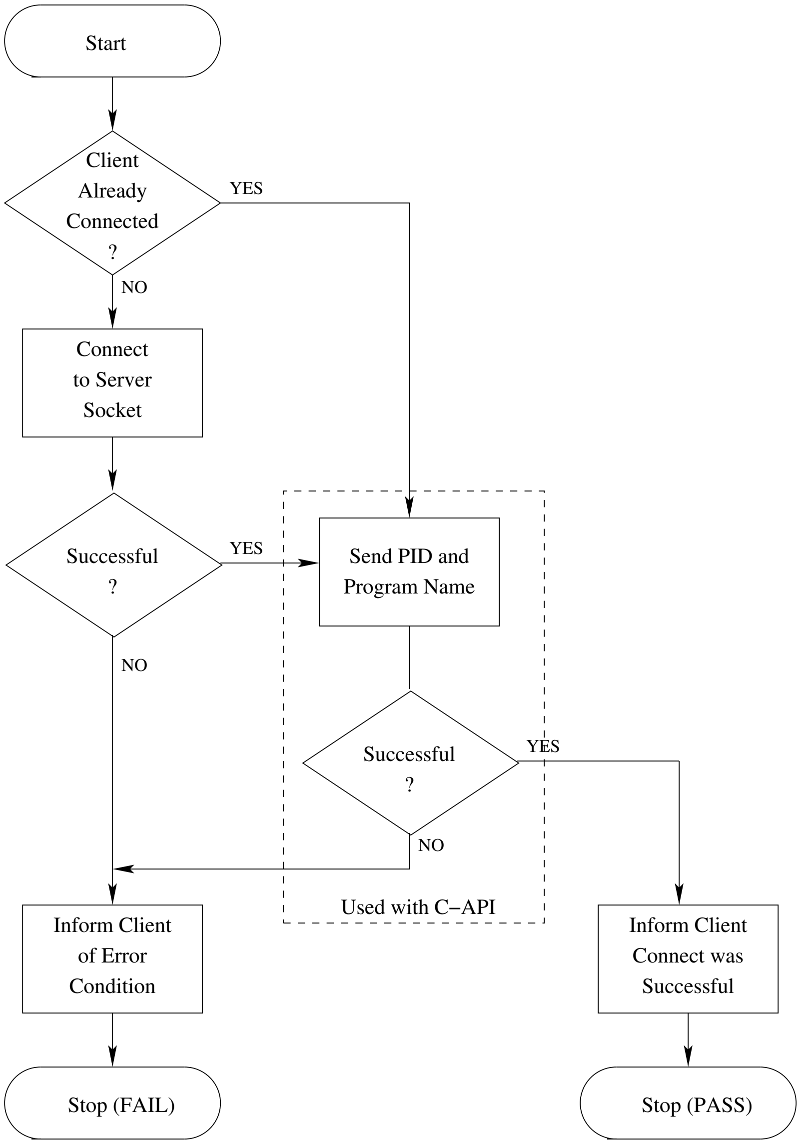

Figure 3: Process Request to Connect to the Status Server

A predefined socket timeout threshold will be defined as part of the Client API. This threshold can be modified by setting a new timeout value via the timeout parameter. If this parameter is less than or equal to 0, the system default will be used. Otherwise, the new value will be used for the socket timeout. The timeout must be specified in seconds.

It is possible for the socket connection between the Client API and Status Server to go down unexpectedly. The Client API will provide the option to retry until the connection is re-established. If the retry option is used, the client program will be blocked until the socket connection is re-established. Any time the Client API retries the connection, it will resend all "touch", "touchdir" and "monitor" commands prior to processing the current API call. When the retry_pause parameter is set to a value greater than or equal to 0, automatic reconnection is turned on. If this value is less than 0, automatic reconnection is turned off. In addition, the retry_pause indicates the number of seconds the Client API will wait to initiate another connection to the Status Server.

The return value from the API call will indicate whether the operation was successful. If the call should happen to fail, more details regarding the failure will be available in cfht_errno.

```
PASSFAIL ssLogon(const char *program_name,
                 const int timeout,
                 const int retry_pause)
```

In addition, the client has the option of setting up client callback functions with the C API for the case where a timeout of the socket occurs or when the socket connection between the Client API and Status Server goes down unexpectedly. If automatic reconnection is enabled, the client callback for disconnection should never be called. Instead a function within the the Client API will be called to initiate a retry.

```
void ssRetryCallback(ss_callback_func retry_fn)
```

```
void ssDiscCallback(ss_disc_func disc_fn)
```

2.3.2 Disconnect from the Status Server

As indicated earlier, the connection between the client and Status Server will remain persistent until the client chooses to disconnect, or the network connection between client and server is broken.

2.3.2.1 Status Server functionality

There are two ways which the Status Server will detect that a client has disconnected from the Status Server. The first is by receiving a specific disconnect message. The second is by detecting that the client connection has been closed. In both cases, the Status Server will clean up the internal resources associated with a client. Some of these resources include:

- Monitors on objects associated with this client connection.
- Buffered input or output transactions.
- All connection specific state information.

The client will detect that the disconnect was successful when an EOF is received across the socket. As a result, the client should expect to see the following response:

- Success - EOF

Figure 4 shows the sequence of events performed by the Status Server upon receiving a disconnect request.

2.3.2.2 Client-side functionality

The C API will disconnect from the Status Server by closing the socket. As part of the disconnect, the Client API must clean up internal resources associated with the Status Server connection. This is also true for the case where the client detects that the client connection with the Status Server has been broken. When using the C API, a disconnect request should not fail unless a client connection isn't available. If the call should happen to fail, more details regarding the failure will be available in cfht_errno.

If the client makes a call to ssLogoff, this will not cause the automatic retry mechanism to trigger. A connection to the Status Server must be subsequently established with another call to ssLogon.

```
PASSFAIL ssLogoff(void)
```

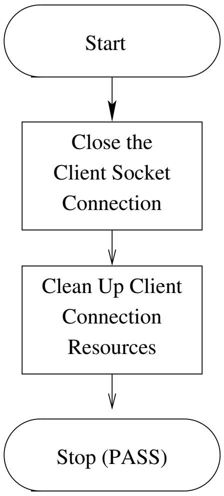

Figure 4: Process Request to Disconnect from the Status Server

2.3.3 Create an object or register the intent to modify 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. While the touch request may seem to be added overhead, there are several reasons why it will be used. First, it can help prevent an inadvertant update of an object due to an incorrect path specification. Secondly, it can provide a mechanism to update the comment and lifetime associated with an object. Finally, it can help prevent inadvertant updates by users using a telnet session. The touch also performs an implicit "mkdir -p" in order to create all the directories required by the object.

2.3.3.1 Status Server functionality

When the Status Server receives a touch request from a client, it will check to see if the object has already been created in the Status Server. If the object does not exist, the Status Server will create the object with an "undefined" (or NULL) value. This means the object exists, but it does not have an associated value yet. If the directory, or set of directories, required to hold the object don't exist they will be created. The Status Server will then check to see if a comment or valid lifetime was specified as part of the touch request. If so, the comment and lifetime values, associated with the object, will be updated. An unspecified lifetime means that the object will not expire.

In response to a touch request the Status Server will send back to the client one of the following responses:

- Success
- Failure - syntax error

Figure 5 shows the sequence of events performed by the Status Server upon receiving a touch request for an object.

2.3.3.2 Client-side functionality

The client must specify the name of the object and optionally specify a comment or lifetime. If a lifetime is not specified, the value of the object will not expire. When using the C API, there should not be any cases where a touch would fail.

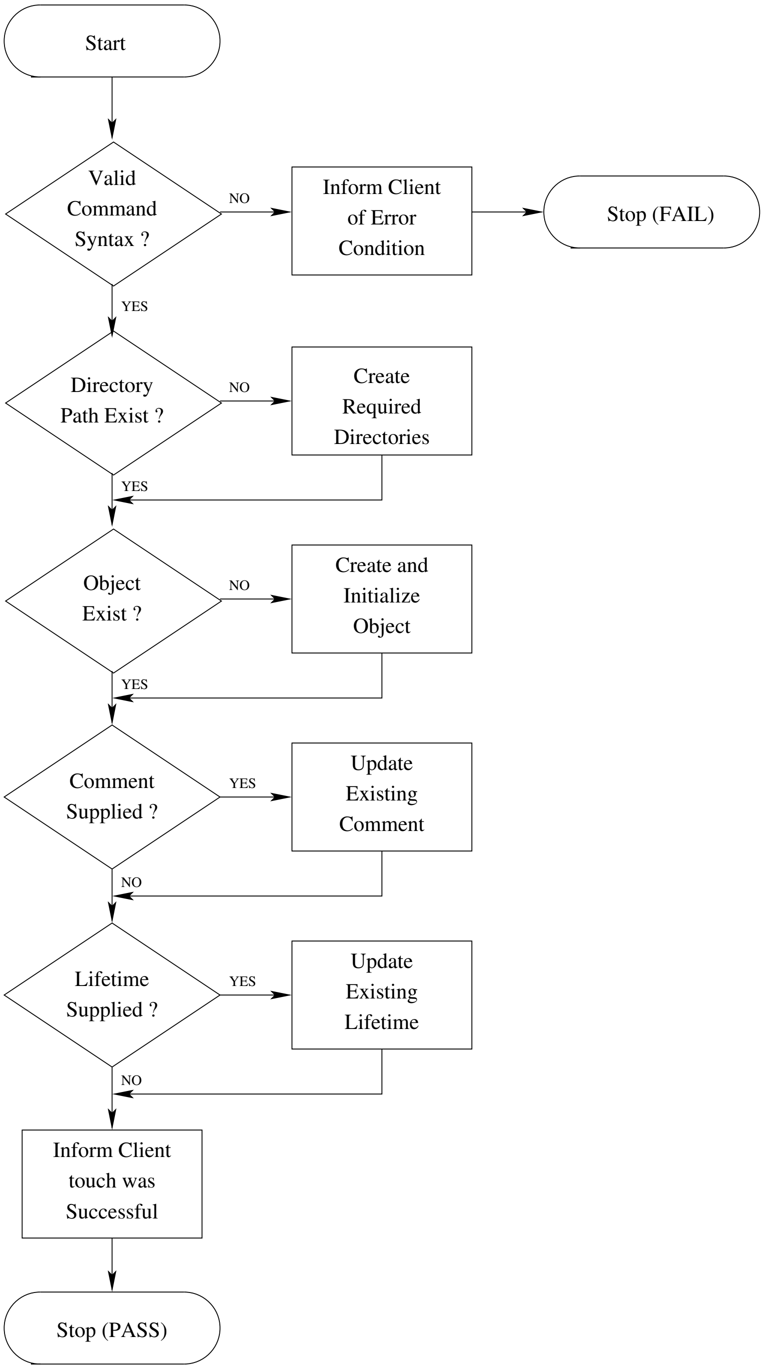

Figure 5: Process Request to Touch an Object

```
void ssTouchObject(const char *name,
                   const char *comment,
                   const int *lifetime)
```

2.3.4 Update an object

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.

2.3.4.1 Status Server functionality

When the Status Server receives an update request from a client, it will check to see if the object exists and whether the client had previously performed a touch request on the object. If not, the server will respond to the client informing it that it does not have permission to update the object. Otherwise, the server will update the value of the object.

As a result of the update, clients monitoring the value of the updated object will be notified as needed. Finally, if a lifetime for the object has been specified, the expiration time for the object will be extended to the current time + lifetime.

The possible responses the client should expect to see from the server as a result of an update request are:

- Success
- Failure - syntax error
- Failure - object does not exist
- Failure - permission denied. (Would occur if the client did not previously perform a touch for the object)

In addition, any clients connected to the Status Server monitoring the updated object may receive a monitor notification message.

Figure 6 shows the sequence of events performed by the Status Server upon receiving a request to update the value of an object.

2.3.4.2 Client-side functionality

The client has the ability to update Status Server objects of the following types:

- **String** - Strings can consist of a sequence of 8 bit ASCII characters with the exception of the NULL character. The NULL character will be used to terminate the string.
- **Boolean** - Data type consisting of two possible values; either TRUE or FALSE.
- **Floating Point** - Double precision floating point number.
- **Integer** - Signed integer number.

Clients using a telnet session will send all data as strings across the socket interface. This is the same way data is sent to the Status Server by the Client API. In order to support the ability to handle 8 bit ASCII characters within a string, the string is encoded into 7 bit ASCII printable characters by the Client API prior to being sent across the interface. In addition, boolean, floating point, and integer data is converted to a string prior to being sent to the Status Server. If the call should happen to fail, more details regarding the failure will be available in cfht_errno.

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

PASSFAIL ssPutDouble(const char *name,
```

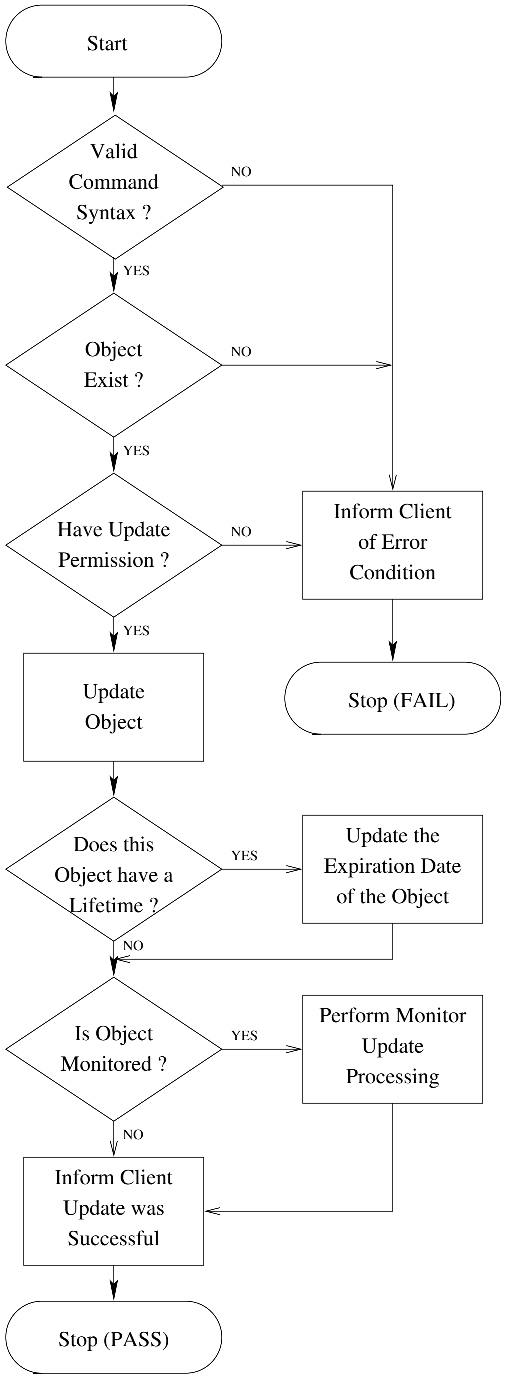

Figure 6: Process Request to Update an Object

```
                        const double value)

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

PASSFAIL ssPutBoolean(const char *name,
```

```
                           const BOOLEAN value)
```

2.3.5 Retrieve an object

Since the Status Server is an open repository without permissions, any client has the ability to retrieve an object from the Status Server.

2.3.5.1 Status Server functionality

When the Status Server receives a request to retrieve information, it will first check to make sure the object exists. If it does exist, it must determine if it has been initialized (not NULL) and whether it hasn't expired. If the object has a valid value, a positive response will be sent to the client with the current value of the object.

The possible responses the client should expect to see as a result of an information retrieval request are:

- Success - value of the object
- Failure - syntax error
- Failure - object does not exist
- Failure - object is expired
- Failure - object is NULL

Figure 7 shows the sequence of events performed by the Status Server upon receiving a request to get the value of an object.

2.3.5.2 Client-side functionality

The client will make a request to retrieve the value of an object. If the request is successful, the value of the Status Server object will be converted from it's encoded string format to the type requested via the C API call. This value will then be stored in the address specified by the user. If the request is not successful, the details regarding the error will be stored in cfht_errno.

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

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

PASSFAIL ssGetString(const char *name,
                     char *value)

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

2.3.6 Check for the existence and status of an object

In some cases, a client may wish to identify the particular state of an object, but not actually retrieve the value of the object itself.

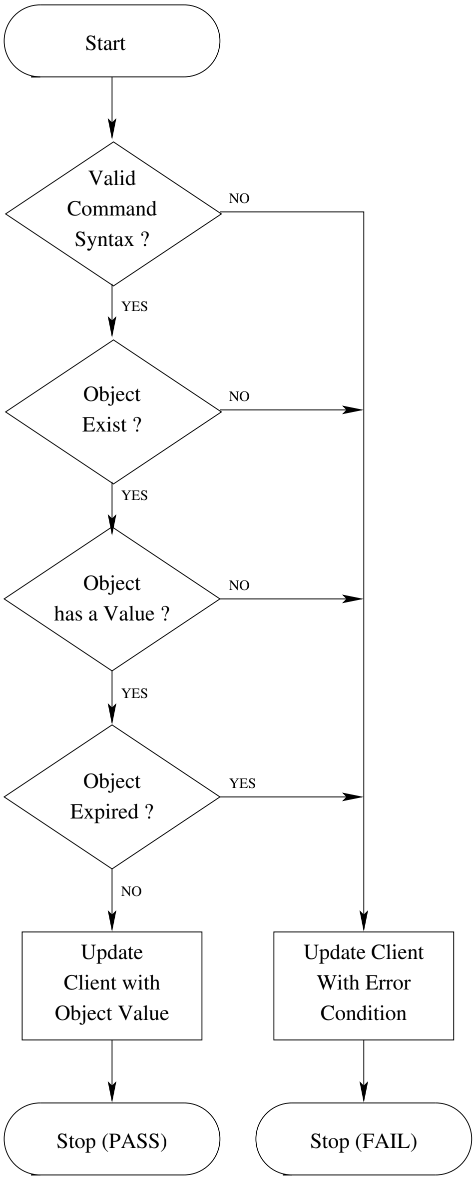

Figure 7: Process Request to Retrieve an Object

2.3.6.1 Status Server functionality

The steps used to identify whether an object exists are very similar to those used when retrieving the value of an object. However, instead of returning an error if the object doesn't exist or when the object value is NULL or expired, these are simply returned as valid object states.

The possible responses the client should expect to see as a result of an object status request are:

- Success - object is valid
- Success - object does not exist
- Success - object value is expired
- Success - object value is undefined (NULL)

- Failure - syntax error

Figure 8 shows the sequence of events performed by the Status Server upon receiving a status request on an object.

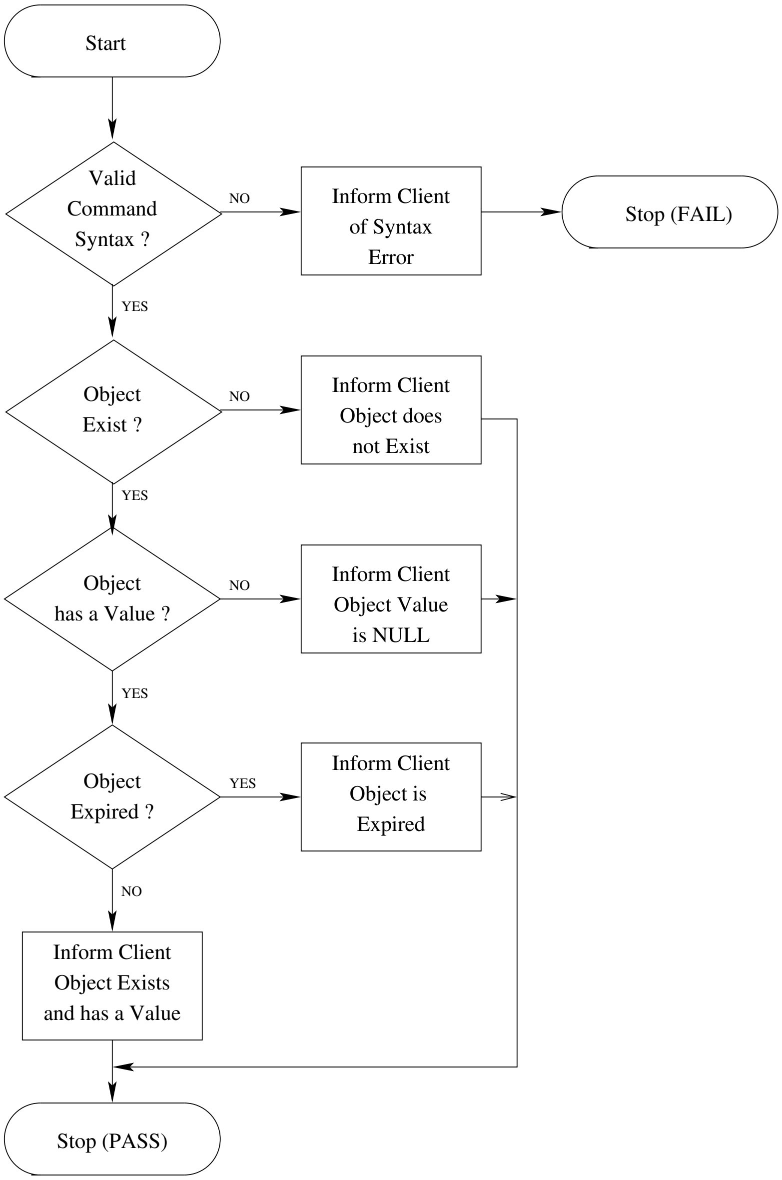

Figure 8: Process Request to Retrieve Object Status

2.3.6.2 Client-side functionality

For clients using the C-API, the status of the object will be stored in a memory location specified by the user. If the call should happen to fail, more details regarding the failure will be available in cfht_errno.

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

2.3.7 Initiate a monitor on an object

Clients can initiate monitors on Status Server objects in order to be informed whenever the value of an object changes. In order to reduce the load on both Status Server and client, a client also has the opportunity to specify a "deadband" range for both floating point and integer objects. Clients connecting via the C-API also have the ability to specify a minimum age for all objects.

It is important to note that, by placing a monitor on an object, this does not guarantee the client will be able to receive and process each modification of an object in the Status Server. Instead, this is a function of the speed and frequency at which the client can process a monitor notification message and initiate a poll message to retrieve monitored information. More details regarding the steps required to process monitored information can be found in both the update and poll flow charts.

2.3.7.1 Status Server functionality

When the Status Server receives a request to monitor an object, it will verify that a valid deadband parameter has been supplied. If so, it will associate a monitoring record for the object with the client connection. As a result, any time the object is modified, a check will be performed to determine whether the client must be notified of the new value of the object.

It is possible to initiate a monitor on a Status Server object which does not exist yet. The monitoring record will simply key off the object name and apply the monitoring rules whenever the object becomes available. This will help to prevent timing issues between clients.

The possible responses the client should expect to see as a result of a monitor request are:

- Success
- Failure - syntax error

Figure 9 shows the sequence of events performed by the Status Server upon receiving a request to monitor an object.

2.3.7.2 Client-side functionality

The client has the option of placing a monitor on any object with optional age and deadband values. If deadband and/or deadband are not desired, it can be set to 0. This indicates the client would like to monitor each change to an object value. When monitors are applied using the C API library, an address must be supplied for both the monitored object value as well as the return code. The return code enables the server to notify a client whenever the state of an object changes. As a result, the client can be informed when an object becomes expired, NULL, or removed.

Once the client is informed that a monitor was successfully applied, the Client API will store an association between the name of the object and the value and return value addresses for the object. The Client API can then process monitor update notification messages and store the value and return codes in the proper memory locations for subsequent use by the client. If the call should happen to fail, more details regarding the failure will be available in cfht_errno.

```
PASSFAIL ssMonitorInt(const char *name,
                      const int deadband,
                      const int age,
                      int *value,
                      ss_ret_t *value_status)

PASSFAIL ssMonitorDouble(const char *name,
                         const double deadband,
                         const int age,
```

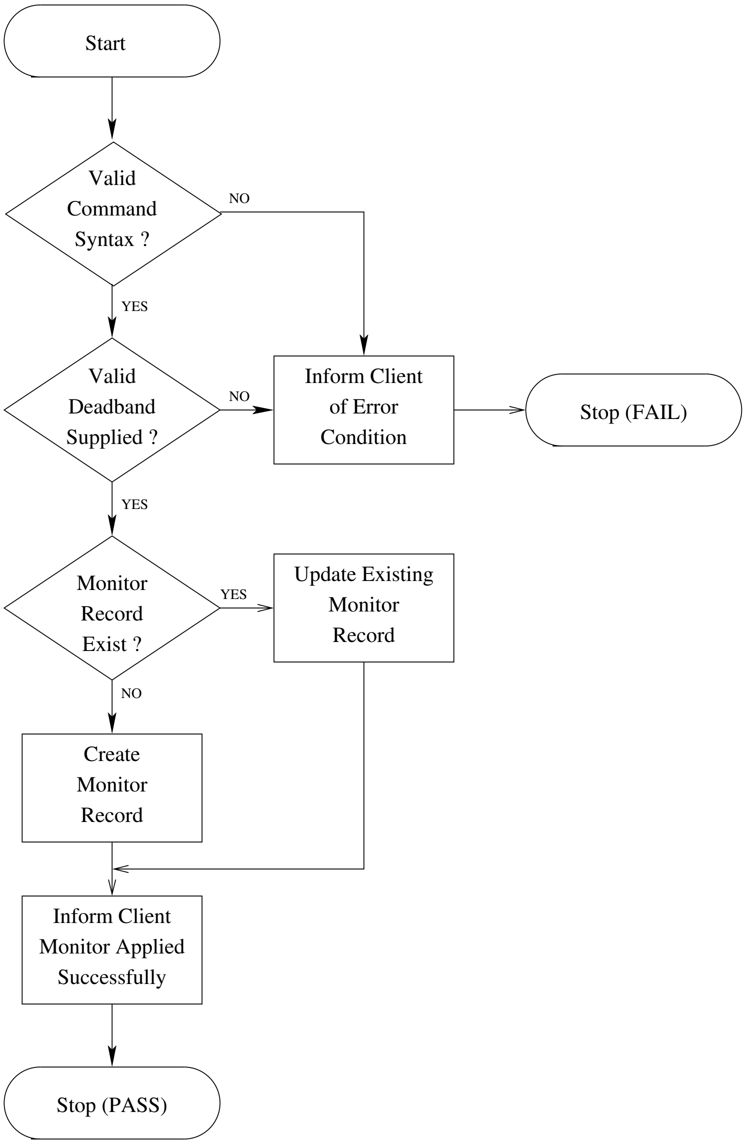

Figure 9: Process Request to Monitor an Object

```
                         double *value,
                         ss_ret_t *value_status)

PASSFAIL ssMonitorString(const char *name,
                         const size_t max_length,
                         const int age,
                         char *value,
                         ss_ret_t *value_status)

PASSFAIL ssMonitorBoolean(const char *name,
                          const int age,
                          BOOLEAN *value,
                          ss_ret_t *value_status)
```

2.3.8 Remove a monitor from an object

Any time a monitor is added to an object in the Status Server, it can be removed by initiating a removal request. Monitors are not automatically removed when an object is removed from the Status Server.

2.3.8.1 Status Server functionality

When the Status Server receives a request to remove a monitor, it will verify that a monitor record was created for this object. If so, it will remove the monitor record.

The possible responses the client should expect to see as a result of a monitor removal request are:

- Success
- Failure - syntax error
- Failure - monitor does not exist

Figure 10 shows the sequence of events performed by the Status Server upon receiving a request to remove a monitor.

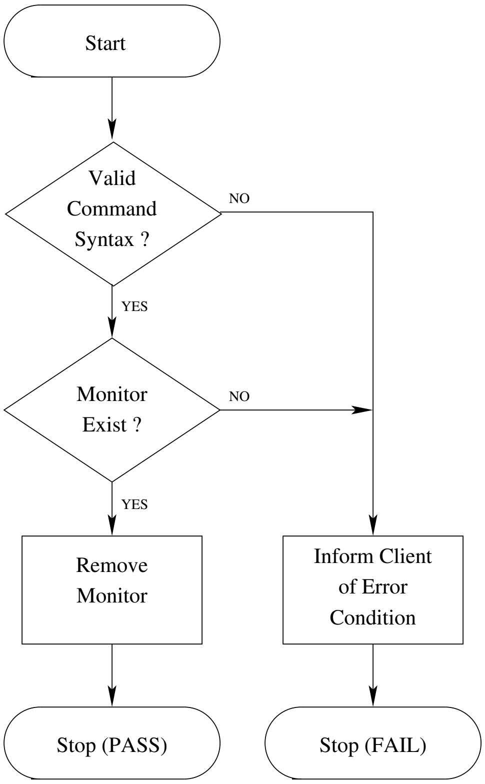

Figure 10: Process Request to Remove a Monitor

2.3.8.2 Client-side functionality

The client will initiate the object removal request and should process the return value. If the call should happen to fail, more details regarding the failure will be available in cfht_errno.

```
ss_ret_t ssKillMonitor(const char *name)
```

2.3.9 Retrieve monitor updates

Several different approaches were reviewed when looking at the best way for a client to receive monitored information. On one hand, it would be nice for the Status Server to send monitored information across the socket whenever it has monitored information to send. However, this can lead to buffering issues if the client connection is not able to receive and process information as quickly as it is being sent by the server.

The approach that will be implemented is for the Status Server send an "out-of-band" notification any time it has monitored information to send to the client and the client has not been notified yet. At this point, the client will most likely respond with a request to retrieve monitor information. The Status Server will then send all monitored information to the client followed by a message indicating to the client that all monitored information has been sent. By using this approach, the client should be guaranteed to get the most current information possible.

An example of a typical message flow between client and server covering the handshaking performed between systems is shown in figure 11.

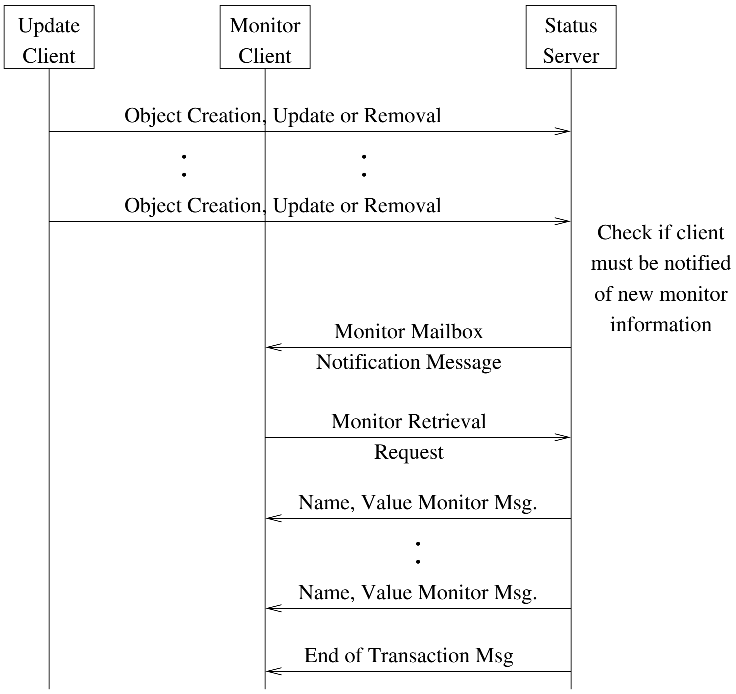

Figure 11: Monitoring flow diagram

2.3.9.1 Status Server functionality

When the Status Server receives a client request to retrieve monitor information, it will verify that the client is currently monitoring Status Server objects. If so, it will cycle through each monitor record and determine if the monitoring criteria indicates that the client should receive an update for the associated monitored object. The following criteria will be applied:

1. If the current value of the object is NULL, expired, or non-existent and this differs from the previous state, the client will be notified with an indication that this object name has an non-valid object value.
2. Is the last value sent to the client appreciably different than the current value? If a deadband has been supplied for this monitored object, a check is made if the current value is different than the last value +/- deadband. If

the current value of the object is such that the deadband offset cannot be applied (example if the value is non-numeric), the deadband value will be ignored and the current and last sent values will be compared based upon the stored string values.

3. If the last case was true, the client is updated with both the name and value of the monitored object.

Once all monitored objects have been processed, an end-of-transaction (EOT) message indication will be sent to the client.

The possible responses the client should expect to see for each monitored object response are:

- Valid: Name = Value
- Invalid: Name = Expired
- Invalid: Name = Not Defined (NULL)
- Invalid: Name = Does not Exist

The possible end of message responses the client should expect to see are:

- Success
- Failure - syntax error

Figure 12 shows the sequence of events performed by the Status Server upon receiving a client request to retrieve updated values for monitored objects.

2.3.9.2 Client-side functionality

The client will initiate the monitor retrieval request and must be prepared to process the return information sent by the Status Server. The client will process each line of data until it receives the end-of-transaction indicator.

For clients using the C API, each line of monitored data will be converted and stored in the memory location which was previously defined during the setup of the monitor. Integer, floating point, and boolean data types will be converted 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. From the C API point of view, this call should not fail unless a client connection is not available. When using the C API, there should not be any cases where a “poll” would fail.

```
void ssPoll(void)
```

2.3.10 Remove an object

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.

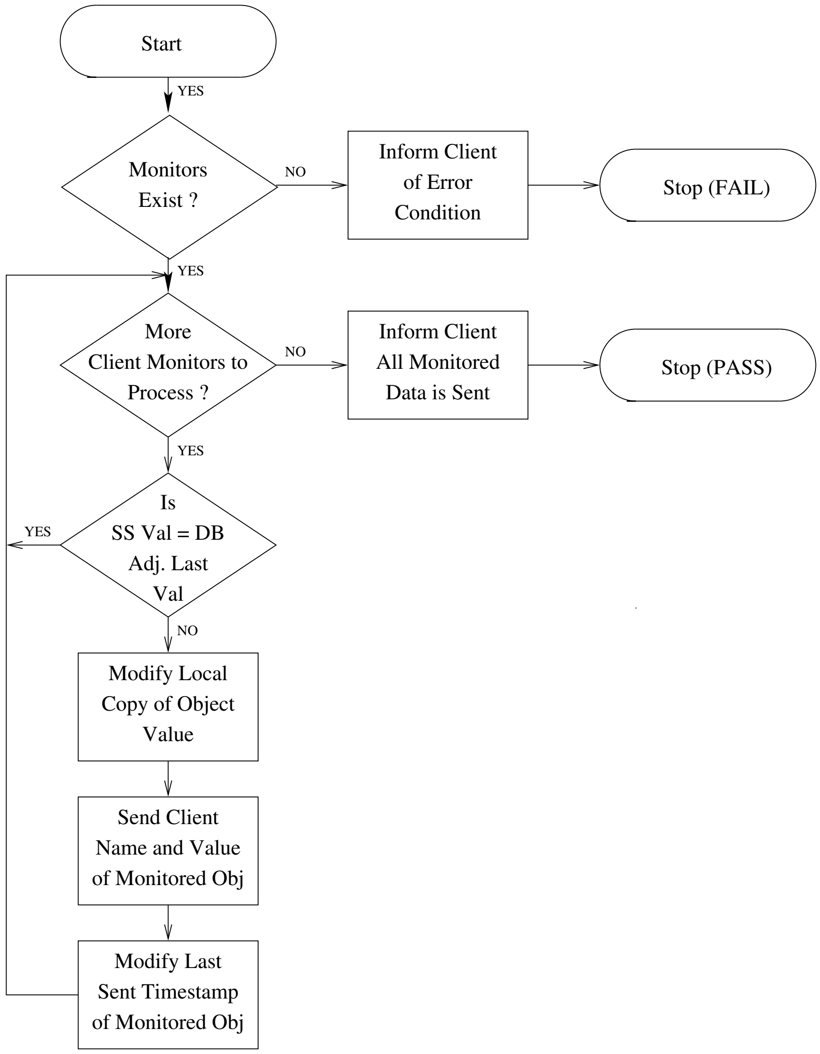

Figure 12: Process Request to Retrieve Monitored Information

2.3.10.1 Status Server functionality

When the Status Server receives an object removal request from a client, it will first check to see if the object exists and whether the client had previously performed a touch on the object. If not, the server will respond indicating that the client does not have permission to remove the object. Otherwise, the object will be removed from the Status Server.

Whenever a removal occurs, the server will check to see if any clients are monitoring the object. If monitor update processing is currently not in progress for those clients, the server will send a monitor notification message if the monitor update criteria has been satisfied.

The possible responses the client should expect to see, from the server, as a result of a removal request are:

- Success
- Failure - syntax error
- Failure - object does not exist
- Failure - permission denied. (Would occur if the client did not previously perform a touch on the object)

In addition, any clients connected to the Status Server monitoring the removed object may receive a monitor notification message.

Figure 13 shows the sequence of events performed by the Status Server upon receiving a client request to remove an object.

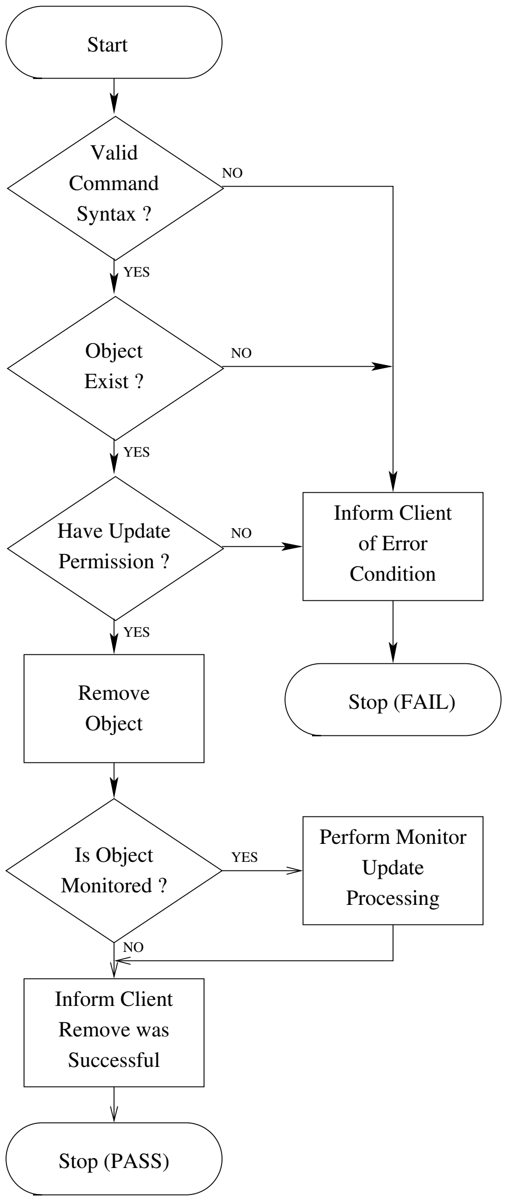

Figure 13: Process Request to Remove an Object

2.3.10.2 Client-side functionality

The client will initiate the removal request and should check the return value to determine whether the operation was successful. If the call should happen to fail, more details regarding the failure will be available in cfht_errno.

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

2.3.11 Get the current directory path

Objects within the Status Server can be referenced either via a fully qualified path-name combination, or a relative path-name combination. In order to manage relative path references, the current path will be maintained for each client connection. This request enables the client to retrieve its current path.

2.3.11.1 Status Server functionality

The Status Server will return what it has defined as the current path for the client. When a client initially connects to the Status Server, its path will be set to the root path of the Status Server directory tree.

The possible responses the client should expect to see from the server as a result of a current directory path request are:

- Success - current path

Figure 14 shows the sequence of events performed by the Status Server upon receiving a client request to retrieve its current directory path.

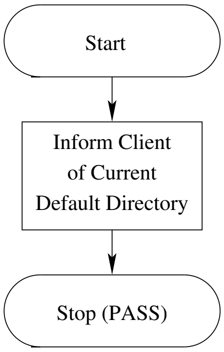

Figure 14: Process Request to Retrieve Current Directory Path

2.3.11.2 Client-side functionality

The client will initiate the current directory request and receive a pointer to a string containing the directory path. When using the C API, there should not be any cases where a request to retrieve the current directory path would fail.

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

2.3.12 Change the current directory

This request will cause the Status Server to modify what it uses as the current directory for relative path references made by a client. The client can specify the new current directory with either a relative path or absolute path.

2.3.12.1 Status Server functionality

The Status Server will check whether the command syntax is correct and whether the requested directory path exists. If so, it will modify what it has defined as the current path for the client.

The possible responses the client should expect to see from the server as a result of a change current directory path request are:

- Success
- Failure - syntax error
- Failure - directory does not exist

Figure 15 shows the sequence of events performed by the Status Server upon receiving a client request to change its current directory path.

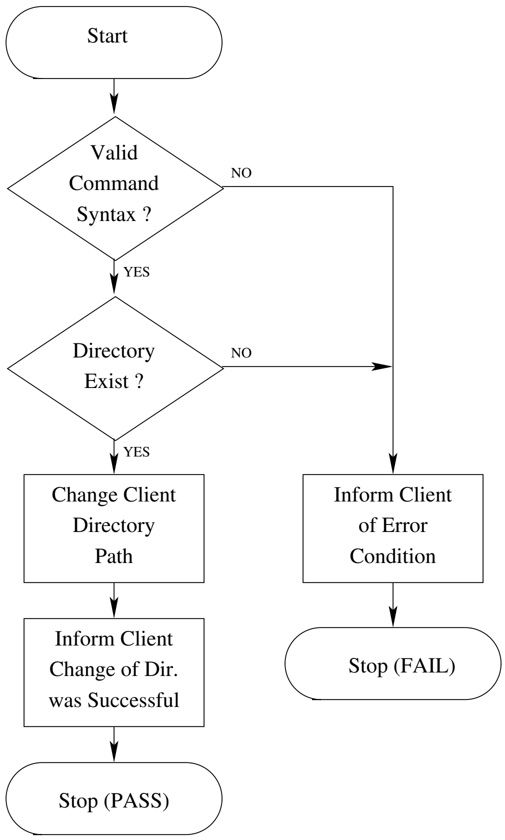

Figure 15: Process Request to Change Current Directory

2.3.12.2 Client-side functionality

The client will initiate the change current directory request and should check the return value to determine whether the operation was successful. If the call should happen to fail, more details regarding the failure will be available in cfht_errno.

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

2.3.13 Create directory or register intent to remove a directory

While required directories are automatically created as part of the touch command when objects are created, it is also possible to explicitly create a directory. This option will create a directory if it doesn't already exist.

In addition, this command must be used prior to removing a directory and all of its contents. When a touch is performed on a directory, it is possible to remove the directory and all the objects within it without performing an explicit touch on each object.

2.3.13.1 Status Server functionality

The Status Server will check whether the command syntax is correct and whether the requested directory path already exists. If the directory does not exist, it will be created. In addition, if additional directories must be created in order to generate the requested path, they will be created as well. The Status Server will then check to see if a comment was specified as part of the touch request. If so, the comment, associated with the directory, will be updated. If not, this value will remain empty.

The possible responses the client should expect to see from the server as a result of a request to change the current directory are:

- Success
- Failure - syntax error

Figure 16 shows the sequence of events performed by the Status Server upon receiving a client request to create or register interest in removing a directory.

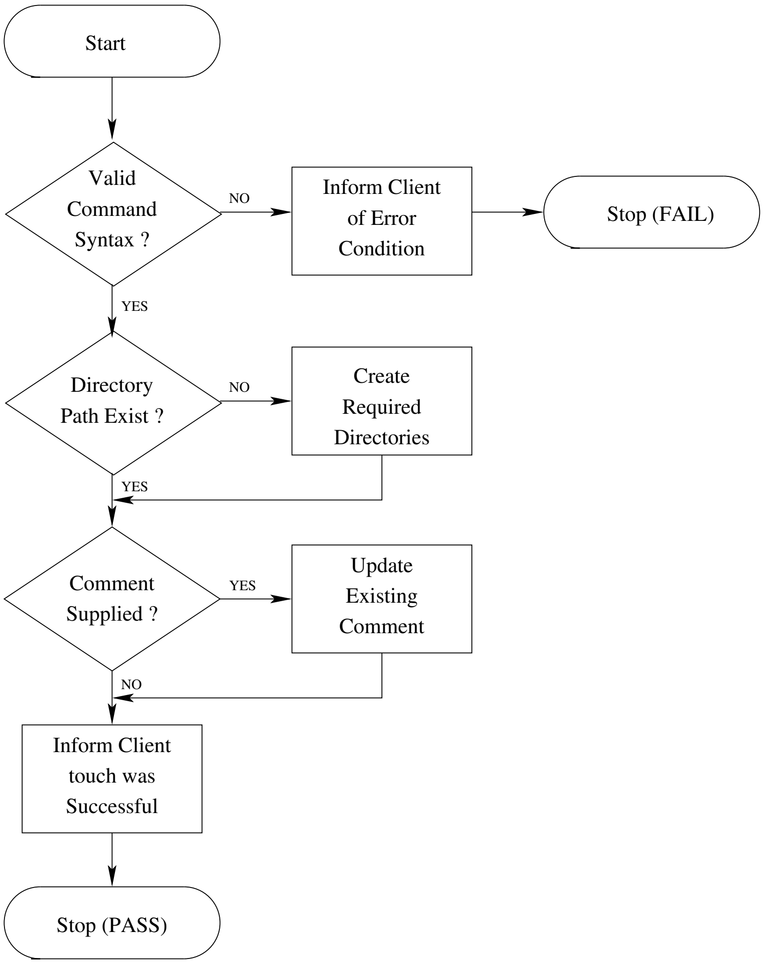

Figure 16: Process Request to Touch a Directory

2.3.13.2 Client-side functionality

Prior to removing a directory and its contents the client must perform a "touchdir" on the directory. The "touchdir" also enables a client to create the directory if it doesn't already exist and to alter the comment associated with a directory. When using the C API, there should not be any cases where a touchdir would fail.

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

2.3.14 Remove a directory

Much like the "rmdir" command on the UNIX file system, it will be possible to remove a directory and all it's objects. In order to help prevent an inadvertant removal of a directory, a touch must be performed on the directory before it can be removed. In addition, the directory must not contain any subdirectories.

2.3.14.1 Status Server functionality

The Status Server will check whether the command syntax is correct, whether the requested directory path already exists, whether the directory does not contain any subdirectories, and whether the client has previously performed a touch request on the directory. If so, this directory and all it's contents (including underlying directories) will be removed.

The possible responses the client should expect to see from the server as a result of a request to remove the current directory are:

- Success
- Failure - syntax error
- Failure - directory not found
- Failure - directory contains subdirectories
- Failure - permission denied (Would occur if the client did not previously perform a touch on this directory)

Figure 17 shows the sequence of events performed by the Status Server upon receiving a client request to remove a directory.

2.3.14.2 Client-side functionality

The client will initiate the directory removal request and should check the return value to determine whether the operation was successful. If the call should happen to fail, more details regarding the failure will be available in cfht_errno.

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

2.3.15 Retrieve the contents of a directory

Much like the "ls" command on the UNIX file system, it will be possible to retrieve the contents of a directory. This command may return more than one line as a response. The client will receive a first line indicating whether the command was successful followed by a sequence of responses with the contents of the directory. The last line sent by the server will indicate the end-of-transaction (EOT). Objects and directories will be returned in a ascending ASCII sort order by name. In addition, this command will allow for the same regular expression matching rules used by the UNIX ls command.

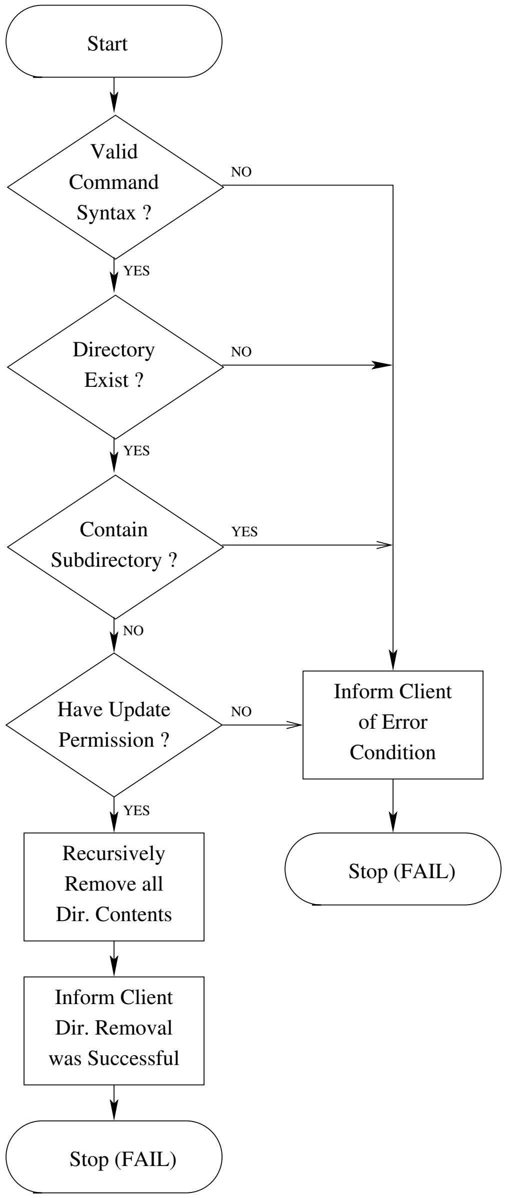

Figure 17: Process Request to Remove a Directory

2.3.15.1 Status Server functionality

The Status Server will check whether the command syntax is correct and whether the requested directory path already exists. If not, it will send an error indication across the interface. Otherwise, it will send across the interface the name and value for all contents in the requested directory. The last response sent across the interface will be the EOT. It is possible that the client would almost immediately receive an EOT without receiving any data. This case would occur if the requested directory is empty. Directory contents are sent to the client in a sorted order based on the object and directory name.

The possible responses the client should expect to see from the server as a result of a directory retrieval request are:

- Success

- Failure - syntax error
- Failure - directory does not exist
- Valid object: Name=Value
- Valid directory: Dir=Name
- EOT indicator

Figure 18 shows the sequence of events performed by the Status Server upon receiving a client request to retrieve the contents of a directory.

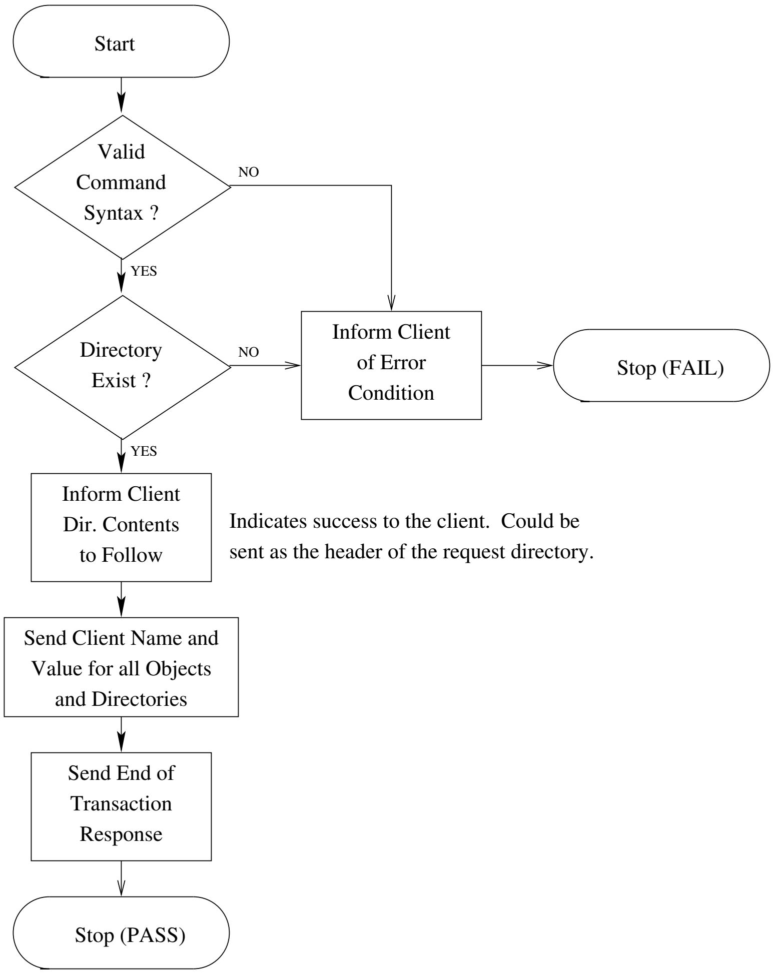

Figure 18: Process Request to Retrieve Contents of a Directory

2.3.15.2 Client-side functionality

The client will initiate the directory content retrieval request and must be prepared to process the return information sent by the Status Server. The client will process each line of data until it receives an EOT.

When using the C API, the client will first make a request to open the directory. The directory can be opened with a recursive option indicating that all underlying contents will be returned as part of the request. The return value from the ssOpendir call will indicate whether the directory could successfully be opened. The client must then call ssReaddir until it receives a NULL indicating that all the directory contents have been returned. Directory contents will be returned in an Object Name=Value and Dir=Name format. For example, /i/cfh12k/etype="FLAT". If the ssOpendir call should happen to fail, more details regarding the failure will be available in cfht_errno.

```
PASSFAIL ssOpendir(const char *path)

char *ssReaddir(void)
```

2.3.16 Initiate a trace

For diagnostic purposes, it may be important to have a more detailed view of what is happening within the Status Server. This may help solve an issue with the way the Status Server is working or help diagnose a misbehaving client.

Trace information will be stored as debug information within the CFHT log. As a result, it will be possible to associate activity within the Status Server with external events to help identify and narrow down problems.

2.3.16.1 Status Server functionality

The Status Server will check whether an existing trace is currently running. If not, a new trace will be initiated.

The output from the trace will be added as debug messages within the CFHT log. The possible responses the client should expect to see from the server as a result of a trace request are:

- Success

Figure 19 shows the sequence of events performed by the Status Server upon receiving a trace request.

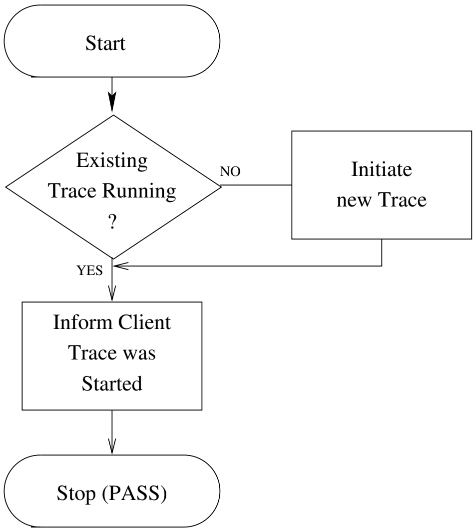

Figure 19: Process Trace Request

2.3.16.2 Client-side functionality

The client can initiate a trace on all Status Server activity. When using the C API, there should not be any cases where a trace initiation request would fail.

```
void ssTraceOn(void)
```

2.3.17 Stop a trace

Once a trace is initiated, a client request must be performed to stop it.

2.3.17.1 Status Server functionality

The Status Server will check whether a trace is currently running. If so, the trace will be stopped.

The possible responses the client should expect to see from the server as a result of a stop trace request are:

- Success

Figure 20 shows the sequence of events performed by the Status Server upon receiving a request to stop a trace.

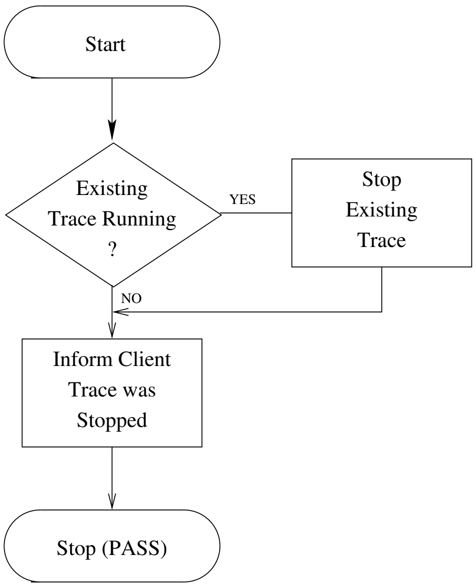

Figure 20: Process Request to Stop a Trace

2.3.17.2 Client-side functionality

A client can stop a trace currently running in the Status Server. When using the C API, there should not be any cases where a request to stop a trace would fail. If a trace was not running and a request was made to stop a trace, the Status Server will still send back a positive response.

```
void ssTraceOff(void)
```

2.3.18 Serialize Status Server data to a file

The Status Server will serialize a copy of itself to disk every 10 minutes. This operation will be done via an automated fashion on the Status Server side. However, it will also be possible for a client to explicitly trigger a serialization to occur via a command.

2.3.18.1 Status Server functionality

The Status Server will serialize a copy of itself to disk any time it receives a message from a client or if the 10 minute interval has expired. The only information which will be saved is the information pertaining to the structure used to hold the Status Server objects. The includes:

- For each object:
  - Object Name - Absolute path of the object
  - Object Value - String representation of the object value
  - Lifetime - Length of time the object can be considered valid
  - Expiration Time - Time the object will be considered expired
  - Comment - Description of the object

By storing object information, in addition to the serialization timestamp, it will be possible to restart the Status Server and have it populate itself based on an previously serialized copy.

It is important to note that information associated with clients will not be stored as part of the serialization process. This is because all client connections will be lost as part of a Status Server restart and restore operation.

The possible responses the client should expect to see from the server as a result of a serialize request are:

- Success

Figure 21 shows the sequence of events performed by the Status Server upon receiving a request to serialize itself to disk.

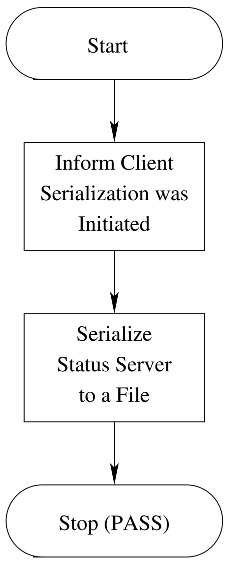

Figure 21: Process Status Server Serialization Request

2.3.18.2 Client-side functionality

A client can request that the Status Server serialize itself to a file. When using the C API, there should not be any cases where a serialization request would fail. It is possible that the serialization itself might fail, but the Status Server does not send a response back to the client once the Serialization is completed. Instead, it sends back a response once it receives a request to initiate serialization.

```
void ssAutosave(void)
```

2.3.19 Initiate a shutdown of the Status Server

For maintenance reasons, it may be necessary to shutdown the Status Server. In order to preserve the current state of information within the server, a copy of the Status Server information will be serialized to disk before an exit is performed.

2.3.19.1 Status Server functionality

Once a shutdown request is received, the Status Server will no longer service any inbound client requests. It will set a timeout interval for the completion of all outbound Status Server initiated messages responses. Once the timeout threshold has been reached, or there are no more outbound responses to send, the Status Server will serialize a copy of itself to disk. Once the serialization is complete, the server will exit.

The client will detect that the shutdown was successful when an EOF is received across the socket. As a result, the client should expect to see the following response:

- Success - EOF

Figure 22 shows the sequence of events performed by the Status Server upon receiving a shutdown request.

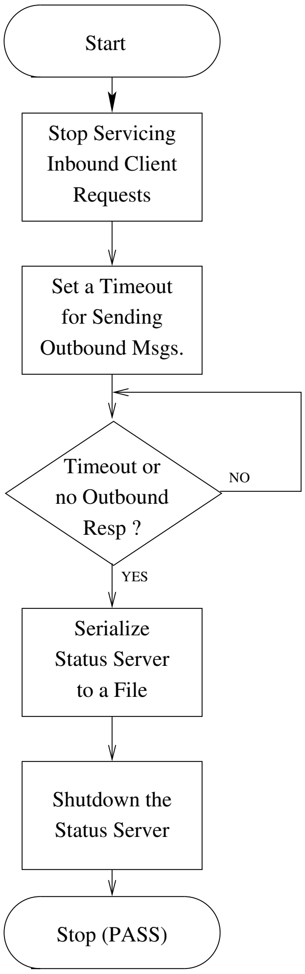

Figure 22: Process Request to Shutdown the Status Server

2.3.19.2 Client-side functionality

The client can request a shutdown of the Status Server. This function will return once the message is sent to the Status Server.

```
void ssShutdown(void)
```

2.4 Example - Storing and Retrieving FITS Header Information

The Status Server will be used as the central staging area for the building of FITS headers replacing the current template files. As a result, it will be possible to build FITS files in a more parallel fashion without the need for client-side synchronization. The change to the existing system will require a modification of the FITS handling library to store and extract FITS information from the Status Server as opposed to a template file. Since this implementation will be one of the first uses of the Status Server, this section outlines how this process will work.

2.4.1 Store a FITS header in the Status Server

When storing a FITS header in the Status Server, the following sequence of operations will be performed by the FITS library.

1. Connect to the Status Server if not already connected.
2. Create a directory to store the FITS information. This could be in the form of /fits/odometer #/.
3. Perform a touch and put to store each FITS card. This can be done in such a way that the object name could be the sequence number for the header and the object value could be the header information itself.

2.4.2 Retrieve a FITS header from the Status Server

When retrieving a FITS header from the Status Server, the following sequence of operations will be performed by the FITS library.

1. Connect to the Status Server if not already connected.
2. Perform a directory retrieval request (ls) to receive all the name-value pair information for objects in the /fits/odometer #/ directory. Since the directory retrieval returns objects in a sorted order by name, they will be received in the order specified by the sequence number imbedded into the name when the FITS keyword objects were created in the Status Server.
3. Perform a touchdir on the /fits/odometer #/ directory.
4. Remove the /fits/odometer #/ directory.

3 Document Change Log

Version	Date	Comments
1.0	March 22, 2002	First release for review.
1.1	April 9, 2002	Revised document based on comments from S. Isani and J. Thomas.
1.2	May 24, 2002	Revised document to reflect changes identified during the Status Server Detailed Design process.