X Session Management Library

X Consortium Standard

Ralph Mor, X Consortium

X Session Management Library: X Consortium Standard

by Ralph Mor

X Version 11, Release 7.7

Version 1.0

Copyright © 1993, 1994 X Consortium

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE X CONSORTIUM BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Except as contained in this notice, the name of the X Consortium shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization from the X Consortium.

X Window System is a trademark of The Open Group.

Table of Contents

Chapter 1. Overview of Session Management

The purpose of the X Session Management Protocol (XSMP) is to provide a uniform mechanism for users to save and restore their sessions. A *session* is a group of clients, each of which has a particular state. The session is controlled by a network service called the *session manager*. The session manager issues commands to its clients on behalf of the user. These commands may cause clients to save their state or to terminate. It is expected that the client will save its state in such a way that the client can be restarted at a later time and resume its operation as if it had never been terminated. A client's state might include information about the file currently being edited, the current position of the insertion point within the file, or the start of an uncommitted transaction. The means by which clients are restarted is unspecified by this protocol.

For purposes of this protocol, a *client* of the session manager is defined as a connection to the session manager. A client is typically, though not necessarily, a process running an application program connected to an X display. However, a client may be connected to more than one X display or not be connected to any X displays at all.

Chapter 2. The Session Management Library

The Session Management Library (SMlib) is a low-level "C" language interface to XSMP. It is expected that higher level toolkits, such as Xt, will hide many of the details of session management from clients. Higher level toolkits might also be developed for session managers to use, but no such effort is currently under way.

SMlib has two parts to it:

- One set of functions for clients that want to be part of a session
- One set of functions for session managers to call

Some applications will use both sets of functions and act as *nested session managers*. That is, they will be both a session manager and a client of another session. An example is a mail program that could start a text editor for editing the text of a mail message. The mail program is part of a regular session and, at the same time, is also acting as a session manager to the editor.

Clients initialize by connecting to the session manager and obtaining a *client-ID* that uniquely identifies them in the session. The session manager maintains a list of properties for each client in the session. These properties describe the client's environment and, most importantly, describe how the client can be restarted (via an SmRestartCommand). Clients are expected to save their state in such a way as to allow multiple instantiations of themselves to be managed independently. For example, clients may use their client-ID as part of a filename in which to store the state for a particular instantiation. The client-ID should be saved as part of the SmRestartCommand so that the client will retain the same ID after it is restarted.

Once the client initializes itself with the session manager, it must be ready to respond to messages from the session manager. For example, it might be asked to save its state or to terminate. In the case of a shutdown, the session manager might give each client a chance to interact with the user and cancel the shutdown.

Chapter 3. Understanding SMlib's Dependence on ICE

The X Session Management Protocol is layered on top of the Inter-Client Exchange (ICE) Protocol. The ICE protocol is designed to multiplex several protocols over a single connection. As a result, working with SMlib requires a little knowledge of how the ICE library works.

The ICE library utilizes callbacks to process messages. When a client detects that there is data to read on an ICE connection, it should call the `IceProcessMessages` function. `IceProcessMessages` will read the message header and look at the major opcode in order to determine which protocol the message was intended for. The appropriate protocol library will then be triggered to unpack the message and hand it off to the client via a callback.

The main point to be aware of is that an application using SMlib must have some code that detects when there is data to read on an ICE connection. This can be done via a `select` call on the file descriptor for the ICE connection, but more typically, `XtAppAddInput` will be used to register a callback that will invoke `IceProcessMessages` each time there is data to read on the ICE connection.

To further complicate things, knowing which file descriptors to call `select` on requires an understanding of how ICE connections are created. On the client side, a call must be made to `SmcOpenConnection` in order to open a connection with a session manager. `SmcOpenConnection` will internally makea call into `IceOpenConnection` which will, in turn, determine if an ICE connection already exists between the client and session manager. Most likely, a connection will not already exist and a new ICE connection will be created. The main point to be aware of is that, on the client side, it is not obvious when ICE connections get created or destroyed, because connections are shared when possible. To deal with this, the ICE library lets the application register watch procedures that will be invoked each time an ICE connection is opened or closed. These watch procedures could be used to add or remove ICE file descriptors from the list of descriptors to call `select` on.

On the session manager side, things work a bit differently. The session manager has complete control over the creation of ICE connections. The session manager has to first call `IceListenForConnections` in order to start listening for connections from clients. Once a connection attempt is detected, `IceAcceptConnection` must be called, and the session manager can simply add the new ICE file descriptor to the list of descriptors to call `select` on.

For further information on the library functions related to ICE connections, see the “Inter-Client Exchange Library” standard.

Chapter 4. Header Files and Library Name

Applications (both session managers and clients) should include the header file <`X11/SM/SMlib.h`>. This header file defines all of the SMlib data structures and function prototypes. `SMlib.h` includes the header file <`X11/SM/SM.h`>, which defines all of the SMlib constants.

Because SMlib is dependent on ICE, applications should link against SMlib and ICElib by using “`-lSM -lICE`”.

Chapter 5. Session Management Client (Smc) Functions

This section discusses how Session Management clients:

- Connect to the Session Manager
- Close the connection
- Modify callbacks
- Set, delete, and retrieve Session Manager properties
- Interact with the user
- Request a “Save Yourself”
- Request a “Save Yourself Phase 2”
- Complete a “Save Yourself”
- Use Smc informational functions
- Handle Errors

Connecting to the Session Manager

To open a connection with a session manager, use `SmcOpenConnection`

SmcConn **SmcOpenConnection**(*network_ids_list*, *context*, *xsmp_major_rev*, *xsmp_minor_rev*, *mask*, *callbacks*, *previous_id*, *client_id_ret*, *error_length*, *error_string_ret*);

network_ids_list	Specifies the network ID(s) of the session manager.
context	A pointer to an opaque object or `NULL`. Used to determine if an ICE connection can be shared (see below [6]).
xsmp_major_rev	The highest major version of the XSMP the application supports.
xsmp_minor_rev	The highest minor version of the XSMP the application supports (for the specified *xsmp_major_rev*).
mask	A mask indicating which callbacks to register.
callbacks	The callbacks to register. These callbacks are used to respond to messages from the session manager.
previous_id	The client ID from the previous session.
client_id_ret	The client ID for the current session is returned.

`error_length`	Length of the *`error_string_ret`* argument passed in.
`error_string_ret`	Returns a null-terminated error message, if any. The *`error_string_ret`* argument points to user supplied memory. No more than *`error_length`* bytes are used.

The *`network_ids_list`* argument is a null-terminated string containing a list of network IDs for the session manager, separated by commas. If *`network_ids_list`* is `NULL`, the value of the `SESSION_MANAGER` environment variable will be used. Each network ID has the following format:

`tcp/<hostname>:<portnumber>`	or
`decnet/<hostname>::<objname>`	or
`local/<hostname>:<path>`	

An attempt will be made to use the first network ID. If that fails, an attempt will be made using the second network ID, and so on.

After the connection is established, `SmcOpenConnection` registers the client with the session manager. If the client is being restarted from a previous session, *`previous_id`* should contain a null terminated string representing the client ID from the previous session. If the client is first joining the session, *`previous_id`* should be set to `NULL`. If *`previous_id`* is specified but is determined to be invalid by the session manager, SMlib will re-register the client with *`previous_id`* set to `NULL`.

If `SmcOpenConnection` succeeds, it returns an opaque connection pointer of type `SmcConn` and the *`client_id_ret`* argument contains the client ID to be used for this session. The *`client_id_ret`* should be freed with a call to `free` when no longer needed. On failure, `SmcOpenConnection` returns `NULL`, and the reason for failure is returned in *`error_string_ret`*.

Note that SMlib uses the ICE protocol to establish a connection with the session manager. If an ICE connection already exists between the client and session manager, it might be possible for the same ICE connection to be used for session management.

The context argument indicates how willing the client is to share the ICE connection with other protocols. If context is `NULL`, then the caller is always willing to share the connection. If context is not `NULL`, then the caller is not willing to use a previously opened ICE connection that has a different non-`NULL` context associated with it.

As previously discussed (section 3, “Understanding SMlib's Dependence on ICE”), the client will have to keep track of when ICE connections are created or destroyed (using `IceAddConnectionWatch` and `IceRemoveConnectionWatch` and will have to call `IceProcessMessages` each time a `select` shows that there is data to read on an ICE connection. For further information, see the “Inter-Client Exchange Library” standard.

The callbacks argument contains a set of callbacks used to respond to session manager events. The mask argument specifies which callbacks are set. All of the callbacks specified in this version of SMlib are mandatory. The mask argument is necessary in order to maintain backwards compatibility in future versions of the library.

The following values may be ORed together to obtain a *`mask`* value:

```
SmcSaveYourselfProcMask
SmcDieProcMask
SmcSaveCompleteProcMask
SmcShutdownCancelledProcMask
```

For each callback, the client can register a pointer to client data. When SMlib invokes the callback, it will pass the client data pointer.

```
typedef struct {

 struct {
  SmcSaveYourselfProc callback;
  SmPointer client_data;
 } save_yourself;

 struct {
  SmcDieProc callback;
  SmPointer client_data;
 } die;

 struct {
  SmcSaveCompleteProc callback;
  SmPointer client_data;
 } save_complete;

 struct {
  SmcShutdownCancelledProc callback;
  SmPointer client_data;
 } shutdown_cancelled;

} SmcCallbacks;
```

The Save Yourself Callback

The Save Yourself callback is of type `SmcSaveYourselfProc`

`typedef void (*`**`SaveYourselfProc`**`)(`*`smc_conn`*`, `*`client_data`*`, `*`save_type`*`, `*`shutdown`*`, `*`interact_style`*`, `*`fast`*`);`

smc_conn	The session management connection object.
client_data	Client data specified when the callback was registered.
save_type	Specifies the type of information that should be saved.
shut_down	Specifies if a shutdown is taking place.
interact_style	The type of interaction allowed with the user.
fast	if True, then client should save its state as quickly as possible.

The session manager sends a “Save Yourself” message to a client either to checkpoint it or just before termination so that it can save its state. The client responds with zero or more calls to `SmcSetProperties` to update the properties indicating how to restart the client. When all the properties have been set, the client calls `SmcSaveYourselfDone`

If *`interact_style`* is `SmInteractStyleNone` the client must not interact with the user while saving state. If *`interact_style`* is `SmInteractStyleErrors` the client may interact with the user only if an error condition arises. If *`interact_style`* is `SmInteractStyleAny` then the client may interact with the user for any purpose. Because only one client can interact with the user at a time, the client must call `SmcInteractRequest` and wait for an "Interact" message from the session manager. When the client is done interacting with the user, it calls `SmcInteractDone` The client may only call `SmcInteractRequest` after it receives a "Save Yourself" message and before it calls `SmcSaveYourselfDone`

If *`save_type`* is `SmSaveLocal` the client must update the properties to reflect its current state. Specifically, it should save enough information to restore the state as seen by the user of this client. It should not affect the state as seen by other users. If *`save_type`* is `SmSaveGlobal` the user wants the client to commit all of its data to permanent, globally accessible storage. If *`save_type`* is `SmSaveBoth` the client should do both of these (it should first commit the data to permanent storage before updating its properties).

Some examples are as follows:

- If a word processor were sent a "Save Yourself" with a type of `SmSaveLocal` it could create a temporary file that included the current contents of the file, the location of the cursor, and other aspects of the current editing session. It would then update its SmRestartCommand property with enough information to find this temporary file.
- If a word processor were sent a "Save Yourself" with a type of `SmSaveGlobal` it would simply save the currently edited file.
- If a word processor were sent a "Save Yourself" with a type of `SmSaveBoth` it would first save the currently edited file. It would then create a temporary file with information such as the current position of the cursor and what file is being edited. Finally, it would update its SmRestartCommand property with enough information to find the temporary file.

The *`shutdown`* argument specifies whether the system is being shut down. The interaction is different depending on whether or not shutdown is set. If not shutting down, the client should save its state and wait for a "Save Complete" message. If shutting down, the client must save state and then prevent interaction until it receives either a "Die" or a "Shutdown Cancelled."

The *`fast`* argument specifies that the client should save its state as quickly as possible. For example, if the session manager knows that power is about to fail, it would set *`fast`* to `True`.

The Die Callback

The Die callback is of type `SmcDieProc`

```
typedef void (*SmcDieProc)(smc_conn, client_data);
```

`smc_conn` The session management connection object.

`client_data`	Client data specified when the callback was registered.

The session manager sends a "Die" message to a client when it wants it to die. The client should respond by calling `SmcCloseConnection`. A session manager that behaves properly will send a "Save Yourself" message before the "Die" message.

The Save Complete Callback

The Save Complete callback is of type `SmcSaveCompleteProc`

```
typedef void (*SmcSaveCompleteProc)(smc_conn, client_data);
```

`smc_conn`	The session management connection object.
`client_data`	Client data specified when the callback was registered.

The Shutdown Cancelled Callback

The Shutdown Cancelled callback is of type `SmcShutdownCancelledProc`

```
typedef void (*SmcShutdownCancelledProc)(smc_conn, client_data);
```

`smc_conn`	The session management connection object.
`client_data`	Client data specified when the callback was registered.

The session manager sends a "Shutdown Cancelled" message when the user cancelled the shutdown during an interaction (see section 5.5, "Interacting With the User"). The client can now continue as if the shutdown had never happened. If the client has not called `SmcSaveYourselfDone` yet, it can either abort the save and then call `SmcSaveYourselfDone` with the success argument set to `False` or it can continue with the save and then call `SmcSaveYourselfDone` with the *`success`* argument set to reflect the outcome of the save.

Closing the Connection

To close a connection with a session manager, use `SmcCloseConnection`

```
SmcCloseStatus SmcCloseConnection(smc_conn, count, reason_msgs);
```

`smc_conn`	The session management connection object.
`count`	The number of reasons for closing the connection.
`reason_msgs`	The reasons for closing the connection.

The *`reason_msgs`* argument will most likely be `NULL` if resignation is expected by the client. Otherwise, it contains a list of null-terminated Compound Text strings representing the reason for termination. The session manager should display these reason messages to the user.

Note that SMlib used the ICE protocol to establish a connection with the session manager, and various protocols other than session management may be active on the ICE connection. When `SmcCloseConnection` is called, the ICE connection will be closed only if all protocols have been shutdown on the connection. Check the

ICElib standard for `IceAddConnectionWatch` and `IceRemoveConnectionWatch` to learn how to set up a callback to be invoked each time an ICE connection is opened or closed. Typically this callback adds/removes the ICE file descriptor from the list of active descriptors to call `select` on (or calls `XtAppAddInput` or `XtRemoveInput`).

`SmcCloseConnection` returns one of the following values:

- `SmcClosedNow` - the ICE connection was closed at this time, the watch procedures were invoked, and the connection was freed.
- `SmcClosedASAP` - an IO error had occurred on the connection, but `SmcCloseConnection` is being called within a nested `IceProcessMessages` The watch procedures have been invoked at this time, but the connection will be freed as soon as possible (when the nesting level reaches zero and `IceProcessMessages` returns a status of `IceProcessMessagesConnectionClosed`
- `SmcConnectionInUse` - the connection was not closed at this time, because it is being used by other active protocols.

Modifying Callbacks

To modify callbacks set up in `SmcOpenConnection` use `SmcModifyCallbacks`

void **SmcModifyCallbacks**(*smc_conn*, *mask*, *callbacks*);

smc_conn	The session management connection object.
mask	A mask indicating which callbacks to modify.
callbacks	The new callbacks.

When specifying a value for the *mask* argument, the following values may be ORed together:

```
SmcSaveYourselfProcMask
SmcDieProcMask
SmcSaveCompleteProcMask
SmcShutdownCancelledProcMask
```

Setting, Deleting, and Retrieving Session Management Properties

To set session management properties for this client, use `SmcSetProperties`

void **SmcSetProperties**(*smc_conn*, *num_props*, *props*);

smc_conn	The session management connection object.
num_props	The number of properties.
props	The list of properties to set.

The properties are specified as an array of property pointers. Previously set property values may be over-written using the `SmcSetProperties` function. Note that the session manager is not expected to restore property values when the session is restarted. Because of this, clients should not try to use the session manager as a database for storing application specific state.

For a description of session management properties and the SmProp structure, see section 7, "Session Management Properties."

To delete properties previously set by the client, use `SmcDeleteProperties`

void **SmcDeleteProperties**(*smc_conn*, *num_props*, *prop_names*);

smc_conn	The session management connection object.
num_props	The number of properties.
prop_names	The list of properties to set.

To get properties previously stored by the client, use `SmcGetProperties`

Status **SmcGetProperties**(*smc_conn*, *prop_reply_proc*, *client_data*);

smc_conn	The session management connection object.
prop_reply_proc	The callback to be invoked when the properties reply comes back.
client_data	This pointer to client data will be passed to the `SmcPropReplyProc` callback.

The return value of `SmcGetProperties` is zero for failure and a positive value for success.

Note that the library does not block until the properties reply comes back. Rather, a callback of type `SmcPropReplyProc` is invoked when the data is ready.

typedef void (***SmcPropReplyProc**)(*smc_conn*, *client_data*, *num_props*, *props*);

smc_conn	The session management connection object.
client_data	This pointer to client data will be passed to the `SmcPropReplyProc` callback.
num_props	The number of properties returned.
props	The list of properties returned.

To free each property, use `SmFreeProperty` (see section 8, "Freeing Data"). To free the actual array of pointers, use `free`

Interacting With the User

After receiving a "Save Yourself" message with an *interact_style* of `SmInteractStyleErrors` or `SmInteractStyleAny` the client may choose to interact with the user. Because only one client can interact with the user at a time, the client must call `SmcInteractRequest` and wait for an "Interact" message from the session manager.

`Status` **`SmcInteractRequest`**`(`*`smc_conn`*`,` *`dialog_type`*`,` *`interact_proc`*`,` *`client_data`*`);`

smc_conn	The session management connection object.
dialog_type	The type of dialog the client wishes to present to the user.
interact_proc	The callback to be invoked when the “Interact” message arrives from the session manager.
client_data	This pointer to client data will be passed to the `SmcInteractProc` callback when the “Interact” message arrives.

The return value of `SmcInteractRequest` is zero for failure and a positive value for success.

The *dialog_type* argument specifies either `SmDialogError` indicating that the client wants to start an error dialog, or `SmDialogNormal` meaning that the client wishes to start a nonerror dialog.

Note that if a shutdown is in progress, the user may have the option of cancelling the shutdown. If the shutdown is cancelled, the clients that have not interacted yet with the user will receive a “Shutdown Cancelled” message instead of the “Interact” message.

The `SmcInteractProc` callback will be invoked when the “Interact” message arrives from the session manager.

`typedef void (*`**`SmcInteractProc`**`)(`*`smc_conn`*`,` *`client_data`*`);`

smc_conn	The session management connection object.
client_data	Client data specified when the callback was registered.

After interacting with the user (in response to an “Interact” message), you should call `SmcInteractDone`

`void` **`SmcInteractDone`**`(`*`smc_conn`*`,` *`cancel_shutdown`*`);`

smc_conn	The session management connection object.
cancel_shutdown	If `True`, indicates that the user requests that the entire shutdown be cancelled.

The *cancel_shutdown* argument may only be `True` if the corresponding “Save Yourself” specified `True` for shutdown and `SmInteractStyleErrors` or `SmInteractStyleAny` for the *interact_style*.

Requesting a Save Yourself

To request a checkpoint from the session manager, use `SmcRequestSaveYourself`

`void` **`SmcRequestSaveYourself`**`(`*`smc_conn`*`,` *`save_type`*`,` *`shutdown`*`,` *`interact_style`*`,` *`fast`*`,` *`global`*`);`

smc_conn	The session management connection object.

save_type	Specifies the type of information that should be saved.
shutdown	Specifies if a shutdown is taking place.
interact_style	The type of interaction allowed with the user.
fast	If `True` the client should save its state as quickly as possible.
global	Controls who gets the “Save Yourself.”

The *save_type*, *shutdown*, *interact_style*, and *fast* arguments are discussed in more detail in section 5.1.1, “The Save Yourself Callback.”

If *global* is set to `True` then the resulting “Save Yourself” should be sent to all clients in the session. For example, a vendor of a Uninterruptible Power Supply (UPS) might include a Session Management client that would monitor the status of the UPS and generate a fast shutdown if the power is about to be lost.

If global is set to `False` then the “Save Yourself” should only be sent to the client that requested it.

Requesting a Save Yourself Phase 2

In response to a “Save Yourself”, the client may request to be informed when all the other clients are quiescent so that it can save their state. To do so, use `SmcRequestSaveYourselfPhase2`

Status **SmcRequestSaveYourselfPhase2**(*smc_conn*, *save_yourself_phase2_proc*, *client_data*);

smc_conn	The session management connection object.
save_type_phase2_proc	The callback to be invoked when the “Save Yourself Phase 2” message arrives from the session manager.
client_data	This pointer to client data will be passed to the `SmcSaveYourselfPhase2Proc` callback when the “Save Yourself Phase 2” message arrives.

The return value of `SmcRequestSaveYourselfPhase2` is zero for failure and a positive value for success.

This request is needed by clients that manage other clients (for example, window managers, workspace managers, and so on). The manager must make sure that all of the clients that are being managed are in an idle state so that their state can be saved.

Completing a Save Yourself

After saving state in response to a “Save Yourself” message, you should call `SmcSaveYourselfDone`

void **SmcSaveYourselfDone**(*smc_conn*, *success*);

smc_conn	The session management connection object.

success If `True` the “Save Yourself” operation was completed successfully.

Before calling `SmcSaveYourselfDone` the client must have set each required property at least once since the client registered with the session manager.

Using Smc Informational Functions

`int` **`SmcProtocolVersion`**`(`*`smc_conn`*`);`

`SmcProtocolVersion` returns the major version of the session management protocol associated with this session.

`int` **`SmcProtocolRevision`**`(`*`smc_conn`*`);`

`SmcProtocolRevision` returns the minor version of the session management protocol associated with this session.

`char *`**`SmcVendor`**`(`*`smc_conn`*`);`

`SmcVendor` returns a string that provides some identification of the owner of the session manager. The string should be freed with a call to `free`

`char *`**`SmcRelease`**`(`*`smc_conn`*`);`

`SmcRelease` returns a string that provides the release number of the session manager. The string should be freed with a call to `free`

`char *`**`SmcClientID`**`(`*`smc_conn`*`);`

`SmcClientID` returns a null-terminated string for the client ID associated with this connection. This information was also returned in `SmcOpenConnection` (it is provided here for convenience). Call `free` on this pointer when the client ID is no longer needed.

`IceConn` **`SmcGetIceConnection`**`(`*`smc_conn`*`);`

`SmcGetIceConnection` returns the ICE connection object associated with this session management connection object. The ICE connection object can be used to get some additional information about the connection. Some of the more useful functions which can be used on the IceConn are `IceConnectionNumber`, `IceConnectionString`, `IceLastSentSequenceNumber`, `IceLastReceivedSequenceNumber`, and `IcePing`. For further information, see the “Inter-Client Exchange Library” standard.

Error Handling

If the client receives an unexpected protocol error from the session manager, an error handler is invoked by SMlib. A default error handler exists that simply prints the error message to `stderr` and exits if the severity of the error is fatal. The client can change this error handler by calling the `SmcSetErrorHandler` function.

`SmcErrorHandler` **`SmcSetErrorHandler`**`(`*`handler`*`);`

The error handler. You should pass `NULL` to restore the default handler.

`SmcSetErrorHandler` returns the previous error handler.

The `SmcErrorHandler` has the following type:

`typedef void (*`**`SmcErrorHandler`**`)(`*`smc_conn`*`, `*`swap`*`, `*`offending_minor_opcode`*`, `*`offending_sequence_num`*`, `*`error_class`*`, `*`severity`*`, `*`values`*`);`

smc_conn	The session management connection object.
swap	A flag that indicates if the specified values need byte swapping.
offending_minor_opcode	The minor opcode of the offending message.
offending_sequence_num	The sequence number of the offending message.
error_class	The error class of the offending message.
severity	`IceCanContinue`, `IceFatalToProtocol`, or `IceFatalToConnection`
values	Any additional error values specific to the minor opcode and class.

Note that this error handler is invoked for protocol related errors. To install an error handler to be invoked when an IO error occurs, use `IceSetIOErrorHandler` For further information, see the "Inter-Client Exchange Library" standard.

Chapter 6. Session Management Server (Sms) Functions

This section discusses how Session Management servers:

- Initialize the library
- Register the client
- Send a “Save Yourself” message
- Send a “Save Yourself Phase 2” message
- Send an “Interact” message
- Send a “Save Complete” message
- Send a “Die” message
- Cancel a shutdown
- Return properties
- Ping a client
- Clean up after a client disconnects
- Use Sms informational functions
- Handle errors

Initializing the Library

`SmsInitialize` is the first SMlib function that should be called by a session manager. It provides information about the session manager and registers a callback that will be invoked each time a new client connects to the session manager.

`Status` **`SmsInitialize`**`(`*`vendor`*`, `*`release`*`, `*`new_client_proc`*`, `*`manager_data`*`, `*`host_based_auth_proc`*`, `*`error_length`*`, `*`error_string_ret`*`);`

vendor	A string specifying the session manager vendor.
release	A string specifying the session manager release number.
new_client_proc	Callback to be invoked each time a new client connects to the session manager.
manager_data	When the `SmsNewClientProc` callback is invoked, this pointer to manager data will be passed.
host_based_auth_proc	Host based authentication callback.

error_length	Length of the *error_string_ret* argument passed in.
error_string_ret	Returns a null-terminated error message, if any. The *error_string_ret* points to user supplied memory. No more than *error_length* bytes are used.

After the `SmsInitialize` function is called, the session manager should call the `IceListenForConnections` function to listen for new connections. Afterwards, each time a client connects, the session manager should call `IceAcceptConnection`

See section 9, "Authentication of Clients," for more details on authentication (including host based authentication). Also see the "Inter-Client Exchange Library" standard for further details on listening for and accepting ICE connections.

Each time a new client connects to the session manager, the `SmsNewClientProc` callback is invoked. The session manager obtains a new opaque connection object that it should use for all future interaction with the client. At this time, the session manager must also register a set of callbacks to respond to the different messages that the client might send.

```
typedef Status (*SmsNewClientProc)(sms_conn, manager_data, mask_ret,
callbacks_ret, failure_reason_ret);
```

sms_conn	A new opaque connection object.
manager_data	Manager data specified when the callback was registered.
mask_ret	On return, indicates which callbacks were set by the session manager.
callbacks_ret	On return, contains the callbacks registered by the session manager.
failure_reason_ret	Failure reason returned.

If a failure occurs, the `SmsNewClientProc` should return a zero status as well as allocate and return a failure reason string in *failure_reason_ret*. SMlib will be responsible for freeing this memory.

The session manager must register a set of callbacks to respond to client events. The *mask_ret* argument specifies which callbacks are set. All of the callbacks specified in this version of SMlib are mandatory. The *mask_ret* argument is necessary in order to maintain backwards compatibility in future versions of the library.

The following values may be ORed together to obtain a mask value:

```
SmsRegisterClientProcMask
SmsInteractRequestProcMask
SmsInteractDoneProcMask
SmsSaveYourselfRequestProcMask
SmsSaveYourselfP2RequestProcMask
SmsSaveYourselfDoneProcMask
SmsCloseConnectionProcMask
SmsSetPropertiesProcMask
SmsDeletePropertiesProcMask
```

`SmsGetPropertiesProcMask`

For each callback, the session manager can register a pointer to manager data specific to that callback. This pointer will be passed to the callback when it is invoked by SMlib.

```
typedef struct {
 struct {
  SmsRegisterClientProc callback;
  SmPointer manager_data;
 } register_client;

 struct {
  SmsInteractRequestProc callback;
  SmPointer manager_data;
 } interact_request;

 struct {
  SmsInteractDoneProc callback;
  SmPointer manager_data;
 } interact_done;

 struct {
  SmsSaveYourselfRequestProc callback;
  SmPointer manager_data;
 } save_yourself_request;

 struct {
  SmsSaveYourselfPhase2RequestProc callback;
  SmPointer manager_data;
 } save_yourself_phase2_request;

 struct {
  SmsSaveYourselfDoneProc callback;
  SmPointer manager_data;
 } save_yourself_done;

 struct {
  SmsCloseConnectionProc callback;
  SmPointer manager_data;
 } close_connection;

 struct {
  SmsSetPropertiesProc callback;
  SmPointer manager_data;
 } set_properties;

 struct {
  SmsDeletePropertiesProc callback;
  SmPointer manager_data;
 } delete_properties;

 struct {
```

```
  SmsGetPropertiesProc callback;
  SmPointer manager_data;
 } get_properties;

} SmsCallbacks;
```

The Register Client Callback

The Register Client callback is the first callback that will be invoked after the client connects to the session manager. Its type is `SmsRegisterClientProc`

`typedef Status (*`**`SmsRegisterClientProc`**`)(`*`sms_conn`*`, `*`manager_data`*`, `*`previous_id`*`);`

sms_conn	The session management connection object.
manager_data	Manager data specified when the callback was registered.
previous_id	The client ID from the previous session.

Before any further interaction takes place with the client, the client must be registered with the session manager.

If the client is being restarted from a previous session, *previous_id* will contain a null-terminated string representing the client ID from the previous session. Call `free` on the *previous_id* pointer when it is no longer needed. If the client is first joining the session, *previous_id* will be `NULL`.

If *previous_id* is invalid, the session manager should not register the client at this time. This callback should return a status of zero, which will cause an error message to be sent to the client. The client should re-register with previous_id set to `NULL`.

Otherwise, the session manager should register the client with a unique client ID by calling the `SmsRegisterClientReply` function (to be discussed shortly), and the `SmsRegisterClientProc` callback should return a status of one.

The Interact Request Callback

The Interact Request callback is of type `SmsInteractRequestProc`

`typedef void (*`**`SmsInteractRequestProc`**`)(`*`sms_conn`*`, `*`manager_data`*`, `*`dialog_type`*`);`

sms_conn	The session management connection object.
manager_data	Manager data specified when the callback was registered.
dialog_type	The type of dialog the client wishes to present to the user.

When a client receives a “Save Yourself” message with an *interact_style* of `SmInteractStyleErrors` or `SmInteractStyleAny` the client may choose to interact with the user. Because only one client can interact with the user at a time, the client must request to interact with the user. The session manager should keep a queue of all clients wishing to interact. It should send an “Interact” message to one client at a time and wait for an “Interact Done” message before continuing with the next client.

The *`dialog_type`* argument specifies either `SmDialogError` indicating that the client wants to start an error dialog, or `SmDialogNormal` meaning that the client wishes to start a nonerror dialog.

If a shutdown is in progress, the user may have the option of cancelling the shutdown. If the shutdown is cancelled (specified in the “Interact Done” message), the session manager should send a “Shutdown Cancelled” message to each client that requested to interact.

The Interact Done Callback

When the client is done interacting with the user, the `SmsInteractDoneProc` callback will be invoked.

`typedef void (*`**`SmsInteractDoneProc`**`)(`*`sms_conn`*`, `*`manager_data`*`, `*`cancel_shutdown`*`);`

`sms_conn`	The session management connection object.
`manager_data`	Manager data specified when the callback was registered.
`cancel_shutdown`	Specifies if the user requests that the entire shutdown be cancelled.

Note that the shutdown can be cancelled only if the corresponding “Save Yourself” specified `True` for shutdown and `SmInteractStyleErrors` or `SmInteractStyleAny` for the *`interact_style`*.

The Save Yourself Request Callback

The Save Yourself Request callback is of type `SmsSaveYourselfRequestProc`

`typedef void (*`**`SaveYourselfRequestProc`**`)(`*`sms_conn`*`, `*`manager_data`*`, `*`save_type`*`, `*`shutdown`*`, `*`interact_style`*`, `*`fast`*`, `*`global`*`);`

`sms_conn`	The session management connection object.
`manager_data`	Manager data specified when the callback was registered.
`save_type`	Specifies the type of information that should be saved.
`shutdown`	Specifies if a shutdown is taking place.
`interact_style`	The type of interaction allowed with the user.
`fast`	If `True` the client should save its state as quickly as possible.
`global`	Controls who gets the “Save Yourself.”

The Save Yourself Request prompts the session manager to initiate a checkpoint or shutdown. For information on the *`save_type`*, *`shutdown`*, *`interact_style`*, and *`fast`* arguments, see section 6.3, “Sending a Save Yourself Message.”

If *`global`* is set to `True` then the resulting “Save Yourself” should be sent to all applications. If *`global`* is set to `False` then the “Save Yourself” should only be sent to the client that requested it.

The Save Yourself Phase 2 Request Callback

The Save Yourself Phase 2 Request callback is of type `SmsSaveYourselfPhase2RequestProc`

`typedef void (*`**`SmsSaveYourselfPhase2RequestProc`**`)(`*`sms_conn`*`, `*`manager_data`*`);`

sms_conn	The session management connection object.
manager_data	Manager data specified when the callback was registered.

This request is sent by clients that manage other clients (for example, window managers, workspace managers, and so on). Such managers must make sure that all of the clients that are being managed are in an idle state so that their state can be saved.

The Save Yourself Done Callback

When the client is done saving its state in response to a “Save Yourself” message, the `SmsSaveYourselfDoneProc` will be invoked.

`typedef void (*`**`SaveYourselfDoneProc`**`)(`*`sms_conn`*`, `*`manager_data`*`, `*`success`*`);`

sms_conn	The session management connection object.
manager_data	Manager data specified when the callback was registered.
success	If `True` the Save Yourself operation was completed successfully.

Before the “Save Yourself Done” was sent, the client must have set each required property at least once since it registered with the session manager.

The Connection Closed Callback

If the client properly terminates (that is, it calls `SmcCloseConnection`, the `SmsCloseConnectionProc` callback is invoked.

`typedef void (*`**`SmsCloseConnectionProc`**`)(`*`sms_conn`*`, `*`manager_data`*`, `*`count`*`, `*`reason_msgs`*`);`

sms_conn	The session management connection object.
manager_data	Manager data specified when the callback was registered.
count	The number of reason messages.
reason_msgs	The reasons for closing the connection.

The *reason_msgs* argument will most likely be `NULL` and the *count* argument zero (0) if resignation is expected by the user. Otherwise, it contains a list of null-terminated Compound Text strings representing the reason for termination. The session manager should display these reason messages to the user.

Call `SmFreeReasons` to free the reason messages. For further information, see section 8, “Freeing Data”

The Set Properties Callback

When the client sets session management properties, the `SmsSetPropertiesProc` callback will be invoked.

typedef void (***SmsSetPropertiesProc**)(*sms_conn*, *manager_data*, *num_props*, *props*);

sms_conn	The session management connection object.
manager_data	Manager data specified when the callback was registered.
num_props	The number of properties.
props	The list of properties to set.

The properties are specified as an array of property pointers. For a description of session management properties and the SmProp structure, see section 7, “Session Management Properties.”

Previously set property values may be over-written. Some properties have predefined semantics. The session manager is required to store nonpredefined properties.

To free each property, use `SmFreeProperty`. For further information, see section 8, “Freeing Data” You should free the actual array of pointers with a call to `free`

The Delete Properties Callback

When the client deletes session management properties, the `SmsDeletePropertiesProc` callback will be invoked.

typedef void (***SmsDeletePropertiesProc**)(*sms_conn*, *manager_data*, *num_props*, *prop_names*);

sms_conn	The session management connection object.
manager_data	Manager data specified when the callback was registered.
num_props	The number of properties.
prop_names	The list of properties to delete.

The properties are specified as an array of strings. For a description of session management properties and the SmProp structure, see section 7, “Session Management Properties.”

The Get Properties Callback

The `SmsGetPropertiesProc` callback is invoked when the client wants to retrieve properties it set.

typedef void (***SmsGetPropertiesProc**)(*sms_conn*, *manager_data*);

sms_conn	The session management connection object.

`manager_data`	Manager data specified when the callback was registered.

The session manager should respond by calling `SmsReturnProperties`. All of the properties set for this client should be returned.

Registering the Client

To register a client (in response to a `SmsRegisterClientProc` callback), use `SmsRegisterClientReply`.

`Status` **`SmsRegisterClientReply`**`(`*`sms_conn`*`, `*`client_id`*`);`

`sms_conn`	The session management connection object.
`client_id`	A null-terminated string representing a unique client ID.

The return value of `SmsRegisterClientReply` is zero for failure and a positive value for success. Failure will occur if SMlib can not allocate memory to hold a copy of the client ID for it's own internal needs.

If a non-`NULL` *`previous_id`* was specified when the client registered itself, *`client_id`* should be identical to *`previous_id`*.

Otherwise, *`client_id`* should be a unique ID freshly generated by the session manager. In addition, the session manager should send a “Save Yourself” message with *`type`* = `Local`, *`shutdown`* = `False`, *`interact-style`* = `None`, and *`fast`* = `False` immediately after registering the client.

Note that once a client ID has been assigned to the client, the client keeps this ID indefinitely. If the client is terminated and restarted, it will be reassigned the same ID. It is desirable to be able to pass client IDs around from machine to machine, from user to user, and from session manager to session manager, while retaining the identity of the client. This, combined with the indefinite persistence of client IDs, means that client IDs need to be globally unique.

You should call the `SmsGenerateClientID` function to generate a globally unique client ID.

`char *`**`SmsGenerateClientID`**`(`*`sms_conn`*`);`

`sms_conn`	The session management connection object.

`NULL` will be returned if the ID could not be generated. Otherwise, the return value of the function is the client ID. It should be freed with a call to `free` when no longer needed.

Sending a Save Yourself Message

To send a “Save Yourself” to a client, use `SmsSaveYourself`.

`void` **`SmsSaveYourself`**`(`*`sms_conn`*`, `*`save_type`*`, `*`shutdown`*`, `*`interact_style`*`, `*`fast`*`);`

`sms_conn`	The session management connection object.

save_type	Specifies the type of information that should be saved.
shutdown	Specifies if a shutdown is taking place.
interact_style	The type of interaction allowed with the user.
fast	If `True` the client should save its state as quickly as possible.

The session manager sends a “Save Yourself” message to a client either to checkpoint it or just before termination so that it can save its state. The client responds with zero or more “Set Properties” messages to update the properties indicating how to restart the client. When all the properties have been set, the client sends a “Save Yourself Done” message.

If *interact_style* is `SmInteractStyleNone` the client must not interact with the user while saving state. If *interact_style* is `SmInteractStyleErrors` the client may interact with the user only if an error condition arises. If *interact_style* is `SmInteractStyleAny` then the client may interact with the user for any purpose. The client must send an “Interact Request” message and wait for an “Interact” message from the session manager before it can interact with the user. When the client is done interacting with the user, it should send an “Interact Done” message. The “Interact Request” message can be sent any time after a “Save Yourself” and before a “Save Yourself Done.”

If *save_type* is `SmSaveLocal` the client must update the properties to reflect its current state. Specifically, it should save enough information to restore the state as seen by the user of this client. It should not affect the state as seen by other users. If *save_type* is `SmSaveGlobal` the user wants the client to commit all of its data to permanent, globally accessible storage. If *save_type* is `SmSaveBoth` the client should do both of these (it should first commit the data to permanent storage before updating its properties).

The *shutdown* argument specifies whether the session is being shut down. The interaction is different depending on whether or not shutdown is set. If not shutting down, then the client can save and resume normal operation. If shutting down, the client must save and then must prevent interaction until it receives either a “Die” or a “Shutdown Cancelled,” because anything the user does after the save will be lost.

The *fast* argument specifies that the client should save its state as quickly as possible. For example, if the session manager knows that power is about to fail, it should set *fast* to `True`.

Sending a Save Yourself Phase 2 Message

In order to send a “Save Yourself Phase 2” message to a client, use `SmsSaveYourselfPhase2`

```
void SmsSaveYourselfPhase2(sms_conn);
```

sms_conn	The session management connection object.

The session manager sends this message to a client that has previously sent a “Save Yourself Phase 2 Request” message. This message informs the client that all other clients are in a fixed state and this client can save state that is associated with other clients.

Sending an Interact Message

To send an “Interact” message to a client, use `SmsInteract`.

`void SmsInteract(sms_conn);`

sms_conn The session management connection object.

The “Interact” message grants the client the privilege of interacting with the user. When the client is done interacting with the user, it must send an “Interact Done” message to the session manager.

Sending a Save Complete Message

To send a “Save Complete” message to a client, use `SmsSaveComplete`.

`void SmsSaveComplete(sms_conn);`

sms_conn The session management connection object.

The session manager sends this message when it is done with a checkpoint. The client is then free to change its state.

Sending a Die Message

To send a “Die” message to a client, use `SmsDie`.

`void SmsDie(sms_conn);`

sms_conn The session management connection object.

Before the session manager terminates, it should wait for a “Connection Closed” message from each client that it sent a “Die” message to, timing out appropriately.

Cancelling a Shutdown

To cancel a shutdown, use `SmsShutdownCancelled`.

`void SmsShutdownCancelled(sms_conn);`

sms_conn The session management connection object.

The client can now continue as if the shutdown had never happened. If the client has not sent a “Save Yourself Done” message yet, it can either abort the save and send a “Save Yourself Done” with the success argument set to `False` or it can continue with the save and send a “Save Yourself Done” with the *success* argument set to reflect the outcome of the save.

Returning Properties

In response to a “Get Properties” message, the session manager should call `SmsReturnProperties`.

void **SmsReturnProperties**(*sms_conn*, *num_props*, *props*);

sms_conn	The session management connection object.
num_props	The number of properties.
props	The list of properties to return to the client.

The properties are returned as an array of property pointers. For a description of session management properties and the SmProp structure, see section 7, “Session Management Properties.”

Pinging a Client

To check that a client is still alive, you should use the `IcePing` function provided by the ICE library. To do so, the ICE connection must be obtained using the `SmsGetIceConnection` (see section 6.12, “Using Sms Informational Functions”).

void **IcePing**(*ice_conn*, *ping_reply_proc*, *client_data*);

ice_conn	A valid ICE connection object.
ping_reply_proc	The callback to invoke when the Ping reply arrives.
client_data	This pointer will be passed to the `IcePingReplyProc` callback.

When the Ping reply is ready (if ever), the `IcePingReplyProc` callback will be invoked. A session manager should have some sort of timeout period, after which it assumes the client has unexpectedly died.

typedef void (***IcePingReplyProc**)(*ice_conn*, *client_data*);

ice_conn	A valid ICE connection object.
client_data	The client data specified in the call to `IcePing`

Cleaning Up After a Client Disconnects

When the session manager receives a “Connection Closed” message or otherwise detects that the client aborted the connection, it should call the `SmsCleanUp` function in order to free up the connection object.

void **SmsCleanUp**(*sms_conn*);

sms_conn	The session management connection object.

Using Sms Informational Functions

int **SmsProtocolVersion**(*sms_conn*);

`SmsProtocolVersion` returns the major version of the session management protocol associated with this session.

int **SmsProtocolRevision**(*sms_conn*);

`SmsProtocolRevision` returns the minor version of the session management protocol associated with this session.

`char *`**`SmsClientID`**`(`*`sms_conn`*`);`

`SmsClientID` returns a null-terminated string for the client ID associated with this connection. You should call `free` on this pointer when the client ID is no longer needed.

To obtain the host name of a client, use `SmsClientHostName`. This host name will be needed to restart the client.

`char *`**`SmsClientHostName`**`(`*`sms_conn`*`);`

The string returned is of the form *protocol/hostname*, where *protocol* is one of {`tcp`, `decnet`, `local`}. You should call `free` on the string returned when it is no longer needed.

`IceConn `**`SmsGetIceConnection`**`(`*`sms_conn`*`);`

`SmsGetIceConnection` returns the ICE connection object associated with this session management connection object. The ICE connection object can be used to get some additional information about the connection. Some of the more useful functions which can be used on the IceConn are `IceConnectionNumber` and `IceLastSequenceNumber`. For further information, see the “Inter-Client Exchange Library” standard.

Error Handling

If the session manager receives an unexpected protocol error from a client, an error handler is invoked by SMlib. A default error handler exists which simply prints the error message (it does not exit). The session manager can change this error handler by calling `SmsSetErrorHandler`.

`SmsErrorHandler `**`SmsSetErrorHandler`**`(`*`handler`*`);`

The error handler. You should pass `NULL` to restore the default handler.

`SmsSetErrorHandler` returns the previous error handler. The `SmsErrorHandler` has the following type:

`typedef void (*`**`SmsErrorHandler`**`)(`*`sms_conn`*`, `*`swap`*`, `*`offending_minor_opcode`*`, `*`offending_sequence_num`*`, `*`error_class`*`, `*`severity`*`, `*`values`*`);`

sms_conn	The session management connection object.
swap	A flag which indicates if the specified values need byte swapping.
offending_minor_opcode	The minor opcode of the offending message.
offending_sequence_num	The sequence number of the offending message.
error_class	The error class of the offending message.
severity	`IceCanContinue`, `IceFatalToProtocol`, or `IceFatalToConnection`

`values`	Any additional error values specific to the minor opcode and class.

Note that this error handler is invoked for protocol related errors. To install an error handler to be invoked when an IO error occurs, use `IceSetIOErrorHandler`. For further information, see the “Inter-Client Exchange Library” standard.

Chapter 7. Session Management Properties

Each property is defined by the SmProp structure:

```
typedef struct {
 char *name; /* name of property */
 char *type; /* type of property */
 int num_vals; /* number of values */
 SmPropValue *vals; /* the list of values */
} SmProp;

typedef struct {
 int length; /* the length of the value */
 SmPointer value; /* the value */
} SmPropValue;
```

The X Session Management Protocol defines a list of predefined properties, several of which are required to be set by the client. The following table specifies the predefined properties and indicates which ones are required. Each property has a type associated with it.

A type of SmCARD8 indicates that there is a single 1-byte value. A type of SmARRAY8 indicates that there is a single array of bytes. A type of SmLISTofARRAY8 indicates that there is a list of array of bytes.

Name	**Type**	**POSIX Type**	**Required**
SmCloneCommand	OS-specific	SmLISTofARRAY8	Yes
SmCurrentDirectory	OS-specific	SmARRAY8	No
SmDiscardCommand	OS-specific	SmLISTofARRAY8	No*
SmEnvironment	OS-specific	SmLISTofARRAY8	No
SmProcessID	OS-specific	SmARRAY8	No
SmProgram	OS-specific	SmARRAY8	Yes
SmRestartCommand	OS-specific	SmLISTofARRAY8	Yes
SmResignCommand	OS-specific	SmLISTofARRAY8	No
SmRestartStyleHint	SmCARD8	SmCARD8	No
SmShutdownCommand	OS-specific	SmLISTofARRAY8	No
SmUserID	SmARRAY8	SmARRAY8	Yes

* Required if any state is stored in an external repository (for example, state file).

- SmCloneCommand

  This is like the SmRestartCommand, except it restarts a copy of the application. The only difference is that the application does not supply its client ID at register time. On POSIX systems, this should be of type SmLISTofARRAY8.

- SmCurrentDirectory

  On POSIX-based systems, this specifies the value of the current directory that needs to be set up prior to starting the SmProgram and should of type SmARRAY8.

- SmDiscardCommand

  The discard command contains a command that when delivered to the host that the client is running on (determined from the connection), will cause it to discard any information about the current state. If this command is not specified, the Session Manager will assume that all of the client's state is encoded in the SmRestartCommand. On POSIX systems, the type should be SmLISTofARRAY8.

- SmEnvironment

  On POSIX based systems, this will be of type SmLISTofARRAY8, where the ARRAY8s alternate between environment variable name and environment variable value.

- SmProcessID

  This specifies an OS-specific identifier for the process. On POSIX systems, this should contain the return value of `getpid` turned into a Latin-1 (decimal) string.

- SmProgram

  This is the name of the program that is running. On POSIX systems, this should be first parameter passed to `execve` and should be of type SmARRAY8.

- SmRestartCommand

  The restart command contains a command that, when delivered to the host that the client is running on (determined from the connection), will cause the client to restart in its current state. On POSIX-based systems, this is of type SmLISTofARRAY8, and each of the elements in the array represents an element in the `argv` array. This restart command should ensure that the client restarts with the specified client-ID.

- SmResignCommand

  A client that sets the SmRestartStyleHint to `SmRestartAnyway` uses this property to specify a command that undoes the effect of the client and removes any saved state. As an example, consider a user that runs **xmodmap** which registers with the Session Manager, sets SmRestartStyleHint to `SmRestartAnyway`, and then terminates. To allow the Session Manager (at the user's request) to undo this, **xmodmap** would register a SmResignCommand that undoes the effects of the **xmodmap**.

- SmRestartStyleHint

  If the SmRestartStyleHint is present, it will contain the style of restarting the client prefers. If this style is not specified, `SmRestartIfRunning` is assumed. The possible values are as follows:

Name	**Value**
`SmRestartIfRunning`	0
`SmRestartAnyway`	1
`SmRestartImmediately`	2
`SmRestartNever`	3

The `SmRestartIfRunning` style is used in the usual case. The client should be restarted in the next session if it was running at the end of the current session.

The `SmRestartAnyway` style is used to tell the Session Manager that the application should be restarted in the next session even if it exits before the current session is terminated. It should be noted that this is only a hint and the Session Manager will follow the policies specified by its users in determining what applications to restart.

A client that uses `SmRestartAnyway` should also set the SmResignCommand and SmShutdownCommand properties to commands that undo the state of the client after it exits.

The SmRestartImmediately style is like SmRestartAnyway, but, in addition, the client is meant to run continuously. If the client exits, the Session Manager should try to restart it in the current session.

`SmRestartNever` style specifies that the client does not wish to be restarted in the next session.

- SmShutdownCommand

  This command is executed at shutdown time to clean up after a client that is no longer running but retained its state by setting SmRestartStyleHint to `SmRestartAnyway`. The client must not remove any saved state as the client is still part of the session. As an example, consider a client that turns on a camera at start up time. This client then exits. At session shutdown, the user wants the camera turned off. This client would set the SmRestartStyleHint to `SmRestartAnyway` and would register a SmShutdownCommand that would turn off the camera.

- SmUserID

  Specifies the user ID. On POSIX-based systems, this will contain the user's name (the *`pw_name`* member of struct passwd).

Chapter 8. Freeing Data

To free an individual property, use `SmFreeProperty`

`void` **`SmFreeProperty`**`(`*`prop`*`);`

`prop`	The property to free.

To free the reason strings from the `SmsCloseConnectionProc` callback, use `SmFreeReasons`

`void` **`SmFreeReasons`**`(`*`count`*`, `*`reasons`*`);`

`count`	The number of reason strings.
`reasons`	The list of reason strings to free.

Chapter 9. Authentication of Clients

As stated earlier, the session management protocol is layered on top of ICE. Authentication occurs at two levels in the ICE protocol:

- The first is when an ICE connection is opened.
- The second is when a Protocol Setup occurs on an ICE connection.

The authentication methods that are available are implementation-dependent (that is., dependent on the ICElib and SMlib implementations in use). For further information, see the “Inter-Client Exchange Library” standard.

Chapter 10. Working in a Multi-Threaded Environment

To declare that multiple threads in an application will be using SMlib (or any other library layered on top of ICElib), you should call `IceInitThreads`. For further information, see the “Inter-Client Exchange Library” standard.

Chapter 11. Acknowledgements

Thanks to the following people for their participation in the X Session Management design: Jordan Brown, Ellis Cohen, Donna Converse, Stephen Gildea, Vania Joloboff, Stuart Marks, Bob Scheifler, Ralph Swick, and Mike Wexler.