SIP API for Solaris Reference Guide

,

Part No:
May 5, 2006

060629@14986

Contents

Preface 7

1 SIP Overview 9
Introduction to SIP 9
Overview of `libsip` 9
Header Management Layer 10
Transaction Management Layer 10
Dialog Management Layer 10
Message Formatting Layer 10
Timer Management Layer 11
Connection Manager 11

2 Solaris SIP Internals 13
SIP Stack Initialization 13
Generic Stack Parameters 13
Upper Layer Registrations 13
Connection Manager Interfaces 14
Custom Header Table 15
Header Management Layer 15
Writing Parsers For Custom Headers 18
Transaction Management Layer 18
Transaction Creation And Maintenance 19
Transaction Creation and `ACK` Signal Generation 19
Transaction Deletion 19
Transaction Lookup 20
Transaction Timers 20
Transaction And Network Errors 20
Dialog Management Layer 20

UAC Dialog Creation 21
UAS Dialog Creation 21
Dialog Caching 21
Dialog Termination, Deletion, and Notification 22
Message Formatting Layer 22
Receiving Messages 22
Sending Messages 23
Connection Manager 23
Connection Object 24
Caching a Connection Object 24
Freeing a Connection Object 24
Sending Messages 25
Receiving Messages 25
Transaction Layer and I/O Errors 25
Timer Management Layer 25
Generating Call-ID, `From` and `To` tags, Branch-ID and `Cseq` 26
Multithreading Support 27

3 SIP API Functions 29
Stack Initialization Function 29
Message Allocation Functions 30
SIP Header Addition Functions 30
SIP Request and Response Creation Functions 37
Header and Message Copying Functions 39
Header and Value Deleting Functions 39
Header Lookup Functions 40
Value Retrieval and Response Description Functions 41
SIP ID Generating Functions 41
VIA Functions 42
SIP Message Sending Function 42
Processing Inbound Messages 43
Transaction Layer Functions 43
Dialog Layer Functions 45
URI Functions 48
SIP Header Value Retrieval Functions 48
Connection Object Functions 58

Miscellaneous Functions 60

4 Programming with the SIP API 63
Initializing the SIP Stack 63
Upper Layer Registrations 63
Connection Manager Interfaces 65
Custom SIP Headers 66
Example of UAS and UAC Use 67

A Transaction Timers 81

Preface

The *SIP API for Solaris Reference Guide* contains a description of the internals of the Solaris implementation of the Session Initiation Protocol (SIP). This book includes descriptions for the function calls in the API. This book also provides usage examples for the API.

Who Should Use This Book

This book is meant for use by application developers that are writing SIP applications.

Related Books

Programming Interfaces Guide

Documentation, Support, and Training

The Sun web site provides information about the following additional resources:

- Documentation (`http://www.sun.com/documentation/`)
- Support (`http://www.sun.com/support/`)
- Training (`http://www.sun.com/training/`)

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P–1 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file. **Note:** Some emphasized items appear bold online.

Shell Prompts in Command Examples

The following table shows the default UNIX® system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell	`machine_name%`
C shell for superuser	`machine_name#`
Bourne shell and Korn shell	`$`
Bourne shell and Korn shell for superuser	`#`

◆ ◆ ◆ CHAPTER 1

SIP Overview

This chapter serves as a brief introduction to the Session Initiation Protocol (SIP) and the library that implements this protocol in the Solaris™ operating system.

Introduction to SIP

The Session Initiation Protocol enables Internet endpoints to negotiate the character of a shared multimedia session. SIP is a tool that can create, modify, and terminate multimedia sessions. SIP works independently of any underlying transport protocols.

Overview of `libsip`

Applications that use SIP must initialize the Solaris SIP stack by registering certain mandatory and optional functions. These functions enable message exchanges and related functionalities between peers. If the application does not register any of the optional functions, the library uses the built-in equivalent when such an equivalent is available.

The SIP library is multithread-safe. Multiple threads can perform simultaneous operations on message headers. Each message maintains a reference count. The stack frees a message when the reference count is zero. The library does not protect messages against the case where an application thread reduces the reference count to 0. In this case, the stack can delete the message while other threads are using the message.

The SIP stack consists of the following operational components:

- Header management layer
- Transaction management layer
- Dialog management layer
- Message formatting layer
- Timer management layer
- Connection manager

Header Management Layer

The header management layer provides the interfaces that are required to build, parse, examine and validate SIP headers.

Transaction Management Layer

SIP is based on a request/response transaction model that is similar to the model that is used by HTTP. A transaction comprises a request that is sent by a client to a server and all of the responses to that request that are sent back from the server. The transaction management layer handles application-level retransmissions, matching responses to requests, and application-level timeouts. A User Agent Client (UAC) performs tasks by using a series of transactions.

Dialog Management Layer

A dialog is a persistent peer-to-peer SIP relationship between two user agents. A dialog facilitates the sequencing of messages and the proper routing of requests between the user agents. The use of dialogs is optional. Dialogs hold state information, which can be used to construct a request within a session. An application can implement its own dialog management layer instead of using the layer that the SIP stack provides. If the stack manages the dialog, it automatically creates and updates dialogs for incoming and outgoing SIP messages. The stack also delivers the matching dialog to the application with the incoming message.

Message Formatting Layer

The message formatting layer constructs a SIP message for an incoming message before delivering the message to the application. If the underlying transport protocol is TCP, this layer also breaks the byte stream at SIP message boundaries. The message formatting layer determines the SIP message boundaries by using the Content-Length header. If a transaction exists, the stack passes the message to the transaction layer for further processing, then passes the message to the application. If the dialog management layer is present, the stack passes the message to the dialog management layer, then passes the message to the application. If neither a transaction nor a dialog management layer are present, the stack passes the message to the application directly. For outbound messages, the message formatting layer adds a Content-Length header that is followed by an empty line, meeting the requirements of RFC 3261. The stack passes the message to the application's send function after passing the message to the transaction layer and dialog layer, if either is present. The stack sends the SIP message to the application's send routine after the stack copies all of the message's headers and contents into a contiguous buffer.

Timer Management Layer

The SIP stack uses several timers. The timer management layer provides timeout and un-timeout routines for these timers. The application can implement its own timeout/un-timeout routines and register them with the stack instead of using the routines that are provided by the library.

Connection Manager

The connection manager provides input/output functionality. The connection manager is not a part of the SIP library, but interacts with the library by using well defined interfaces. A connection is identified by two endpoints, one local and one remote, as well as by the transport protocol that the connection uses. A connection is represented within the stack by a connection object. The stack requires that the first member of the connection object be a void pointer. The stack uses this first member to store data that is related to the connection object. The application must initialize the connection object. The application is responsible for the following functions:

Compliance with RFC 3263	The library expects the application to supply the address of the next recipient of the SIP message and to handle network errors. The application must follow the procedures defined in RFC 3263 for locating SIP servers.
Sent-By and Received parameters	Section 18.1.1 of RFC 3261 states that the transport layer must add a "sent-by" parameter to the topmost VIA header in every request. The application must include this information when constructing the VIA header. The application must register all values it could use for the sent-by parameter with the stack to enable the stack to validate incoming responses as required by section 18.1.2 of RFC 3261. Section 18.2.1 of RFC 3261 lists the conditions under which the transport layer must add a "received" parameter to the top VIA header of incoming messages. This "received" header must indicate the source of the message. The stack caches the original transaction-creation request in the created transaction's connection object, and does not add a received parameter. The stack caches the connection object that it used to send the transaction creating requestTo satisfy the requirement of the RFC, the stack attempts retransmissions within a transaction by using the cached connection object before using the connection object that is passed in. This fallback occurs in the case of a network error.
TTL and `maddr` parameters	The application should add the TTL and `maddr` parameters in the VIA header when required.

◆ ◆ ◆ CHAPTER 2

Solaris SIP Internals

The following sections describe various components of the stack in detail. The interfaces described in this section are internal to the library unless otherwise specified.

SIP Stack Initialization

An application initializes the stack before performing any other tasks. The initialization parameters can be broadly subdivided into the following four types:

- Generic stack parameters
- Upper layer registrations
- Connection manager interfaces
- Custom header table

After initializing the stack, an application creates requests by using the interfaces that are provided by the header management layer. Applications send requests by using the message formatting layer's interface. Applications receive incoming requests and responses after initializing the stack and pass the requests and responses to the stack for processing.

Generic Stack Parameters

SIP version — SIP_STACK_VERSION defines the SIP stack version.

Stack flags — SIP_STACK_DIALOGS instructs the stack to maintain dialogs. If this flag is not set, the stack does not maintain any dialog information.

Upper Layer Registrations

Upper layer receive routine
: The stack uses this function to deliver SIP messages to the application. Applications must register this routine.

Application specific timeout and un-timeout routines
: The application can define timeout and un-timeout routines for the stack to use. Applications must define both routines or neither routine. If the application does not define these routines, the stack uses the built-in timeout routines.

Transactions error notifications
: The application can register an error notification routine. The stack invokes the error notification routine when the transaction layer encounters a network error while the transaction layer is sending a SIP message. If the application does not register an error notification routine, the stack will not send notifications in case of a network error.

Dialog delete notification
: If the stack is configured to maintain dialogs, the application can register a callback routine. The stack invokes the callback routine when a dialog is deleted. If the application does not register a callback routine, the stack will not send notifications when a dialog is deleted.

Transaction state transition notifications
: An application can register a routine that is invoked when a transaction changes state. The routine is invoked with the transaction handle, the message that resulted in the transition, and the previous and current states. If an application does not register this routine, the stack will not send notifications when a transaction changes state.

Dialog state transition notifications
: If the stack is configured to maintain dialogs, the application can register a callback routine. The stack invokes the callback routine when a dialog changes state. The callback routine is invoked with the dialog handle, the message that resulted in the transition, and the previous and current state. If an application does not register this routine, the stack will not send notifications when a dialog changes state.

Connection Manager Interfaces

Applications must register the interfaces in this section with the SIP library.

Send routine	The stack calls this routine to send SIP messages.
Hold/Release functions	The application must provide functions to increment and decrement reference counts on a connection object.
Connection attributes.	The application must register functions to query the following attributes of a connection object:

`is_stream`	The connection is a byte-stream
`is_reliable`	The connection is reliable
remote address	Remote endpoint information
local address	Local endpoint information
transport	The transport type

The application can register the following timer attributes that the API uses to obtain timer values for a connection object:

Timer1 This value defines the RTT estimate

Timer2 This value defines the maximum retrieval interval for non-INVITE requests and INVITE responses

Timer4 This value defines the maximum duration that a message remains in the network

TimerD This value defines the wait time for response retransmits

If an application does not specify values for these timer attributes, the stack uses default values.

Custom Header Table

An application can register a table of custom headers with the library. The table must include parsing functions for those headers. The application can include standard headers in the table. If the application includes standard headers in the custom header table, the stack uses the parsing functions that are defined in the custom header table instead of the built-in parsing functions.

Header Management Layer

This layer provides interfaces that enable an application to create, parse, modify, and examine SIP messages. A SIP message consists of a start line, a variable number of headers, and an optional message body. The start line and headers are terminated by a single Carriage Return/Line Feed (CRLF) character. The message body is preceded by an empty line that contains only a CRLF. SIP allows the combination of multiple headers of the same name type under one header. A single header can have multiple values. A header value consists of a number of descriptive parameters and an optional name-value pair. The Backus-Naur Form (BNF) for a VIA header is defined in section 24 of RFC 3261 as the following:

```
via        = ("Via" / "v") HCOLON via-parm * (COMMA via-parm)
via-parm   = sent-protocol LWS sent-by *(SEMI via-params)
via-params = via-ttl / via-maddr / via-branch / via-received / via-extension
```

The values of `via-parm` are descriptive parameters. The values of `via-params` are name-value pairs. The combination of `via-parm` and `via-params` constitute the value of the header. The stack maintains a `sip_`*header_name*`_value_t` structure for each SIP header. In the case of a VIA header, the structure is `sip_via_value_t`. A header can have multiple values separated by a comma. A header with multiple values lists a `sip_`*header_name*`_value_t` structure for each value. For example, a SIP header with multiple VIA headers would list multiple `sip_via_value_t` structures, with a corresponding `via-parm` parameter for each structure.

The SIP message contains a list of all the headers and the content. The stack uses the receive function that the application registered with the stack to pass the message to the application.

SIP messages implement a reference count mechanism. An application can hold a reference to a SIP message by using the `sip_hold_msg()` function. An application can release a reference to a SIP message by using the `sip_free_msg()` function. The stack destroys messages whose reference count drops to zero when the application calls the `sip_free_msg()` function.

The stack can forward a SIP message to a SIP entity directly or after modifying the message. The stack cannot modify a message after sending it.

```
typedef struct sip_header_general {
        char                            *sip_hdr_start;
        char                            *sip_hdr_end;
        char                            *sip_hdr_current;
        sip_parsed_header_t           *sip_hdr_parsed;
}sip_hdr_general_t;

typedef struct header_function_table {
        char                *header_name;
        char                *header_short_name;
        int                 (*header_parse_func)(struct sip_header *,
                                    struct sip_parsed_header **);
        boolean_t           (*header_check_compliance)(struct sip_parsed_header *);
        boolean_t           (*header_is_equal)(struct sip_parsed_header *,
                                    struct sip_parsed_header *);
        void                (*header_free)(struct sip_parsed_header *);
}sip_header_function_t;
```

The SIP header contains a `sip_header_general_t` structure that provides pointers to the start and end of the header string. The structure also has a pointer to the current position in the string that a parser can use when it is processing the header. When the stack parses a header for the first time, it caches the result in `sip_hdr_parsed` for future reference.

SIP allows the inclusion of custom message headers that are not defined in RFC 3261. An application can provide a table of custom headers and parsing functions to support those custom headers. The application provides an array, `sip_header_function_t`, for this purpose. The table entries that the application provides override the header entries that are in the default function table.

The library maintains a table that has an entry for each header that is defined in RFC 3261. This table specifies the header name in long form, as well as the compact form where present. The table also specifies the associated parsing function and any other functions listed in the `sip_header_function_t` array.

When the application deletes a header, the value of the `sip_header_state` attribute is set to `SIP_HEADER_DELETED`. Existing references to the header remain, but new lookups will not return the deleted header. The library frees all of the message headers at the same time that the library frees the message.

The following structure represents a parsed SIP header:

```
typedef struct sip_parsed_header {
        int                  sip_parsed_header_version;
        struct sip_value     *value;
        sip_header_t        sip_header;
}sip_parsed_header_t;
```

The member value is a pointer to a `sip_`*header_name*`_value_t` structure. The stack sets the structure as a result of parsing the SIP header. The first field of the `sip_`*header_name*`_value_t` structure is always a `sip_value_t` structure. The `sip_value_t` structure is defined in the following code:

```
typedef struct sip_value {
        int                         sip_value_version;
        void                        *next;
        sip_param_t                 *param_list;
        sip_value_state_t           value_state;
        sip_parsed_header_t         *parsed_header;
        char                        *value_start;
        char                        *value_end;
        sip_str_t                   *sip_value_uri_str;
        sip_uri_t                   sip_value_parse_uri;
}sip_value_t;
```

The values in the `param_list` array is the list of parameters for the value that the system is parsing. The `sip_param_t` structure is defined in the following code:

```
typedef struct sip_param {
        sip_str_t               param_name;
        sip_str_t               param_value;
        struct sip_param         *param_next;
}sip_param_t;


typedef struct sip_str {
        char             *sip_str_ptr;
        int             sip_str_len;
}sip_str_t;
```

When the library deletes a value from a header, the library sets the value of the `sip_header_state` attribute to `SIP_HEADER_DELETED_VAL`. The library also sets the value of the `value_state` attribute to `SIP_VALUE_DELETED`. Existing references to the value remain. New lookups ignore the deleted value.

Traverse the following hierarchy to get to a value from a SIP message:

```
sip_msg_t -> sip_header_t -> sip_parsed_header_t -> sip_<header_name>_value_t
```

Using the free function that corresponds to a given structure also frees all of the structures that are under it in the hierarchy.

The library performs lazy header parsing. The library only parses or checks for compliance headers whose values are used by the application. This behavior complies with section 16.3 of RFC 3261, *Request Validation* and *Reasonable syntax check*. When the parser encounters an invalid value, the parser sets the `value_state` attribute to `SIP_VALUE_BAD`.

Writing Parsers For Custom Headers

An application can define custom headers or specialized versions of standard headers. Applications that define headers must register parsing functions with the library to support those headers. Use the following structures to write parsing functions:

`sip_header_general_t`	This structure provides the start and end of a header within a message. The parser sets the parsed header element to point to the `sip_parsed_header_t` structure that the parser allocates.
`sip_parsed_header_t`	The parser allocates and assigns this structure. The parser passes this structure back to the caller.
`sip_value_t`	The application defines the value of this structure depending on the header. The first element of the structure must be `sip_value_t`. The parser creates a linked list of values if multiple values exist. The parser sets the value of the value field in the parsed header to the first value. The application must provide any access function that value members require.

Transaction Management Layer

The transaction management layer creates and maintains transaction states for both clients and servers. The transaction management layer complies with Section 17 of RFC 3261.

The stack maintains a hash table of transactions. The index of this hash table is an MD5 hash. For messages that comply with RFC 3261, the MD5 hash is the hash of the branch ID. For messages that comply with RFC 2543, the MD5 hash is of a combination of the branch ID, Call-ID, From, To, and Cseq fields. The stack uses the branch ID in the topmost VIA header to identify the RFC that a message complies with. Messages that comply with RFC 3261 prefix their branch ID with the string `z9hG4bK`.

The library initializes the transaction layer as part of the stack initialization. The library also initializes the transaction error and state transition callback functions if the application provides them.

Transaction Creation And Maintenance

The stack scans the transaction hash table for an existing transaction when the stack receives incoming requests and responses. If a transaction that matches an incoming response does not exist, the stack handles the transaction statelessly. When an incoming request requires the stack to create a transaction, the application notifies the stack of the need to create a transaction when the application sends the response. In both cases, the stack passes the message to the application. If a request matches a transaction, the stack considers the request a retransmission and the transaction layer retransmits the last response. The stack drops the incoming request and does not pass the request to the application. The `CANCEL` request and an `ACK` for a non-2xx response are exceptions to this behavior. A `CANCEL` request matches the INVITE transaction that it is cancelling. An `ACK` for a non-2xx response also matches the INVITE request whose response the `ACK` is acknowledging. In these two cases, the request is not a retransmission and the stack sends the request sent to the application. If a response matches a transaction, the stack processes the response and statefully passes the message to the application.

The stack only creates a transaction for an outgoing response when the application requests the transaction creation by using the `sip_sendmsg()` function. The stack only creates a transaction for outgoing requests if the application requests the transaction creation by using the `sip_sendmsg()` function. The stack caches the transaction creating message and the last message that was sent on a transaction. This cache facilitates retransmissions and the processing of retransmitted requests. The stack initializes timers A, B, D, E, F, G, H, I, J and K for new transactions. See Appendix A for more information about timers.

When a message causes a transaction to change states, the stack invokes a callback function if the application registered a callback function with the stack.

Transaction Creation and ACK Signal Generation

A SIP entity must send an `ACK` signal for each final response that the SIP entity receives to an `INVITE` request. The procedure for sending the `ACK` signal depends on the type of response. For final responses between 300 and 699, the transaction layer handles the `ACK` signal processing. For 2xx responses, the application generates the `ACK` signal. In all cases, the application receives the response.

Transaction Deletion

The MD5 hash that is stored in the transaction is used to look up and delete the transaction from the hash table. When a transaction enters a terminated state, the transaction management layer destroys

the transaction if the reference count drops to zero. If the application registered a callback function during initialization, that callback function notifies the application when the transaction changes state. If the application needs to use the transaction in the future, it must hold a reference to the transaction by using the `sip_hold_trans()` function to prevent the stack from destroying the transaction. After using the transaction the application must release the reference by using the `sip_release_trans()` function.

The four terminated states for transactions are listed below:

- Client invite terminated
- Client non-invite terminated
- Server invite terminated
- Server non-invite terminated

Transaction Lookup

An application can look up a transaction for a SIP message by using the `sip_get_trans()` function. A successful lookup returns the transaction and increments the transaction's reference count. The application must release the transaction by using the `sip_release_trans()` function after use.

Transaction Timers

The transaction layer maintains a set of timers for timing out transactions or sending retransmissions. Appendix A lists these timers. If the application initializes timeout reporting, the transaction layer notifies the application when a transaction times out.

Transaction And Network Errors

The transaction layer uses a cached connection object to send messages. In the event of a network error, the transaction layer releases the connection object and calls the error callback function, if the application provided one. If the callback function not does return a value of 0, or if the application does not provide a callback function, the stack terminates the transaction and frees the associated resources.

Dialog Management Layer

A dialog represents a persistent peer-to-peer relationship between two user agents. A dialog facilitates the sequencing of messages between the user agents and the proper routing of requests between the user agents. The dialog represents a context in which to interpret SIP messages. A dialog is uniquely identified by a dialog id, which is a combination of the `Call-ID`, `From`, and `To` tags.

Dialog use is optional. You can create dialogs using `INVITE` and `SUBSCRIBE` as methods. A response to an `INVITE` in the 101–299 range combined with a `To` tag creates a dialog. A response to a `SUBSCRIBE` in the 200–299 range or a `NOTIFY` request creates a dialog. Provisional responses create dialogs that are marked as `EARLY` dialogs. Final responses in the 200–299 range create dialogs that are marked as `CONFIRMED` dialogs. An `EARLY` dialog becomes a confirmed dialog when the stack receives a response in the 200–299 range. If the application registers a callback function with the stack to send notification when a dialog changes state, the stack invokes that function.

An application can perform dialog management internally, or delegate dialog management to the stack. If the stack manages dialogs, the stack automatically creates and maintains dialogs. The stack also delivers any dialogs, matching any incoming messages to the application. If the stack is not managing dialogs, there is no dialog-related interaction between the application and the stack.

UAC Dialog Creation

For clients, if the stack is configured to maintain dialogs, the stack creates a dialog for an incoming response to a dialog creating request. Dialog creating requests are either `INVITE` or `SUBSCRIBE`. For `SUBSCRIBE` requests, a corresponding `NOTIFY` request also creates a dialog. The stack can create multiple dialogs for a single request if the application specifies multiple dialogs by using the `sip_sendmsg()` function.

For servers, if the stack is configured to maintain dialogs, the stack creates a dialog when the application sends a response to the dialog creating request. The stack also creates a dialog when it sends a `NOTIFY` request in response to a `SUBSCRIBE` request.

UAS Dialog Creation

When the stack receives a dialog creating request, it creates a partial dialog by using the request. The stack sends the partial dialog to the application along with the request. The stack completes the dialog when the application sends a response. Because the stack does not insert the partial dialog into a hash table, the stack does not return partial dialogs as results of a lookup. The stack starts a timer when it creates the partial dialog. The timer's duration is set to the duration of an `INVITE` transaction timeout. If the UAS does not respond during this time interval, the stack deletes the partial dialog. If the application registers a callback function with the stack for dialog deletion notifications, the stack invokes the callback function before deleting the partial dialog. The stack also deletes the partial dialog if the response is outside the 100–299 range.

Dialog Caching

For an incoming message, the stack sends any existing dialog, along with the message to the application. When the application's receive function returns, the stack decrements the reference count to the dialog. If the dialog's reference count is zero after this decrement, the stack destroys the

dialog if the dialog is in a terminated state. If the application needs to use the dialog in the future, it must hold a reference to the dialog by using the `sip_hold_dialog()` function to prevent the stack from destroying the dialog After using the dialog the application must release the reference by using the `sip_release_dialog()` function.

Dialog Termination, Deletion, and Notification

When the stack processes a SIP message, the processing can result in the termination of the dialog. A dialog marked as `EARLY` terminates if the final response is not in the 200–299 range, or if no response arrives. The termination mechanism for confirmed dialogs are method specific. The `BYE` method terminates an `INVITE` dialog and the session that is associated with it. An application can explicitly delete a dialog by using the `sip_delete_dialog()` function.

When the stack terminates a dialog, it notifies the application if the application has registered a callback function with the stack for this purpose.

Message Formatting Layer

The message formatting layer represents the message in the form that the next component or layer requires.

If the incoming message arrives over TCP, the message formatting layer breaks the byte stream at message boundaries. The message formatting layer parses the message in order to represent the message as a SIP message.

When the application is sending a message, the message formatting layer copies all of the message's headers and contents into a contiguous buffer before delivering the message to the application's send function.

Receiving Messages

The application sends the incoming message to the stack along with the connection object. If the transport is TCP, the message formatting layer breaks the byte stream at message boundaries. The SIP message's Content-Length header determines the message boundaries. A Content-Length header must be present in every message delivered over TCP in order to comply with RFC 3261. If the stack receives a message over TCP that does not contain a Content-Length header, the stack's behavior is unspecified.

The message formatting layer holds excess data that is not part of the current message for use with the next packet. When the system closes or reuses a connection, the connection manager must notify the stack. After receiving the notification, the stack frees the data and resources that the stack

allocated to that connection. The stack delivers the message to the application by calling the receive function. The application registers the receive function with the stack during stack initialization.

Sending Messages

An application sends a SIP message by using the `sip_sendmsg()` function with the connection object and any message specific flags. The message formatting layer adds a Content-Length header and a line that contains only a Carriage Return/Line Feed (CRLF) character to the message and delivers the packet to the next layer. The message formatting layer copies all of the SIP message's headers and contents into a contiguous buffer before delivering the message to the application's send function. The connection manager provides the send function.

Connection Manager

The connection manager provides I/O functionality. The connection manager is not part of the library but interacts with the stack using well defined interfaces. This section describes the usage model of the connection manager, its interface with the stack, and the requirements that are imposed by the library. The connection manager must register the following mandatory interfaces with the stack as part of stack initialization:

```
int          sip_conn_send(const sip_conn_object_t, char *, int);
void         sip_hold_conn_object(sip_conn_object_t);
void         sip_rel_conn_object(sip_conn_object_t);
boolean_t    sip_conn_is_reliable(sip_conn_object_t);
boolean_t    sip_conn_is_stream(sip_conn_object_t);
int          sip_conn_remote_address(sip_conn_object_t,
                    struc sockaddr *, socklen_t *);
int          sip_conn_local_address(sip_conn_object_t,
                    struct sockaddr *, socklen_t *);
int          sip_conn_transport(sip_conn_object_t);
```

If the application uses timer values that are specific to a connection object, the application must register the following interfaces to provide those values for Timer 1, Timer 2, Timer 4, and Timer D:

```
int        sip_conn_timer1(sip_conn_object_t);
int        sip_conn_timer2(sip_conn_object_t);
int        sip_conn_timer4(sip_conn_object_t);
int        sip_conn_timerd(sip_conn_object_t);
```

A connection object represents a connection. A connection is identified by the local endpoint, the remote endpoint, and the transport.

The library requires that the first element of the connection object is a void pointer. The stack reserves this void pointer for its own use. The application must initialize each connection object by calling the `sip_init_conn_object()` function before using the object. The connection object is opaque to the stack.

Connection Object

When the stack initializes a connection object, the stack sets the first element of a connection object to point to a structure. The structure tracks transactions that cache the connection object. The stack also uses the structure to break a TCP stream at the message boundaries.

Because the library does not interact with listening endpoints, it does not impose any restriction on creating or maintaining listening endpoints.

The connection manager maintains a unique connection object for each remote address, local address, and transport tuple. This behavior is particularly important for UDP, where the underlying endpoint does not uniquely identify a local/remote endpoint pair.

Caching a Connection Object

The stack caches connection objects for use by the transaction layer. The stack increments the reference count on the connection object to prevent the connection object from being freed by the system when the connection object is in use by the stack. When the transport protocol is TCP, the stack adds a reference count on the connection object when the stack allocates resources to break the TCP stream at message boundaries. The connection manager registers the `sip_hold_conn_object()` and `sip_rel_conn_object()` functions for the purposes of holding and releasing connection objects.

Freeing a Connection Object

The connection manager can close a connection at any time. The connection manager cannot free the connection object while references to the connection object exist. To free a connection object that has existing references, the connection manager calls the `sip_conn_destroyed()` function. The library provides this function. When the connection manager invokes the `sip_conn_destroyed()` function, the stack locates and terminates the transactions that are caching the connection object. The stack frees any resources that had been allocated for TCP handling after terminating the transactions.

Sending Messages

When an application sends a message by using the `sip_sendmsg()` function, the stack calls the `sip_conn_send()` function. The connection manager registers the `sip_conn_send()` function with the stack. The stack calls the `sip_conn_send()` function after the stack processes the outbound message. In case of a network error, the application can define the response of the connection manager, including returning an error. The connection manager's behavior must be consistent for all subsequent calls that use the same connection object. The connection manager can create a new connection to update an existing connection. The connection manager must flush any stale TCP data that is left in the connection object as a result of breaking the TCP stream at message boundaries. The connection manager uses the library function `sip_clear_stale_data()` to flush any stale TCP data. The Connection Manager must not change the transport type of a connection object.

Receiving Messages

The Connection Manager delivers new packets to the stack by calling the `sip_process_new_packet()`. After processing the message, the stack calls the receive callback function to pass the message to the application. The application must register the receive callback function with the stack at initialization. The stack frees the message and releases the connection object when the application's receive function returns. To queue a message, an application must manage the reference counts on the connection object as described in the Caching Connection Objects section. The application uses the `sip_hold_msg()` and `sip_free_msg()` functions to manage a SIP message's reference counts.

Transaction Layer and I/O Errors

The transaction layer uses a cached connection object to send messages such as retransmitting requests or responses. In the event of a network error, the transaction layer releases the connection object. If the application registered a callback function for transaction error notification, the transaction layer invokes that function. If the application did not register a callback function for transaction error notification, or if the callback function returns a nonzero value, the stack terminates the transaction. If the callback function returns a value of zero, the stack does not release the cached object.

Timer Management Layer

The timer management facility mimics the timeout and un-timeout functionality of the UNIX kernel. A thread maintains a list of timer objects sorted by time. The thread invokes the callback function at the interval that the timer specifies.

The stack initializes the timer layer during stack initialization. Initializing the timer layer starts the timer thread.

The sip_timeout() function takes the following arguments:

- The callback function to use
- The time interval to wait before invoking the callback function
- The argument to pass to the callback function

The return value of the timeout function is a timeout identifier. The tack can cancel a timeout by passing the timeout ID of the timeout to cancel to the sip_untimeout() function

Generating Call-ID, From and To tags, Branch-ID and Cseq

The library provides the sip_guid() function to generate unique identifiers for the Call-ID, From, and To tags. The stack generates the identifier by combining the upper 32 bits that are returned by the gethrtime() function with a 32–bit random number from the /dev/urandom pseudo-device. The stack randomly replaces numbers in the result with alphabetic characters. The application is responsible for freeing the string that is returned by the sip_guid() function.

The library provides the sip_branchid() function to generate Branch-ID fields. If the application invokes the sip_branchid() function without a SIP message, or if the SIP message does not have a VIA header, the stack forms a Branch-ID field by using the sip_guid() function. If the SIP message already has a VIA header, the stack generates the Branch-ID by using the MD5 hash of the following fields:

- To tag
- From tag
- Call-ID
- Request URI
- Topmost VIA header
- Sequence number from the Cseq header

All branch identifiers begin with the string z9hG4bK. This behavior complies with RFC 3261. The application is responsible for freeing the string that is returned by the sip_branchid() function.

An application can use the sip_get_cseq() or sip_get_rseq() functions to obtain the initial sequence number. The sip_get_cseq() and sip_get_rseq() functions use the 31 most significant bits of the value that is returned by the time(2) function call.

Multithreading Support

The library is completely multithreaded with respect to handling headers and header values. Multiple application threads can work on the same header of a message. When the stack or the application deletes a header or one of the values in a header, the stack marks the header as deleted. Headers marked as deleted are not available via lookups.

Calling the `sip_free_msg()` function to free a SIP message deletes the message if the reference count for the message falls to zero. You can increment the reference count on a message by using the `sip_hold_msg()` function.

◆ ◆ ◆ CHAPTER 3

SIP API Functions

This section describes the functions in the SIP API.

Stack Initialization Function

```
int sip_stack_init(sip_stack_int_t *stack_init);
```

This function initializes the SIP stack. This is the stack initialization structure:

```
typedef struct sip_stack_init_s {
        int                     sip_version;
        uint32_t                sip_stack_flags;
        sip_io_pointers_t       *sip_io_pointers;
        sip_ulp_pointers_t      *sip_ulp_pointers;
        sip_header_function_t   *sip_function_table;
}sip_stack_init_t;
```

The structure has the following elements:

`sip_version`	The value of this variable is equal to the value of the `SIP_STACK_VERSION` attribute.
`sip_stack_flags`	Set the value of this variable to `SIP_STACK_DIALOGS` to make the stack manage dialogs.
`sip_io_pointers`	The values in this field provide the connection manager interfaces.
`sip_ulp_pointers`	The values in this field contain the upper layer registrations. Upper layer registrations are functions that the application must register with the stack during initialization.
`sip_function_table`	The values in this field are the custom headers that the application registers and the parsing functions that are associated with those headers. The application can specify the following elements for each custom header:

- `header_name`
- `header_short_name` (optional)
- `header_parse_func`
- `header_check_compliance` (optional)
- `header_is_equal` (optional)
- `header_free`

Message Allocation Functions

```
sip_msg_t sip_new_msg();
```

The `sip_new_msg()` function allocates and returns a SIP message.

```
void sip_free_msg(sip_msg_t sip_msg);
```

The `sip_free_msg()` function frees the resources that are associated with a SIP message. The function decrements the reference count. If the decremented reference count is zero, the function destroys the SIP message. If the decremented reference count is not zero, the function does not destroy the SIP message until the last thread that holds a reference to the message releases that reference.

```
void sip_hold_msg(sip_msg_t sip_msg);
```

Threads use the `sip_hold_msg()` function to hold a reference to the SIP message. A SIP message persists while the number of references to the message is greater than zero.

SIP Header Addition Functions

All of the functions that are described in this section, with the exception of the `sip_add_param()` function, take the SIP message as the first argument. These functions also take header specific attributes as arguments.

```
int sip_add_header(sip_msg_t sip_msg, char *header_string);
```

The `sip_add_header()` function takes a SIP header string as the second argument and appends it to the SIP message. This function creates the SIP header by appending a CRLF to the value provided in the *header_string* parameter, then adding it to the end of the SIP message.

```
sip_header_t sip_add_param(sip_header_t sip_header, char *param, int *error);
```

The `sip_add_param()` function adds the value in the *param* parameter to the SIP header in the first argument. The value of the *param* parameter cannot be null. The function returns the header with the parameter added.

Note – Adding a parameter with this function creates a new header and marks the previous header as deleted. Developers working on multi-threaded applications should monitor headers that are being modified by multiple threads simultaneously.

All of the following header adding functions add a CRLF to the header before adding the header to the SIP message.

`int` **`sip_add_from`**`(sip_msg_t` *`sip_msg`*`, char *`*`display_name`*`, char *`*`from_uri`*`, char *`*`from_tag`*`, boolean_t` *`add_aquot`*`, char *`*`from_param`*`);`

The `sip_add_from()` function appends a `FROM` header to a SIP message. This function creates the header using the *display_name*, *from_uri* and *from_tag* parameters. The calling thread must provide the URI in the *from_uri* parameter. If the calling thread provides the value of the *display_name* parameter, the value of the *add_aquot* parameter must be `B_TRUE`. This value of the *add_aquot* parameter encloses URI within the delimiting characters < and >. If the *display_name* parameter has a value, this function quotes that name within " characters and adds the quoted name to the header. If the values in the *from_tag* parameter are `NULL`, the resulting SIP header does not contain a `TAG` parameter. You can specify any generic parameter in a *from_param* parameter.

Note – Only one of the pair of parameters *from_tag* or *from_param* can have a non-*NULL* value. If both parameters have non-*NULL* values, the resulting *FROM* header contains only the *TAG* parameter.

`int` **`sip_add_to`**`(sip_msg_t` *`sip_msg`*`, char *`*`display_name`*`, char *`*`to_uri`*`, char *`*`to_tag`*`, boolean_t` *`add_aquot`*`, char *`*`to_param`*`);`

The `sip_add_to()` function appends a `TO` header to a SIP message. This function creates the header using the *display_name*, *to_uri* and *to_tag* parameters. The calling thread must provide the URI in the *to_uri* parameter. If the calling thread provides the value of the *display_name* parameter, the value of the *add_aquot* parameter must be `B_TRUE`. This value of the *add_aquot* parameter encloses the URI within the delimiting characters < and >. If the *display_name* parameter has a value, this function quotes that name within " characters and adds the quoted name to the header. If the values in the *to_tag* parameter are `NULL`, the resulting SIP header does not contain a `TAG` parameter. You can specify any generic parameter in a *to_param* parameter.

Note – Only one of the pair of parameters *to_tag* or *to_param* can have a non-*NULL* value. If both parameters have non-*NULL* values, the resulting *TO* header contains only the *TAG* parameter.

`int` **`sip_add_contact`**`(sip_msg_t` *`sip_msg`*`, char *`*`display_name`*`, char *`*`contact_uri`*`, boolean_t` *`add_aquot`*`, char *`*`contact_param`*`);`

The `sip_add_contact()` function appends a `CONTACT` header to a SIP message. This function creates the header using the *display_name* and *contact_uri* parameters. The calling thread must provide the URI in the *contact_uri* parameter. If the calling thread provides the value of the *display_name* parameter, the value of the *add_aquot* parameter must be `B_TRUE`. This value of the *add_aquot*

parameter encloses the URI within the delimiting characters < and >. If the *display_name* parameter has a value, this function quotes that name within " characters and adds the quoted name to the header. You can specify any generic parameter in a *contact_param* parameter.

```
int sip_add_via(sip_msg_t sip_msg, char *protocol_transport, char *sent_by_host, int
sent_by_port, char *via_params);
```

The `sip_add_via()` function appends a VIA header to the SIP message that is specified by the value of the *sip_msg* parameter. The VIA header has the transport protocol that is specified in the *protocol_transport* parameter. The VIA header has the IP address or hostname that is specified by the *sent_by_host* parameter. The VIA header has the port that is specified in the *sent_by_port* parameter. If the *sent_by_port* parameter has a value of zero, the resulting VIA header does not have any port information. The VIA header's protocol is set to `SIP` and the version is set to `2.0`. Specify any header-specific parameters in the *via_params* parameter. For example, you can specify the branch parameter in the *via_params* parameter with the string `"branch=x9hG4b-mybranchid"`.

```
int sip_add_branchid_to_via(sip_msg_t sip_msg, char *branchid);
```

The `sip_add_branchid_to_via()` function adds a branch parameter to the topmost VIA header in the SIP message. The branch parameter cannot already exist in the VIA header.

Note – Adding a parameter with this function creates a new header and marks the previous header as deleted. Developers working on multi-threaded applications should monitor headers that are being modified by multiple threads simultaneously.

```
int sip_add_maxforward(sip_msg_t sip_msg, uint_t maxforward);
```

The `sip_add_maxforward()` function appends a `Max-Forwards` header. The value of the header is equal to the value of the *maxforward* parameter.

```
int sip_add_callid(sip_msg_t sip_msg, char *callid);
```

The `sip_add_callid()` function appends a `Call-ID` header to the SIP message by using the value of the *callid* parameter. If the *callid* parameter has a value of `NULL`, the `sip_add_callid()` function generates a random value for the `Call-ID` header.

```
int sip_add_cseq(sip_msg_t sip_msg, sip_method_t method, uint32_t cseq);
```

The `sip_add_cseq()` function appends a `CSeq` header using the method and the value of *cseq* provided. The *method* can take on the following values:

- `INVITE`
- `ACK`
- `OPTIONS`
- `BYE`
- `CANCEL`
- `REGISTER`
- `REFER`

- INFO
- SUBSCRIBE
- NOTIFY
- PRACK

The value of the *cseq* parameter must be a positive integer.

```
int sip_add_content_type(sip_msg_t sip_msg, char *, char *);
```

The `sip_add_content_type()` function adds a `Content-Type` header using the type and subtype specified in the arguments. The type and subtype cannot be null.

```
int sip_add_content(sip_msg_t sip_msg, char *contents);
```

The `sip_add_content()` function adds the message body to the SIP message. The value of the *contents* parameter must be a null-terminated string.

```
int sip_add_accept(sip_msg_t sip_msg, char *type, char *subtype, char *mparams, char
*params);
```

The `sip_add_accept()` function adds an `Accept` header using the values of the *type* and *subtype* parameters. If the values of the *type* and *subtype* parameters are `NULL`, the function adds an empty `Accept` header. If the value of the *type* parameter is not `NULL` and the value of the *subtype* parameter is `NULL`, the function adds an `Accept` header that uses the value of the *type* parameter for the header's type value. The value of the *subtype* parameter for this header is equal to `*`. You can provide *media* and *accept* parameters by passing them as the *mparams* and *params* arguments.

```
int sip_add_accept_enc(sip_msg_t sip_msg, char *code, char *params);
```

The `sip_add_accept_enc()` function adds an `Accept-Encoding` header with the coding value that is specified in the *code* parameter. You can provide acceptance parameters in the *params* argument.

```
int sip_add_accept_lang(sip_msg_t sip_msg, char *lang, char *params);
```

The `sip_add_accept_lang()` function adds an `Accept-Language` header with the language that is specified in the *lang* parameter. You can provide acceptance parameters in the *params* argument.

```
int sip_add_alert_info(sip_msg_t sip_msg, char *alert, char *params);
```

The `sip_add_alert_info()` function adds an `Alert-Info` header with the alert parameter that is specified in the *alert* parameter. You can provide alert parameters in the *params* argument.

```
int sip_add_allow(sip_msg_t sip_msg, sip_method_t method);
```

The `sip_add_allow()` function adds an `ALLOW` header with the method that is specified in the *method* parameter. These are the valid values for the *method* parameter:

- INVITE
- ACK
- OPTIONS
- BYE

- CANCEL
- REGISTER
- REFER
- INFO
- SUBSCRIBE
- NOTIFY
- PRACK

```
int sip_add_call_info(sip_msg_t sip_msg, char *uri, char *params);
```

The `sip_add_call_info()` function adds a `Call-Info` header by using the URI passed in the value of the *uri* parameter. You can provide URI parameters in the value of the *params* parameter.

```
int sip_add_content_disp(sip_msg_t sip_msg, char *dis_type, char *params);
```

The `sip_add_content_disp()` function adds a `Content-Disposition` header using the display type that is given by the value of the *disp_type* parameter. You can provide disposition parameters in the value of the *params* parameter.

```
int sip_add_content_enc(sip_msg_t sip_msg, char *code);
```

The `sip_add_content_enc()` adds a `Content-Encoding` header using the content encoding that is given by the value of the *code* parameter.

```
int sip_add_content_lang(sip_msg_t sip_msg, char *lang);
```

The `sip_add_content_lang()` adds a `Content-Language` header using the language that is given by the value of the *lang* parameter.

```
int sip_add_date(sip_msg_t sip_msg, char *date);
```

The `sip_add_date()` function adds a `Date` header using the value given in the *date* parameter. This value is a string from the `SIP-Date` field. See section 25.1 of RFC 3261 for more information.

```
int sip_add_error_info(sip_msg_t sip_msg, char *uri, char *params);
```

The `sip_add_error_info()` function adds an `Error-Info` header using the value for `error-uri` that is specified in the *uri* parameter. You can provide URI parameters in the value of the *params* parameter.

```
int sip_add_expires(sip_msg_t sip_msg, int secs);
```

The `sip_add_expires()` function adds an `Expires` header that uses the number of seconds that is specified by the value of the *secs* parameter.

```
int sip_add_in_reply_to(sip_msg_t sip_msg, char *reply_id);
```

The `sip_add_in_reply_to()` function adds an `In-Reply-To` header using the value for `callid` that is passed by the *reply_id* parameter.

```
int sip_add_mime_version(sip_msg_t sip_msg, char *version);
```

The `sip_add_mime_version()` function adds a `Mime-Version` header using the version number that is specified by the value of the *version* parameter.

```
int sip_add_min_expires(sip_msg_t sip_msg, int secs);
```

The `sip_add_min_expires()` function adds a `Min-Expires` header using number of seconds that is specified by the value of the *secs* parameter.

```
int sip_add_org(sip_msg_t sip_msg, char *org);
```

The `sip_add_org()` function adds an `Organization` header. The header has the value that is specified in the *org* parameter.

```
int sip_add_priority(sip_msg_t sip_msg, char *prio);
```

The `sip_add_priority()` function adds a `Priority` header. The header has the value that is specified in the *prio* parameter.

```
int sip_add_reply_to(sip_msg_t sip_msg, char *display_name, char *addr, char *plist,
boolean_t add_aquot);
```

The `sip_add_reply_to()` function appends a `REPLY-TO` header to a SIP message. This function creates the header using the *display_name* and *addr* parameters, in addition to any parameters you specify in *plist*. If the value of the *add_aquot* parameter is `B_TRUE`, this function encloses the value of the *addr* parameter within the delimiting characters < and >. If the *display_name* parameter has a value, the value of the *add_aquot* parameter must be `B_TRUE`.

```
int sip_add_passertedid(sip_msg_t sip_msg, char *display_name, char *uri, boolean_t
add_aquot);
```

The `sip_add_passertedid()` function appends a `P-ASSERTED-IDENTITY` header to a SIP message. This function creates the header using the *display_name* and *uri* parameters. The *uri* cannot have a null value. If the value of the *add_aquot* parameter is `B_TRUE`, this function encloses the value of the *uri* parameter within the delimiting characters < and >. If the *display_name* parameter has a value, the value of the *add_aquot* parameter must be `B_TRUE`.

```
int sip_add_ppreferredid(sip_msg_t sip_msg, char *display_name, char *uri, boolean_t
add_aquot);
```

The `sip_add_ppreferredid()` function appends a `P-PREFERRED-IDENTITY` header to a SIP message. This function creates the header using the *display_name* and *uri* parameters. The *uri* cannot have a null value. If the value of the *add_aquot* parameter is `B_TRUE`, this function encloses the value of the *uri* parameter within the delimiting characters < and >.

```
int sip_add_require(sip_msg_t sip_msg, char *req);
```

The `sip_add_require()` function adds a `Require` header by using the value of `option-tag` that is specified in the *req* parameter.

```
int sip_add_retry_after(sip_msg_t sip_msg, int secs, char *cmt, char *params);
```

The `sip_add_retry_after()` function adds a `Retry-After` header with a value equal to the number of seconds that is specified by the value of the *secs* parameter. You can provide retry parameters in the value of the *params* parameter. You can provide comments in the value of the *cmt* parameter.

`int` **`sip_add_route`**`(sip_msg_t` *sip_msg*`, char` **display_name*`, char` **route_uri*`, char` **route_param*`);`

The `sip_add_route()` function appends a `ROUTE` header to a SIP message. This function creates the header using the *display_name* and *route_uri* parameters, in addition to any parameters that you specify in *route_param*. This function encloses the value of the *route_uri* parameter within the delimiting characters < and > before adding that value to the header.

`int` **`sip_add_record_route`**`(sip_msg_t` *sip_msg*`, char` **display_name*`, char` **route_uri*`, char` **route_param*`);`

The `sip_add_record_route()` function appends a `RECORD-ROUTE` header to a SIP message. This function creates the header using the *display_name* and *route_uri* parameters, in addition to any parameters that you specify in *route_param*. This function encloses the value of the *route_uri* parameter within the delimiting characters < and > before adding that value to the header.

`int` **`sip_add_server`**`(sip_msg_t` *sip_msg*`, char` **svr_val*`);`

The `sip_add_server()` function adds a `Server` header by using the value of `server-val` that is given by the `svr_val` parameter.

`int` **`sip_add_subject`**`(sip_msg_t` *sip_msg*`, char` **subject*`);`

The `sip_add_subject()` function adds a `Subject` header with the value given in the *subject* parameter.

`int` **`sip_add_supported`**`(sip_msg_t` *sip_msg*`, char` **support*`);`

The `sip_add_supported()` function adds a `Supported` header with the value of `option-tag` that is given in the *support* parameter.

`int` **`sip_add_tstamp`**`(sip_msg_t` *sip_msg*`, char` **time*`, char` **delay*`);`

The `sip_add_tstamp()` function adds a `Timestamp` header using the timestamp value that is given in the *time* parameter. If the value of the delay is not `NULL`, this function adds the delay value given in the **delay* parameter.

`int` **`sip_add_unsupported`**`(sip_msg_t` *sip_msg*`, char` **unsupport*`);`

The `sip_add_unsupported()` function adds an `Unsupported` header using the value for `option-tag` that is specified by the *unsupport* parameter.

`int` **`sip_add_user_agent`**`(sip_msg_t` *sip_msg*`, char` **usr*`);`

The `sip_add_user_agent()` function adds a `User-Agent` header using the value for `server-val` that is specified by the *usr* parameter.

```
int sip_add_warning(sip_msg_t sip_msg, int code, char *addr, char *msg);
```

The `sip_add_warning()` function adds a `Warning` header using the warning code that is specified by the value of the *code* parameter. The function uses the warning agent that is specified by the value of the *addr* parameter. The function uses the warning text that is specified by the value of the *msg* parameter.

SIP Request and Response Creation Functions

```
int sip_add_response_line(sip_msg_t sip_response, int response, char *response_code);
```

The `sip_add_response_line()` function adds a response line that uses the values that are passed in the *response* and *response_code* parameters. The *response_code* parameter is the reason phrase for the response. The value of the *response_code* parameter cannot be `NULL`. The SIP version in the response line is SIP/2.0. The `sip_add_request_line()` function adds a CRLF to the response line before adding the line to the SIP message. For the response listed in Section 21 of RFC 3261, obtain the value of the *response_code* parameter by using the `sip_get_resp_desc()` function.

```
int sip_add_request_line(sip_msg_t sip_request, sip_method_t method, char
*request_uri);
```

The `sip_add_request_line()` function adds a request line to a SIP message using the method and the request URI. The request URI cannot be `NULL`. The `method` can take on the following values:

- `INVITE`
- `ACK`
- `OPTIONS`
- `BYE`
- `CANCEL`
- `REGISTER`
- `REFER`
- `INFO`
- `SUBSCRIBE`
- `NOTIFY`
- `PRACK`

The added request line has a SIP version of SIP/2.0. The API adds a CRLF to the request line before adding the line to the SIP message.

```
sip_msg_t sip_create_response(sip_msg_t sip_request, int response, char *response_code,
char *totag, char *contacturi);
```

The `sip_create_response()` function creates a response for the SIP request that is provided by the *sip_request* parameter. The *response* and **response_code* parameters serve the same purpose as in the `sip_add_response_line()` function. You can specify a tag value that the `sip_create_response()` function adds to the `TO` header. If you do not specify the tag value, this function adds a tag value unless the response is `100 (Trying)`. In this case, the application must specifically add the tag. The

SIP response message is created by adding a `Response` line by using the *response* and **response_code* parameters with a SIP version of SIP/2.0. This function then copies the following headers from the request to the response message:

- All VIA headers
- `FROM` header
- `TO` header
- `Call-ID` header
- `Cseq` header
- All `Record-Route` headers

The `sip_create_response()` function returns the resulting SIP response message.

`sip_msg_t` **`sip_create_dialog_req`**`(sip_method_t` *method*`, sip_dialog_t` *dialog*`, char` **transport*`, char` **sent_by*`, int` *sent_by_port*`, char` **via_param*`, uint32_t` *maxforward*`, int` *cseq*`);`

The `sip_create_dialog_req()` function creates an in-dialog request using the state that is maintained in the *dialog* parameter. This function uses the method that is specified in the *method* parameter to create the request line. The request URI is either the contact URI from the specified dialog or the first URI from the `Route` set, if that set is present. For more details on the `Route` set, see section 12.2 of RFC 3261. The `FROM`, `TO`, `CONTACT` and `Call-ID` headers are added to the request using the state that is maintained in the dialog. The `Max-Forwards` header is added using the value of the *maxforward* parameter. If the value of the *cseq* parameter is not negative, this function adds the `Cseq` header using the provided value. In all other cases, this function increments the dialog's local *cseq* and uses the result as the sequence number.

This function adds a VIA header to the request. This function constructs the VIA header by using the values in the *transport*, *sent_by*, and *sent_by_port* parameters, as well as any values you specify in the *via_params* parameter. The VIA header's protocol is `SIP` and the version number is `2.0`.

`int` **`sip_create_OKack`**`(sip_msg_t` *response*`, sip_msg_t` *ack_msg*`, char` **transport*`, char` **sent_by*`, int` *sent_by_port*`, char` **via_params*`);`

The `sip_create_OKack()` function constructs an `ACK` signal for a final 2xx response, in compliance with section 13.2.2.4 of RC 3261. The value of the *response* parameter is the 2xx response.

This function adds a VIA header to the request. This function constructs the VIA header by using the values in the *transport*, *sent_by*, and *sent_by_port* parameters, as well as any values you specify in the *via_params* parameter. The VIA header's protocol is `SIP` and the version number is `2.0`.

Header and Message Copying Functions

`int` **`sip_copy_start_line`**`(sip_msg_t` *`from`*`, sip_msg_t` *`to`*`);`

The `sip_copy_start_line()` function copies the first line of the SIP message. The first line is either the request or the response line. The line is copied from the position that is indicated by the value of `msg_from` to the position that is indicated by the value of `msg_to`.

`int` **`sip_copy_header`**`(sip_msg_t` *`sip_msg`*`, sip_header_t` *`sip_header`*`, char` *`*param`*`);`

The `sip_copy_header()` function creates a new SIP header using the values given by the *sip_header* and *param* parameters, if the values for these parameters are not NULL. This function adds the new header to the message that is indicated by the value of the *sip_msg* parameter.

`int` **`sip_copy_header_by_name`**`(sip_msg_t` *`from`*`, sip_msg_t` *`to`*`, char` *`*header_name`*`, char` *`*param`*`);`

The `sip_copy_header_by_name()` function copies the header given by the value of the `header_name` parameter from the position specified in `msg_from` to the position specified in `msg_to`. The value of the *header_name* parameter can be either the long name or the compact name. The function creates a new SIP header using the contents of the `header_name` field in `msg_from`, then adds the value of the *param* parameter if that value is not `NULL`. The function adds the resulting header to `msg_to`.

`int` **`sip_copy_all_headers`**`(sip_msg_t` *`from`*`, sip_msg_t` *`to`*`);`

The `sip_copy_all_headers()` function copies all of the headers, except for the start line, from the SIP message indicated in the *from* parameter to the SIP message indicated in the *to* parameter. The API will not copy headers from a message that is marked as deleted.

`sip_msg_t` **`sip_clone_msg`**`(const sip_msg_t` *`sip_msg`*`);`

The `sip_clone_msg()` function returns a message that is a copy of the SIP message that is indicated by the value of the *sip_msg* parameter. The function returns a message that contains all of the headers and the contents that are present in the SIP message that is indicated by the value of the *sip_msg* parameter.

Header and Value Deleting Functions

`void` **`sip_delete_start_line`**`(sip_msg_t` *`sip_msg`*`);`

The `sip_delete_start_line()` function deletes the start line from the SIP message that is indicated by the value of the *sip_msg* parameter. The start line can be either the request line or the response line.

`int` **`sip_delete_header`**`(sip_header_t` *`sip_header`*`);`

The `sip_delete_header()` function deletes the header that is indicated by the value of the *sip_header* parameter from its associated SIP message. The function marks that header as deleted. The stack destroys that header when the SIP message is destroyed.

```
int sip_delete_header_by_name(sip_msg_t sip_msg, char *header_name);
```

The `sip_delete_header_by_name()` function deletes the header that is specified by the value of the *header_name* parameter from the SIP message that is indicated by the value of the *sip_msg* parameter. The value of the *header_name* parameter can be either the long name or the compact name for the SIP header. This function marks the header as deleted. The stack destroys this header when it destroys the SIP message.

```
int sip_delete_value(sip_header_t header, sip_header_value_t value);
```

The `sip_delete_value()` function deletes the value that is specified by the *sip_value* parameter from the header that is specified by the *sip_header* parameter. The stack marks the value as deleted and destroys the value when it destroys the header.

Header Lookup Functions

```
const struct sip_header *sip_get_header(sip_msg_t sip_msg, char *header_name,
sip_header_t old_header, int *error);
```

The `sip_get_header()` function obtains the SIP message header that is specified by the value of the *header_name* parameter. If the value of the *header_name* parameter is `NULL`, the function returns the first header of the SIP message. If the value of the *old_header* parameter is not `NULL`, the function searches for a header that matches the value of the *header_name* parameter. This search begins from the header specified by the value of the *old_header* parameter. For example, to search for the first VIA header, use `VIA` as the value for the *header_name* parameter and NULL as the value for the *old_header* parameter. To get the next VIA header from the message, use the value of the *sip_header* parameter that the previous call to the `sip_get_header()` function returned as the value of the *old_header* parameter. This function ignores headers that are marked as deleted. A lookup for a deleted header returns a result of NULL unless the message has another header of the same name that is not marked as deleted. The value of the *header_name* parameter can be either the long or the compact name.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value. If the value of the `sip_msg` parameter is null, the function sets the value of the error parameter to `EINVAL`. If the function cannot find the header, it sets the value of the error parameter to `ENOENT`.

Value Retrieval and Response Description Functions

```
const struct sip_value *sip_get_header_value(const struct sip_header *sip_header, int
*error);
```

The `*sip_get_header_value()` function returns the parsed value for the header that is specified by the value of the *sip_header* parameter. This function does not return any values that are marked as deleted.

```
const struct sip_value *sip_get_next_value(sip_header_value_t old_value, int *error);
```

the `*sip_get_next_value()` function returns the value, if any, that follows the value that is specified by the value of the *old_value* parameter. This function does not return any values that are marked as deleted.

```
const sip_str_t *sip_get_param_value(sip_header_value_t value, char *param_name,
int *error);
```

The `*sip_get_param_value()` function returns the parameter that is specified by the value of the *param_name* parameter from the SIP message header that is specified by the *header_value* parameter.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
const sip_param_t *sip_get_params(sip_header_value_t value, int *error);
```

The `*sip_get_params()` function returns the parameter list that is associated with the value that is specified in the *header_value* parameter.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
char *sip_get_resp_desc(int resp_code);
```

The `*sip_get_resp_desc()` function returns the reason phrase for the response code that is specified by the value of the *resp_code* parameter. This function supports the response codes that are listed in section 21 of RFC 3261. This function returns `UNKNOWN` for unknown response codes.

SIP ID Generating Functions

```
char *sip_branchid(sip_msg_t sip_msg);
```

The `*sip_branchid()` function returns a randomly generated string for the branch parameter in a VIA header. This function prefixes the string with the string `z9hG4bK` to meet the requirements of RFC 3261. If the value of the *sip_msg* parameter is a valid SIP message, this function constructs the

branch ID by using an MD5 hash of the VIA, TO, FROM, CALL-ID, CSeq, and Request URI headers. If the value of the *sip_msg* parameter is NULL, this function returns a random string that is prefixed with the string z9hG4bK. The calling thread must free the string that this function returns.

```
char *sip_guid();
```

The *sip_guid() function returns a randomly generated string. The calling thread must free the string that this function returns.

```
uint32_t sip_get_cseq();
```

The sip_get_cseq() function returns the most significant 31 bits returned by the time(2) system call. You can use this value as the initial sequence number.

VIA Functions

```
int sip_get_num_via(sip_msg_t sip_msg, int *error);
```

The sip_get_num_via() function returns the number of VIA headers that are in the SIP message.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
char *sip_get_branchid(sip_msg_t sip_msg, int *error);
```

The *sip_get_branchid() function returns the value of the branch parameter in the topmost VIA header in the SIP message. The calling thread must free the string that this function returns.

SIP Message Sending Function

```
int sip_sendmsg(sip_conn_object_t obj, sip_msg_t sip_msg, sip_dialog_t sip_dialog,
uint32_t flags);
```

The sip_sendmsg() function sends a SIP message to the SIP stack and, subsequently, to the peer. The function passes the connection object that is specified by the value of the *obj* parameter along with the message that is specified by the value of the *sip_msg* parameter. This function provides the dialog that is associated with the specified SIP message, if that dialog exists and the SIP stack is maintaining dialogs. This function uses flags to indicate any special processing that the stack needs to do for the specified SIP message. The values of flags are the result of a boolean OR operation performed on the following attributes:

SIP_SEND_STATEFUL — Send the request or response statefully. The stack creates and maintains a transaction for the request or response in compliance with section 17 of RFC 3261.

`SIP_DIALOG_ON_FORK` When this flag is set, the stack creates multiple dialogs when the application forks the request. This flag is only meaningful when the stack maintains dialogs. When this flag is not set, only the first dialog creating request creates a dialog.

Processing Inbound Messages

`void` **`sip_process_new_packet`**`(sip_conn_object_t` *conn_object*`, void *`*msgstr*`, size_t` *msglen*`);`

The `sip_process_new_packet()` function processes the incoming message string from the application using the message string's length and associated connection object. This function transforms the incoming message string into a SIP message. If the transport is byte-stream oriented, this function manages breaking the byte stream at the SIP message boundaries. The stack delivers the SIP message to the application after completing any other required stack processing.

Transaction Layer Functions

`const struct sip_xaction *`**`sip_get_trans`**`(sip_msg_t` *sip_msg*`, int` *which*`, int *`*error*`);`

The `*sip_get_trans()` function returns the transaction that is associated with the SIP message in the *sip_msg* parameter. This function follows the specifications of sections 17.1.3 and 17.2.3 of RFC 3261 to determine a matching transaction. If this function finds a matching transaction, this call increments the reference count on the transaction. The calling thread must release the hold on the transaction by using the `sip_release_trans()` function. The *which* parameter specifies whether the transaction of interest is a client or a server transaction. When a client receives a response, the value of the which parameter is `SIP_CLIENT_TRANSACTION`. When a server gets a request, the value of the which parameter is `SIP_SERVER_TRANSACTION`.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

`sip_method_t` **`sip_get_trans_method`**`(sip_transaction_t` *sip_trans*`, int *`*error*`);`

The `sip_get_trans_method()` function returns the method that created the transaction that is specified by the value of the *sip_trans* parameter.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

`int` **`sip_get_trans_state`**`(sip_transaction_t` *sip_trans*`, int *`*error*`);`

The `sip_get_trans_state()` function returns the state of the transaction that is specified by the value of the *sip_trans* parameter. A CLIENT transaction can have the following states:

- `SIPS_CLNT_CALLING`
- `SIPS_CLNT_INV_PROCEEDING`
- `SIPS_CLNT_INV_COMPLETED`
- `SIPS_CLNT_INV_TERMINATED`
- `SIPS_CLNT_TRYING`
- `SIPS_CLNT_NONINV_PROCEEDING`
- `SIPS_CLNT_NONINV_COMPLETED`
- `SIPS_CLNT_NONINV_TERMINATED`

A SERVER transaction can have the following states:

- `SIPS_SRV_INV_PROCEEDING`
- `SIPS_SRV_INV_COMPLETED`
- `SIPS_SRV_CONFIRMED`
- `SIPS_SRV_INV_TERMINATED`
- `SIPS_SRV_TRYING`
- `SIPS_SRV_NONINV_PROCEEDING`
- `SIPS_SRV_NONINV_COMPLETED`
- `SIPS_SRV_NONINV_TERMINATED`

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

`const struct sip_message` ***sip_get_trans_orig_msg***`(sip_transaction_t` *sip_trans*`, int` **error*`);`

The `*sip_get_trans_orig_msg()` function returns the original message that created this transaction. A client transaction returns the request. A server transaction returns the response sent to the transaction creating request.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

`const struct sip_conn_object` ***sip_get_trans_conn_obj***`(sip_transaction_t` *sip_trans*`, int` **error*`);`

The `*sip_get_trans_conn_obj()` function returns the cached connection object for the transaction that is specified by the value of the *sip_trans* parameter. This connection object is the object that the application passed in, along with the most recent message for this transaction.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
void sip_hold_trans(sip_transaction_t sip_trans, int *error);
```

The `sip_hold_trans()` function increments the reference count on the transaction that is specified by the value of the *sip_trans* parameter. The caller of the `sip_get_trans()` function must call the `sip_release_trans()` function for the returned transaction after using the transaction.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
void sip_release_trans(sip_transaction_t sip_trans, int *error);
```

The `sip_release_trans()` function decrements the reference count on the transaction that is specified by the value of the *sip_trans* parameter. The caller of the `sip_get_trans()` function must call the `sip_release_trans()` function for the returned transaction after using the transaction. If the reference count drops to zero and the transaction is in a terminated state, the stack destroys the transaction.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

Dialog Layer Functions

```
int sip_get_dialog_state(sip_dialog_t dialog, int *error);
```

The `sip_get_dialog_state()` function returns the state of a dialog. A dialog be in the following states:

- `NEW-DIALOG`
- `EARLY-DIALOG`
- `CONFIRMED-DIALOG`
- `DESTROYED-DIALOG`

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
const sip_str_t sip_get_dialog_callid(sip_dialog_t dialog, int *error);
```

The `sip_get_dialog_callid()` function returns the pointer to the *callid* value for the dialog that is specified by the value of the *dialog* parameter. This function returns a pointer to the value and the length of the value.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
const sip_str_t sip_get_dialog_local_tag(sip_dialog_t dialog, int *error);
```

The `sip_get_dialog_local_tag()` function returns a pointer to the local tag value for the dialog that is specified by the value of the *dialog* parameter. This function returns a pointer to the value and the length of the value.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
const sip_str_t sip_get_dialog_remote_tag(sip_dialog_t dialog, int *error);
```

The `sip_get_dialog_remote_tag()` function returns a pointer to the remote tag value for the dialog that is specified by the value of the *dialog* parameter. This function returns a pointer to the value and the length of the value.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
const struct uri_t sip_get_dialog_local_uri(sip_dialog_t dialog, int *error);
```

The `sip_get_dialog_local_uri()` function returns the local URI for the dialog that is specified by the value of the *dialog* parameter. This function returns a parsed URI.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
const struct uri_t sip_get_dialog_remote_uri(sip_dialog_t dialog, int *error);
```

The `sip_get_dialog_remote_uri()` function returns the remote URI for the dialog that is specified by the value of the *dialog* parameter. This function returns a parsed URI.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
const struct uri_t sip_get_dialog_remote_target_uri(sip_dialog_t dialog, int *error);
```

The `sip_get_dialog_remote_target_uri()` function returns the remote target URI for the dialog that is specified by the value of the *dialog* parameter. This function returns a URI in the `sip_uri_t` variable, which is a parsed URI.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
const sip_str_t sip_get_dialog_route_set(sip_dialog_t dialog, int *error);
```

The `sip_get_dialog_route_set()` function returns the route set for the dialog that is specified by the value of the *dialog* parameter. The return value is a pointer to the list of comma separated routes and the length of the list.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
uint32_t sip_get_dialog_local_cseq(sip_dialog_t dialog, int *error);
```

The `sip_get_dialog_local_cseq()` function returns the sequence number for the next request in the dialog.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
uint32_t sip_get_dialog_remote_cseq(sip_dialog_t dialog, int *error);
```

The `sip_get_dialog_remote_cseq()` function returns the remote CSeq number for the dialog.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
int sip_get_dialog_type(sip_dialog_t dialog, int *error);
```

The `sip_get_dialog_type()` function returns the type of the dialog. A dialog can have the following types:

`SIP_UAC_DIALOG` UAC created dialog

`SIP_UAS_DIALOG` UAS created dialog

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
void sip_hold_dialog(sip_dialog_t dialog, int *error);
```

The `sip_hold_dialog()` function increments the reference number for the dialog that is specified by the value of the *dialog* parameter. The stack will not destroy a dialog with a reference count greater than zero. An application that holds a dialog must release the dialog after using it.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
void sip_release_dialog(sip_dialog_t dialog, int *error);
```

The `sip_release_dialog()` function decrements the reference number for the dialog that is specified by the value of the *dialog* parameter. The stack will not destroy a dialog with a reference count greater than zero. An application that holds a dialog must release the dialog after using it.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

`void` **`sip_delete_dialog`**`(sip_dialog_t` *dialog*`, int *`*error*`);`

The `sip_delete_dialog()` function terminates the dialog that is specified by the value of the *dialog* parameter. This function first decrements the reference count for the dialog, then destroys it if the count is zero. The stack will not destroy a dialog with a reference count greater than zero. An application that holds a dialog must release the dialog after using it.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

URI Functions

`const struct uri_t` **`sip_get_uri_parsed`**`(sip_header_value_t` *value*`, int *`*error*`);`

The `sip_get_uri_parsed()` function returns the parsed URI from the value that is specified by the *value* parameter. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

`const struct uri_t` **`sip_get_request_uri`**`(sip_msg_t` *sip_msg*`, int *`*error*`);`

The `sip_get_request_uri()` function returns the parsed URI from the start line of a SIP request. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

SIP Header Value Retrieval Functions

`boolean_t` **`sip_msg_is_request`**`(const sip_msg_t` *sip_msg*`, int *`*error*`);`

The `sip_msg_is_request()` function returns a value of `B_TRUE` if the message is a request. This function returns a value of `B_FALSE` in all other cases.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
boolean_t sip_msg_is_response(const sip_msg_t sip_msg, int *error);
```

The `sip_msg_is_response()` function returns a value of `B_TRUE` if the message is a response. This function returns a value of `B_FALSE` in all other cases.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
sip_method_t sip_get_request_method(const sip_msg_t sip_msg, int *error);
```

The `sip_get_request_method()` function returns the SIP method from the start line of the SIP request message that is contained in the *sip_msg* parameter.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
int sip_get_response_code(sip_msg_t sip_msg, int *error);
```

The `sip_get_response_code()` function returns the response code from the start line of the SIP response message that is contained in the *sip_msg* parameter.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
const sip_str_t *sip_get_response_phrase(sip_msg_t sip_msg, int *error);
```

The `sip_get_response_phrase()` function returns the reason phrase from the start line of the SIP response message that is contained in the *sip_msg* parameter.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
const sip_str_t *sip_get_sip_version(sip_msg_t sip_msg, int *error);
```

The `sip_get_sip_version()` function returns the SIP version from the start line of the SIP response message that is contained in the *sip_msg* parameter.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
int sip_get_msg_len(sip_msg_t sip_msg, int *error);
```

The `sip_get_msg_len()` function returns the length of the SIP message that is contained in the *sip_msg* parameter.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
const sip_str_t sip_get_contact_display_name(sip_header_value_t value, int *error);
```

The `sip_get_contact_display_name()` function returns the display name from a `CONTACT` header. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
const sip_str_t sip_get_from_display_name(sip_msg_t sip_msg, int *error, int *error);
```

The `sip_get_from_display_name()` function returns the display name from a `FROM` header. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_to_display_name(sip_msg_t sip_msg, int *error);
```

The `sip_get_to_display_name()` function returns the display name from a `TO` header. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t *sip_get_from_tag(sip_msg_t sip_msg, int *error);
```

The `sip_get_from_tag()` function returns the tag value from a `FROM` header. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_to_tag(sip_msg_t sip_msg, int *error);
```

The `sip_get_to_tag()` function returns the tag value from a `TO` header. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_callid(sip_msg_t sip_msg, int *error);
```

The `sip_get_callid()` function gets the `callid` value from a `Call-ID` header. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the

value of the *error parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the *error parameter to the error number.

```
int sip_get_callseq_num(sip_msg_t sip_msg, int *error);
```

The sip_get_callseq_num() function returns the call sequence value from the Cseq header. If the value of the *error parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the *error parameter to the error number.

```
sip_method_t sip_get_callseq_method(sip_msg_t sip_msg, int *error);
```

The sip_get_callseq_method() function returns the method from the Cseq header. If the value of the *error parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the *error parameter to the error number.

```
const sip_str_t sip_get_via_sent_by_host(sip_header_value_t value, int *error);
```

The sip_get_via_sent_by_host() function returns the value of the host from a VIA header.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
int sip_get_via_sent_by_port(sip_header_value_t value, int *error);
```

The sip_get_via_sent_by_port() function returns the value of the port from a VIA header.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
const sip_str_t sip_get_via_sent_protocol_version(sip_header_value_t value, int
*error);
```

The sip_get_via_sent_protocol_version() function returns the value of the protocol version from a VIA header.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
const sip_str_t sip_get_via_sent_protocol_name(sip_header_value_t value, int *error);
```

The sip_get_via_sent_protocol_name() function returns the value of the protocol name from a VIA header.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
const sip_str_t sip_get_via_sent_transport(sip_header_value_t value, int *error);
```

The `sip_get_via_sent_transport()` function returns the value of the transport from a VIA header.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
int sip_get_maxforward(sip_msg_t sip_msg, int *error);
```

The `sip_get_maxforward()` function returns the number of hops that is listed as a value in the `Max-forwards` header. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
int sip_get_content_length(sip_msg_t sip_msg, int *error);
```

the `sip_get_content_length()` function returns the length that is listed as a value in the `Content-Length` header of a SIP message. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_content_type(sip_msg_t sip_msg, int *error);
```

The `sip_get_content_type()` function returns the content type from the `Content-Type` header of a SIP message. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_content_sub_type(sip_msg_t sip_msg, int *error);
```

The `sip_get_content_sub_type()` function returns the content subtype from the `Content-Type` header of a SIP message. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
char sip_get_content(sip_msg_t sip_msg, int *error);
```

The `sip_get_content()` function returns the message body of the SIP message. The calling thread must free the string that this function returns. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_accept_type(sip_header_value_t value, int *error);
```

The `sip_get_accept_type()` function returns the type from the `Accept` header value of a SIP message. This function returns a pointer to the header and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_accept_sub_type(sip_header_value_t value, int *error);
```

The `sip_get_accept_sub_type()` function returns the subtype from the `Accept` header value of a SIP message. This function returns a pointer to the header and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_accept_enc(sip_header_value_t value, int *error);
```

The `sip_get_accept_enc()` function returns the encoding from the `Accept-Encoding` header value of a SIP message. This function returns a pointer to the header and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_accept_lang(sip_header_value_t value, int *error);
```

The `sip_get_accept_lang()` function returns the language from the `Accept-Language` header value of a SIP message. This function returns a pointer to the header and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_alert_info_uri(sip_header_value_t value, int *error);
```

The `sip_get_alert_info_uri()` function returns the URI string *alert-param* from the `Alert-Info` header value of a SIP message. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
sip_method_t sip_get_allow_method(sip_header_value_t value, int *error);
```

The `sip_get_allow_method()` function returns the method from the `Allow` header value of a SIP message. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
int sip_get_min_expires(sip_msg_t sip_msg, int *error);
```

The `sip_get_min_expires()` function returns the number of seconds that is specified in the `Min-Expires` header value of the SIP message that is specified in the *sip_msg* parameter. If the value

of the *error parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the *error parameter to the error number.

```
const sip_str_t sip_get_mime_version(sip_msg_t sip_msg, int *error);
```

The sip_get_mime_version() function returns the MIME version value of a SIP message. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the *error parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the *error parameter to the error number.

```
const sip_str_t sip_get_org(sip_msg_t sip_msg, int *error);
```

The sip_get_org() function returns the value of the Organization header value of a SIP message. This function returns a pointer to the header and the length of the value.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
const sip_str_t *sip_get_priority(sip_msg_t sip_msg, int *error);
```

The sip_get_priority() function returns the value of the Priority header value of a SIP message. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the *error parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the *error parameter to the error number.

```
const sip_str_t sip_get_replyto_display_name(sip_msg_t sip_msg, int *error);
```

The sip_get_replyto_display_name() function returns the display name from the Reply-To header value of a SIP message. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the *error parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the *error parameter to the error number.

```
const sip_str_t sip_get_replyto_uri_str(sip_msg_t sip_msg, int *error);
```

The sip_get_replyto_uri_str() function returns the URI from the Reply-To header value of a SIP message. If the value of the *error parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the *error parameter to the error number.

```
const sip_str_t sip_get_date_time(sip_msg_t sip_msg, int *error);
```

The sip_get_date_time() function returns the value of the time from the Date header of a SIP message. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the *error parameter is not NULL, the function sets the value

to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
int sip_get_date_day(sip_msg_t sip_msg, int *error);
```

The `sip_get_date_day()` function returns the value of the day from the `Date` header of a SIP message. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_date_month(sip_msg_t sip_msg, int *error);
```

The `sip_get_date_month()` function returns the value of the month from the `Date` header of a SIP message. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_date_wkday(sip_msg_t sip_msg, int *error);
```

The `sip_get_date_wkday()` function returns the value of the weekday from the `Date` header of a SIP message. This function returns a pointer to the header and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
int sip_get_date_year(sip_msg_t sip_msg, int *error);
```

The `sip_get_date_year()` function returns the value of the year from the `Date` header of a SIP message. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_date_timezone(sip_msg_t sip_msg, int *error);
```

The `sip_get_date_timezone()` function returns the value of the time zone from the `Date` header of a SIP message. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_content_disp(sip_msg_t sip_msg, int *error);
```

The `sip_get_content_disp()` function returns the value of the `Content-Disposition` header in a SIP message. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_date_content_enc(sip_header_value_t value, int *error);
```

The `sip_get_date_content_enc()` function returns the encoding from the `Content-Encoding` header value. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_error_info_uri(sip_header_value_t value, int *error);
```

The `sip_get_error_info_uri()` function returns the URI string from the `Error-Info` header value. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
int sip_get_expires(sip_msg_t sip_msg, int *error);
```

The `sip_get_expires()` function returns the `Expires` header value. This header value is an integer. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_require(sip_header_value_t value, int *error);
```

The `sip_get_require()` function returns the value of the `Require` header. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_subject(sip_msg_t sip_msg, int *error);
```

The `sip_get_subject()` function returns the value of the `Subject` header in a SIP message. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_supported(sip_header_value_t value, int *error);
```

The `sip_get_supported()` function returns the value of the `supported` extension from the `Supported` header. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_tstamp_delay(sip_msg_t, int *, int *error);
```

The `sip_get_tstamp_delay()` function returns the value of the `Timestamp` header in a SIP message. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value.

The value of the error parameter is set to zero when this function completes successfully. When this function completes unsuccessfully, it sets the value of the error parameter to the appropriate error value.

```
const sip_str_t sip_get_unsupported(sip_header_value_t value, int *error);
```

The `sip_get_unsupported()` function returns the list of the unsupported features from the `Unsupported` header value in a SIP message. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_server(sip_msg_t sip_msg, int *error);
```

The `sip_get_server()` function returns the value of the `Server` header in a SIP message. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_user_agent(sip_msg_t sip_msg, int *error);
```

The `sip_get_user_agent()` function returns the value of the `User-Agent` header in a SIP message. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
int sip_get_warning_code(sip_header_value_t value, int *error);
```

The `sip_get_warning_code()` function returns the value of the warning code from the `Warning` header value in a SIP message. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_warning_agent(sip_header_value_t value, int *error);
```

The `sip_get_warning_agent()` function returns the value of the warning agent from the `Warning` header value in a SIP message. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_warning_text(sip_header_value_t value, int *error);
```

The `sip_get_warning_text()` function returns the value of the warning text from the `Warning` header value in a SIP message. This function returns a pointer to the header and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_call_info_uri(sip_header_value_t value, int *error);
```

The `sip_call_info_uri()` function returns the URI string from the `Call-Info` header value in a SIP message. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_in_reply_to(sip_header_value_t value, int *error);
```

The `sip_get_in_reply_to()` function returns the value of `callid` from the `In-Reply-To` header value in a SIP message. This function returns a pointer to the display name (or the tag or callid, as appropriate) and the length of the value. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
int sip_get_retry_after_time(sip_msg_t sip_msg, int *error);
```

The `sip_get_retry_after_time()` function returns the value of the time interval from the `Retry-After` header in a SIP message. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

```
const sip_str_t sip_get_retry_after_cmts(sip_msg_t sip_msg, int *error);
```

The `sip_get_retry_after_cmts()` function returns the value of the comments from the `Retry-After` header in a SIP message. If the value of the `*error` parameter is not NULL, the function sets the value to zero if the function completes successfully. If the function does not complete successfully, the function sets the value of the `*error` parameter to the error number.

Connection Object Functions

```
int sip_conn_send(const sip_conn_object_t conn_obj, char *msg, int msglen);
```

The `sip_conn_send()` function is an interface that the application registers. This function sends out the message in the **msg* parameter with a length defined by the *msglen* parameter over the connection that is represented by the *conn_obj* parameter. The stack invokes this function to send the message to the peer after processing.

```
void sip_hold_conn_object(sip_conn_object_t conn_obj);
```

The `sip_hold_conn_object()` function is an interface that the application registers. The stack invokes this interface to increment the reference count on the connection object referenced by the

conn_obj parameter. The stack invokes this function when it caches the connection object. The application cannot free the connection object if the object has any existing references.

```
void sip_rel_conn_object(sip_conn_object_t conn_obj);
```

The `sip_rel_conn_object()` function is an interface that the application registers. The stack invokes this interface to release a reference to a connection object referenced by the *conn_obj* parameter. The stack invokes this function after removing any cached connection object. The application can free connection objects that have no references.

```
boolean_t sip_conn_is_reliable(sip_conn_object_t conn_obj);
```

The `sip_conn_is_reliable()` function is an interface that the application registers. The stack invokes this interface to query whether or not the transport protocol for the connection object referenced by the *conn_obj* parameter is reliable. Reliable transport protocols include TCP and SCTP. UDP is an unreliable protocol.

```
boolean_t sip_conn_is_stream(sip_conn_object_t conn_obj);
```

The `sip_conn_is_stream()` function is an interface that the application registers. The stack invokes this interface to query whether or not is byte-stream oriented. TCP is a byte-stream protocol. SCTP and UDP are message-oriented protocols.

```
int sip_conn_remote_address(sip_conn_object_t conn_obj, struct sockaddr *name,
socklen_t *namelen);
```

The `sip_conn_remote_address()` function provides the remote address in the `sockaddr` structure provided in the *name* parameter. On invocation, the *namelen* parameter contains the length of the address that the calling thread provides to this function. When this function returns from the call, the *len* parameter contains the actual length of the address.

```
int sip_conn_local_addresstimerd(sip_conn_object_t conn_obj, struct sockaddr *name,
socklen_t *namelen);
```

The `sip_conn_local_address()` function provides the local address in the `sockaddr` structure provided in the *name* parameter. On invocation, the *namelen* parameter contains the length of the address that the calling thread provides to this function. When this function returns from the call, the *len* parameter contains the actual length of the address.

```
int sip_conn_transport(const sip_conn_object_t conn_obj);
```

The `sip_conn_transport()` function returns the transport protocol that is associated with the connection object that is specified in the *conn_obj* parameter.

```
int sip_conn_timer1(const sip_conn_object_t conn_obj);
```

An application can register the `sip_conn_timer1()` function to provide a timer value for the T1 timer that is specific to the connection object that is defined in the *conn_obj* parameter. If the application does not register this function, the stack uses the default value of 500 msec. Timer T1 is the round trip time (RTT) estimate.

```
int sip_conn_timer2(const sip_conn_object_t conn_obj);
```

An application can register the `sip_conn_timer2()` function to provide a timer value for the T2 timer that is specific to the connection object that is defined in the *conn_obj* parameter. If the application does not register this function, the stack uses the default value of 4 seconds. Timer T2 is the maximum retransmit interval for non-`INVITE` requests and `INVITE` responses.

```
int sip_conn_timer4(const sip_conn_object_t conn_obj);
```

An application can register the `sip_conn_timer4()` function to provide a timer value for the T4 timer that is specific to the connection object that is defined in the *conn_obj* parameter. If the application does not register this function, the stack uses the default value of 5 seconds. Timer T4 is the maximum duration that a message will remain in the network.

```
int sip_conn_timerd(const sip_conn_object_t conn_obj);
```

An application can register the `sip_conn_timerd()` function to provide a timer value for the TD timer that is specific to the connection object that is defined in the *conn_obj* parameter. If the application does not register this function, the stack uses the default value of 32 seconds. Timer TD is the wait time for response retransmits.

Miscellaneous Functions

```
char sip_msg_to_str(sip_msg_t sip_msg, int *error);
```

The `sip_msg_to_str()` function returns a string that contains all of the headers and content in the SIP message that is defined by the *sip_msg* parameter. The calling thread must free the string that this function returns.

```
char sip_hdr_to_str(sip_header_t sip_header, int *error);
```

The `sip_hdr_to_str()` function returns the value of the *sip_header* parameter as a string. The calling thread must free the string that this function returns.

```
char sip_reqline_to_str(sip_msg_t sip_msg, int *error);
```

The `sip_reqline_to_str()` function returns the request line in the SIP message that is defined by the *sip_msg* parameter as a string. The calling thread must free the string that this function returns.

```
char *sip_respline_to_str(sip_msg_t sip_msg, int *error);
```

The `sip_respline_to_str()` function returns the response line in the SIP message that is defined by the *sip_msg* parameter as a string. The calling thread must free the string that this function returns.

```
int sip_init_conn_object(sip_conn_object_t conn_obj);
```

The `sip_init_conn_object()` function initializes a connection object with library specific private data.

```
void sip_clear_stale_data(sip_conn_object_t conn_obj);
```

The `sip_clear_stale_data()` function clears any library specific private data that is stored in the connection object that is specified by the *conn_obj* parameter.

```
void sip_conn_destroyed(sip_conn_object_t conn_obj);
```

The `sip_conn_destroyed()` function clears any library specific private data that is stored in the connection object that is defined by the value of the *obj* parameter. This function also performs cleanup if this connection object has been cached by the system.

◆ ◆ ◆ CHAPTER 4

Programming with the SIP API

This chapter has specific information about working with the SIP API and examples of use.

Initializing the SIP Stack

The SIP stack must be initialized before performing any other functions. This section discusses the initialization parameters in some detail. The initialization structure is given by the following structure:

```
typedef struct sip_stack_init_s {
            int                     sip_version;
            uint32_t                sip_stack_flags;
            sip_io_pointers_t       *sip_io_pointers;
            sip_ulp_pointers_t      *sip_ulp_pointers;
            sip_header_function_t   *sip_function_table;
    }sip_stack_init_t;
```

`sip_version` This variable must be set to `SIP_STACK_VERSION`

`sip_stack_flags` If the application wants the SIP stack to maintain dialogs, this flag must be set to `SIP_STACK_DIALOGS`, otherwise this must be set to 0. If `SIP_STACK_DIALOGS` is not set, the stack does not deal with dialogs at all.

Upper Layer Registrations

The upper layer registrations are callbacks that the stack invokes to deliver processed inbound SIP messages. The stack also invokes the upper layer registrations when certain events occur. The structure is given by the following code:

```
void (*sip_ulp_recv)(const sip_conn_object_t, sip_msg_t,
                        const sip_dialog_t);
```

This code is a routine that the application must register for the stack to deliver inbound SIP message after processing. The SIP stack invokes this function with the connection object on which the message arrived the SIP message, and associated dialog, if any. The SIP message will be freed once this function returns. If the application wishes to use the message beyond that, it has to hold a reference to the message by using the `sip_hold_msg()` and release it by using the `sip_free_msg()` function after use. Similarly, if the application wishes to cache the dialog, it must increment the reference to the dialog by using the `sip_hold_dialog()` and release it by using the `sip_release_dialog()` function after use.

```
uint_t (*sip_ulp_timeout)(void *, void (*func)(void *),
                              struct timeval *);
boolean_t (*sip_ulp_untimeout)(uint_t);
```

An application can register these two routines if it wants to implement its own timer routines for the stack timers. Typically, an application may not provide these and let the stack use its own built-in timer routines. The timer routines implemented by the stack are only for use by the stack. These routines are not available to applications. If the application registers one, then it must also register the other.

```
int (*sip_ulp_trans_error)(sip_transaction_t, int, void *);
```

The application can register this routine if it wants to be notified of a transaction error. Typically this error occurs when the transaction layer tries to send a message (could be a retransmission or responding to a retransmitted request or sending an `ACK` request for a non-2xx final response) using a cached connection object which fails. If this routine is not registered, on such a failure, the transaction is terminated. The final argument is for future use, it is always set to NULL.

```
void (*sip_ulp_dlg_del)(sip_dialog_t, sip_msg_t, void *);
```

An application can register this routine if it wants to be notified when a dialog is deleted. The dialog that is going to be deleted is passed along with the SIP message, that caused the dialog to be deleted. The final argument is for future use and is always set to NULL.

```
void (*sip_ulp_trans_state_cb)(sip_transaction_t, sip_msg_t, int, int);
void (*sip_ulp_dlg_state_cb)(sip_dialog_t, sip_msg_t, int, int);
```

These two callback routines, if registered, are invoked by the stack when a transaction (`sip_ulp_trans_state_cb`) or a dialog (`sip_ulp_dlg_state_cb`) changes states. The message causing the transition is passed along with the transaction/dialog and also the old (before transition) and current (after transition) states.

Connection Manager Interfaces

The following interfaces must be registered by the application to provide I/O related functionalities to the stack. The interfaces act on a connection object that is defined by the application and transparent to the stack. The application registers these interfaces for the stack to work with the connection object. The connection object is application defined, but the stack requires that the first member of the connection object is a void *, which is used by the stack to store connection object specific information that is private to the stack.

```
int (*sip_conn_send)(const sip_conn_object_t, char *, int);
```

This function will be invoked by the stack after processing an outbound SIP message. This function will actually send the SIP message to the remote endpoint. A return of 0 indicates success. The SIP message is passed to this function a string along with the length information.

```
void (*sip_hold_conn_object)(sip_conn_object_t);
void (*sip_hold_conn_object)(sip_conn_object_t);
```

The application provides mechanism for the stack to indicate that a connection object is in use by the stack and must not be freed. The stack calls the `sip_hold_conn_object()` function to increment the reference count on a connection object and the `sip_hold_conn_object()` function to release it after use. The stack expects that the application won't free the connection object if the application increments the references to that connection object.

```
boolean_t (*sip_conn_is_stream)(sip_conn_object_t);
boolean_t (*sip_conn_is_reliable)(sip_conn_object_t);
```

These interfaces are used by the stack to obtain attributes of the connection object. The `sip_conn_is_stream()` function states if the connection object is byte-stream oriented (TCP) or not (SCTP and UDP) and the `sip_conn_is_reliable()` function indicates if the transport is reliable (TCP and SCTP) or not (UDP).

```
int (*sip_conn_local_address)(sip_conn_object_t, struct sockaddr *,
        socklen_t *);
int (*sip_conn_remote_address)(sip_conn_object_t, struct sockaddr *,
        socklen_t *);
```

These two interfaces are used by the stack to obtain endpoint information for a connection object. The `sip_conn_local_address()` function provides the local address/port information while the `sip_conn_remote_address()` function provides the peer's address/port information. The caller allocates the buffer and passes its associated length along with it. On return the length is updated to reflect the actual length.

```
int (*sip_conn_transport)(sip_conn_object_t);
```

This interface is used to obtain the transport used by a connection object (TCP, SCTP or UDP).

```
int (*sip_conn_timer1)(sip_conn_object_t);
int (*sip_conn_timer2)(sip_conn_object_t);
int (*sip_conn_timer4)(sip_conn_object_t);
int (*sip_conn_timerd)(sip_conn_object_t);
```

These four interfaces may be registered by an application to provide connection object specific timer information. If these are not registered the stack uses default values. The interfaces provide the timer values for Timer 1 (RTT estimate - default 500 msec, Timer 2 (maximum retransmit interval for non-INVITE request and INVITE response - default 4 secs), Timer 4 (maximum duration a message will remain in the network - default 5 secs) and Timer D (wait time for response retransmit interval - default 32 secs).

Custom SIP Headers

An application can optionally provide a table of custom headers and associated parsing functions. The table is an array with an entry for each header. The table contains the following for each header:

```
char *header_name
```

The full name of the header. The application must take care that this does not conflict with existing headers. If the same header name is present in the application registered function table and the one maintained by the stack, the one registered by the application takes precedence.

```
char *header_short_name
```

Compact name, if any, for the header.

```
int (*header_parse_func)(struct sip_header *,
            struct sip_parsed_header **);
```

The parsing function for the header. The parser will set the second argument to the parsed structure. A return value of 0 indicates success.

```
void (*header_free)(struct sip_parsed_header *);
```

The function that will free the parsed header.

```
boolean_t (*header_check_compliance)(struct sip_parsed_header *);
```

An application can optionally provide this function that will check if the header is compliant or not. The compliance for a custom header will be defined by the application.

```
boolean_t (*header_is_equal)(struct sip_parsed_header *,
           struct sip_parsed_header *);
```

An application can optionally provide this function to check if the two input headers are equivalent. The equivalence criteria is defined by the application.

Example of UAS and UAC Use

Both the UAS and UAC in this section use the following connection object definition and associated functions:

EXAMPLE 4–1 Connection Object Definition for UAS and UAC

```
typedef struct my_conn_obj {
        void                    *my_conn_resv;  /* for lib use */
        int                     my_conn_fd;
        struct sockaddr         *my_conn_local;
        struct sockaddr         *my_conn_remote;
        int                     my_conn_transport;
        int                     my_conn_refcnt;
        int                     my_conn_af;
        pthread_mutex_t         my_conn_lock;
        uint32_t                my_conn_flags;
        int                     my_conn_thr;
        int                     my_conn_timer1;
        int                     my_conn_timer2;
        int                     my_conn_timer4;
        int                     my_conn_timerD;
} my_conn_obj_t;


int
my_conn_transport(sip_conn_object_t obj){
    my_conn_obj_t   *conn_obj;
    int             transport;

    conn_obj = (my_conn_obj_t *)obj;
    (void) pthread_mutex_lock(&conn_obj->my_conn_lock);
    transport = conn_obj->my_conn_transport;
    (void) pthread_mutex_unlock(&conn_obj->my_conn_lock);

    return (transport);
}

boolean_t
my_conn_isreliable(sip_conn_object_t obj) {
    my_conn_obj_t   *conn_obj;
    int             transport;
```

EXAMPLE 4–1 Connection Object Definition for UAS and UAC *(Continued)*

```
    conn_obj = (my_conn_obj_t *)obj;
    (void) pthread_mutex_lock(&conn_obj->my_conn_lock);
    transport = conn_obj->my_conn_transport;
    (void) pthread_mutex_unlock(&conn_obj->my_conn_lock);

    return (!(transport == IPPROTO_UDP));
}

boolean_t
my_conn_isstream(sip_conn_object_t obj)
{
    my_conn_obj_t   *conn_obj;
    int             transport;

    conn_obj = (my_conn_obj_t *)obj;
    (void) pthread_mutex_lock(&conn_obj->my_conn_lock);
    transport = conn_obj->my_conn_transport;
    (void) pthread_mutex_unlock(&conn_obj->my_conn_lock);

    return (transport == IPPROTO_TCP);
}

void
my_conn_refhold(sip_conn_object_t obj)
{
    my_conn_obj_t    *conn_obj;

    conn_obj = (my_conn_obj_t *)obj;

    (void) pthread_mutex_lock(&conn_obj->my_conn_lock);
    conn_obj->my_conn_refcnt++;
    (void) pthread_mutex_unlock(&conn_obj->my_conn_lock);
}

void
my_conn_refrele(sip_conn_object_t obj)
{
    my_conn_obj_t    *conn_obj;

    conn_obj = (my_conn_obj_t *)obj;

    (void) pthread_mutex_lock(&conn_obj->my_conn_lock);
    if (conn_obj->my_conn_refcnt <= 0) {
        printf("my_conn_refrele: going to break!!\n");
```

EXAMPLE 4–1 Connection Object Definition for UAS and UAC *(Continued)*

```
        }
        assert(conn_obj->my_conn_refcnt > 0);
        conn_obj->my_conn_refcnt--;
        if (conn_obj->my_conn_refcnt > 0) {
            (void) pthread_mutex_unlock(&conn_obj->my_conn_lock);
            return;
        }
        assert(conn_obj->my_conn_flags & MY_CONN_DESTROYED);
        (void) pthread_mutex_destroy(&conn_obj->my_conn_lock);
        if (conn_obj->my_conn_local != NULL)
            free(conn_obj->my_conn_local);
        if (conn_obj->my_conn_remote != NULL)
            free(conn_obj->my_conn_remote);
        free(conn_obj);
    }

    int
    my_conn_local(sip_conn_object_t obj, struct sockaddr *sa,
        socklen_t *len)
    {
        my_conn_obj_t    *conn_obj;
        int        alen;

        conn_obj = (my_conn_obj_t *)obj;
        (void) pthread_mutex_lock(&conn_obj->my_conn_lock);
        if (conn_obj->my_conn_local == NULL) {
            (void) pthread_mutex_unlock(&conn_obj->my_conn_lock);
            return (-1);
        }
        if (conn_obj->my_conn_local->sa_family == AF_INET)
            alen = sizeof (struct sockaddr_in);
        else
            alen = sizeof (struct sockaddr_in6);

        if (*len < alen) {
            (void) pthread_mutex_unlock(&conn_obj->my_conn_lock);
            return (EINVAL);
        }
        bcopy(conn_obj->my_conn_local, sa, alen);
        (void) pthread_mutex_unlock(&conn_obj->my_conn_lock);
        *len = alen;
        return (0);
    }

    int
```

EXAMPLE 4–1 Connection Object Definition for UAS and UAC *(Continued)*

```
my_conn_remote(sip_conn_object_t obj, struct sockaddr *sa,
    socklen_t *len)
{
    my_conn_obj_t    *conn_obj;
    int        alen;

    conn_obj = (my_conn_obj_t *)obj;
    (void) pthread_mutex_lock(&conn_obj->my_conn_lock);
    if (conn_obj->my_conn_remote == NULL) {
        (void) pthread_mutex_unlock(&conn_obj->my_conn_lock);
        return (-1);
    }
    if (conn_obj->my_conn_remote->sa_family == AF_INET)
        alen = sizeof (struct sockaddr_in);
    else
        alen = sizeof (struct sockaddr_in6);

    if (*len < alen) {
        (void) pthread_mutex_unlock(&conn_obj->my_conn_lock);
        return (EINVAL);
    }
    bcopy(conn_obj->my_conn_remote, sa, alen);
    (void) pthread_mutex_unlock(&conn_obj->my_conn_lock);
    *len = alen;

    return (0);
}

int
my_conn_send(const sip_conn_object_t obj, char *msg, int msglen)
{
    my_conn_obj_t    *conn_obj;
    size_t        nleft;
    int        nwritten;
    const char    *ptr;
    socklen_t    tolen;

    conn_obj = (my_conn_obj_t *)obj;
    if (conn_obj->my_conn_fd == NULL)
        return (EINVAL);

    ptr = msg;
    nleft = msglen;

    if (conn_obj->my_conn_remote->sa_family == AF_INET)
```

EXAMPLE 4–1 Connection Object Definition for UAS and UAC *(Continued)*

```
            tolen = sizeof (struct sockaddr_in);
        else
            tolen = sizeof (struct sockaddr_in6);

        while (nleft > 0) {
            if (conn_obj->my_conn_transport == IPPROTO_UDP) {
                if ((nwritten = sendto(conn_obj->my_conn_fd,
                    ptr, nleft, 0, conn_obj->my_conn_remote,
                    tolen)) <= 0) {
                    return (-1);
                }
            } else {
                if ((nwritten = write(conn_obj->my_conn_fd, ptr,
                    nleft)) <= 0) {
                    return (-1);
                }
            }
            nleft -= nwritten;
            ptr += nwritten;
        }
        return (0);
    }
```

Additionally, both UAS and UAC register the following transaction and dialog state transition callback functions:

EXAMPLE 4–2 Transaction and Dialog State Transition Callback Functions for UAS and UAC

```
void
    ulp_dialog_state_cb(sip_dialog_t dialog, sip_msg_t sip_msg,
        int pstate, int nstate)
    {
        printf("\t\t\t%p %d ==> %d\n", dialog, pstate, nstate);
    }

    void
    ulp_trans_state_cb(sip_transaction_t sip_trans, sip_msg_t sip_msg,
        int pstate, int ostate)
    {
        char            *bid;
        sip_method_t    method;
        int        err;

        /* Not checking for err in the following functions */
```

EXAMPLE 4–2 Transaction and Dialog State Transition Callback Functions for UAS and UAC *(Continued)*

```
        if (sip_msg != NULL) {
            if (sip_msg_is_request(sip_msg, &err)) {
                method = sip_get_request_method(sip_msg, &err);
            } else {
                method = sip_get_callseq_method(sip_msg, NULL,
                    &err);
            }
        }
        bid = sip_get_trans_branchid(sip_trans);
        printf("\tTransaction (%s) %s\n\t\t\t%d ==> %d\n",
            sip_msg == NULL ? "Null": sip_method_to_str(method),
            bid, pstate, ostate);
        free(bid);
    }
```

EXAMPLE 4–3 Sample UAS Callback Reception Code

The UAS is a simple server that waits for an INVITE message and responds with a 2xx response and gets an ACK request back.

```
    void
    my_ulp_recv(sip_conn_object_t obj, sip_msg_t msg,
        sip_dialog_t sip_dialog) {
        sip_msg_t    sip_msg_resp;
        int        resp_code;
        int        error;
        sip_method_t    method;
        char        *totag;

        /* Drop if not a request */
        if (!sip_msg_is_request(msg, &error))
            return;

        method = sip_get_request_method(msg, error);
        if (error != 0)
            return; /* error getting request method);
        if (method == ACK) {
            printf("ACK received\n");
            return;
        }

        if (method != INVITE) {
            printf("not processing %d request\n", method);
            return;
```

EXAMPLE 4–3 Sample UAS Callback Reception Code *(Continued)*

```
        }

        /* Create an OK  response */
        resp_code = SIP_OK;

        /* This will probably not be  done for each incoming request */
        totag = sip_guid();
        if (totag == NULL) {
            printf("error generating TO tag\n");
            return;
        }
        sip_msg_resp = sip_create_response(msg, resp_code,
            sip_get_resp_desc(resp_code), totag,
            "sip:mycontactinfo@123.1.1.4");
        if (sip_msg_resp == NULL) {
            printf("error creating response\n");
            return;
        }

        /* send message statefully */
        sip_sendmsg(obj, sip_msg_resp, sip_dialog, SIP_SEND_STATEFUL);

        /* free message */
        sip_free_msg(sip_msg_resp);

        /* free totag */
        free(totag);
    }

    /*
     * Main program:
     * Stack initialization:
     *        Stack maintains dialogs.
     *        Dialog and transaction state transition callbacks
     *        registerd.
     */
    sip_stack_init_t    sip_init[1];
    sip_io_pointers_t    sip_io[1];
    sip_ulp_pointers_t    sip_ulp;

    bzero(sip_init, sizeof (sip_stack_init_t));
    bzero(sip_io, sizeof (sip_io_pointers_t));

    sip_io->sip_conn_send = my_conn_send;
    sip_io->sip_hold_conn_object = my_conn_refhold;
```

EXAMPLE 4–3 Sample UAS Callback Reception Code *(Continued)*

```
sip_io->sip_rel_conn_object = my_conn_refrele;
sip_io->sip_conn_is_stream = my_conn_isstream;
sip_io->sip_conn_is_reliable = my_conn_isreliable;
sip_io->sip_conn_remote_address = my_conn_remote;
sip_io->sip_conn_local_address = my_conn_local;
sip_io->sip_conn_transport = my_conn_transport;

sip_init->sip_version = SIP_STACK_VERSION;
sip_init->sip_io_pointers = sip_io;
bzero(&sip_ulp, sizeof (sip_ulp_pointers_t));
sip_ulp.sip_ulp_recv = my_ulp_recv;
sip_init->sip_stack_flags |= SIP_STACK_DIALOGS;
sip_ulp.sip_ulp_dlg_state_cb = my_dialog_cb;
sip_ulp.sip_ulp_trans_state_cb = ulp_trans_state_cb;
sip_init->sip_ulp_pointers = &sip_ulp;

/* Open a socket and accept a connection */
sock = socket(af, SOCK_STREAM, IPPROTO_TCP);
/* Check for socket creation error */

/* onoff is set to 1 */
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &onoff, sizeof (onoff));
/* check for setsockopt() error */

/* fill in bind information in sockaddr_in struct sa */
bind(sock, sa, slen);
/* check for bind error */

listen (sock, 5);

accept_fd = accept(sock, NULL, NULL);
/* check for accept error */

/*
 * create a connection object, nobj is of type my_conn_obj_t
 */
nobj = malloc(sizeof (my_conn_obj_t));
/* check for memory failure */

nobj->my_conn_fd = accept_fd;
nobj->my_conn_transport = IPPROTO_TCP;
nobj->my_conn_refcnt = 1;

/* set address family in nobj->my_conn_af */
```

EXAMPLE 4–3 Sample UAS Callback Reception Code *(Continued)*

```
/* Initialize lock */
(void) pthread_mutex_init(&nobj->my_conn_lock, NULL);

/* Set local and remote addresses in nobj */

/* INITIALIZE connection object */
sip_init_conn_object((sip_conn_object_t)nobj);

/* Termination not shown */
for (;;) {
    /*
     * Read incoming message on the connection object
     * my_conn_receive(), not shown, is an application function that
     * reads on nobj->my_conn_fd into buf, nread is the length of
     * the message read.
     */
    nread = my_conn_receive(cobj, buf, MY_BUFLEN);
    /* check for any error */

    /* Call into the SIP stack for processing this message */
    sip_process_new_packet((sip_conn_object_t)cobj, buf, nread);
}
```

EXAMPLE 4–4 Sample UAC Reception Code

The UAC is a simple client that sends an INVITE and responds with an ACK when it gets a 2xx final response.

```
/* dialog not used */
void
my_ulp_recv(sip_conn_object_t obj, sip_msg_t msg,
    sip_dialog_t sip_dialog)
{
    sip_msg_t               ack_msg;
    uint32_t                resp;
    int                     resp_code;
    int            err;
    int            transport;

    /* Not checking for err */
    if (sip_msg_is_request(msg, &err)) {
        printf("received request\n");
        return;
    }
```

EXAMPLE 4–4 Sample UAC Reception Code *(Continued)*

```
        resp_code = sip_get_response_code(msg);
        if (sip_get_callseq_method(msg, NULL, &err) != INVITE) {
            printf("received response non-invite\n");
            return;
        }

        if (!SIP_OK_RESP(resp_code)) {
            printf("received non-2xx response %d\n", resp_code);
            return;
        }

         ack_msg = sip_new_msg();
         /* check for null ack_msg */

        transport = my_conn_transport(obj);

        sip_create_OKack(msg, ack_msg,
            sip_proto_to_transport(transport));
        /* check for failure */


        /* Send the ACK message */
        sip_sendmsg(obj, ack_msg, sip_dialog, SIP_SEND_STATEFUL);

        /* free message */
        sip_free_msg(ack_msg);
    }

    /* Function to create an request */
    sip_msg_t
    my_get_sip_req_msg(sip_method_t method, char *from_name,
        char *from_uri, char *to_name, char *to_uri, char *from_tag,
        char *to_tag, char *contact, char *transport, char *via_param,
        char *sent_by, uint_t max_forward, uint_t cseq, char *callid)
    {
        sip_msg_t    sip_msg;

        sip_msg = sip_new_msg();
        /* check for memory failure */

        /* Check for failure in each case below */

        /* Add request line */
        sip_add_request_line(sip_msg, method, to_uri);
```

EXAMPLE 4–4 Sample UAC Reception Code *(Continued)*

```
        /* Add FROM */
        sip_add_from(sip_msg, from_name, from_uri, from_tag, B_TRUE,
            NULL);

        /* Add TO */
        sip_add_to(sip_msg, to_name, to_uri, to_tag, B_TRUE, NULL);

        /* Add CONTACT */
        sip_add_contact(sip_msg, NULL, contact, B_TRUE, NULL);

        /* Add VIA, no port */
        sip_add_via(sip_msg, transport, sent_by, 0, via_param);

        /* Add Max-Forwards */
        sip_add_maxforward(sip_msg, max_forward);

        /* Add Call-Id */
        sip_add_callid(sip_msg, callid);

        /* Add Cseq */
        sip_add_cseq(sip_msg, method, cseq);
    }


    /*
     * Main program:
     * Stack initialization:
     *        Stack maintains dialogs.
     *        Dialog and transaction state transition callbacks
     *        registerd.
     */
    sip_stack_init_t    sip_init[1];
    sip_io_pointers_t    sip_io[1];
    sip_ulp_pointers_t    sip_ulp;

    bzero(sip_init, sizeof (sip_stack_init_t));
    bzero(sip_io, sizeof (sip_io_pointers_t));

    sip_io->sip_conn_send = my_conn_send;
    sip_io->sip_hold_conn_object = my_conn_refhold;
    sip_io->sip_rel_conn_object = my_conn_refrele;
    sip_io->sip_conn_is_stream = my_conn_isstream;
    sip_io->sip_conn_is_reliable = my_conn_isreliable;
    sip_io->sip_conn_remote_address = my_conn_remote;
    sip_io->sip_conn_local_address = my_conn_local;
```

EXAMPLE 4–4 Sample UAC Reception Code *(Continued)*

```
sip_io->sip_conn_transport = my_conn_transport;

sip_init->sip_version = SIP_STACK_VERSION;
sip_init->sip_io_pointers = sip_io;
bzero(&sip_ulp, sizeof (sip_ulp_pointers_t));
sip_ulp.sip_ulp_recv = my_ulp_recv;
sip_init->sip_stack_flags |= SIP_STACK_DIALOGS;
sip_ulp.sip_ulp_dlg_state_cb = my_dialog_cb;
sip_ulp.sip_ulp_trans_state_cb = ulp_trans_state_cb;
sip_init->sip_ulp_pointers = &sip_ulp;

/* Open a socket and accept a connection */
sock = socket(af, SOCK_STREAM, IPPROTO_TCP);
/* Check for socket creation error */

/* connect to peer - sa is filled appropriately with peer info. */
connect(sock, sa, slen);
/* check for connect failure */


/*
 * create a connection object, nobj is of type my_conn_obj_t
 */
obj = malloc(sizeof (my_conn_obj_t));
/* check for memory failure */

obj->my_conn_fd = accept_fd;
obj->my_conn_transport = IPPROTO_TCP;
obj->my_conn_refcnt = 1;

/* set address family in obj->my_conn_af */

/* Initialize lock */
(void) pthread_mutex_init(&obj->my_conn_lock, NULL);

/* Set local and remote addresses in obj */

/* INITIALIZE connection object */
sip_init_conn_object((sip_conn_object_t)obj);

/* Create an INVITE request */
req_msg = my_get_sip_req_msg(INVITE, "me", "sip:me@mydomain.com",
    "you", "sip:user@example.com", "tag-from-01", NULL,
    "sip:myhome.host.com",
    sip_proto_to_transport(obj->my_conn_transport),
```

EXAMPLE 4–4 Sample UAC Reception Code *(Continued)*

```
        "branch=z9hG4bK-via-01", "123.1.2.3", 70, 111, NULL);

    /* Send request statefully */
    sip_sendmsg((sip_conn_object_t)obj, req_msg, NULL, SIP_SEND_STATEFUL);

    /* free msg */
    sip_free_msg(req_msg);


    /* Termination condition not shown */
    for (;;) {
        /*
         * Read incoming message on the connection object
         * my_conn_receive(), not shown, is an application function that
         * reads on nobj->my_conn_fd into buf, nread is the length of
         * the message read.
         */
        nread = my_conn_receive(obj, buf, MY_BUFLEN);
        /* check for any error */

        /* Call into the SIP stack for processing this message */
        sip_process_new_packet((sip_conn_object_t)obj, buf, nread);
    }
```

◆ ◆ ◆ APPENDIX A

Transaction Timers

TABLE A–1 SIP Transaction Timers

Timer	Default Value	RFC 3261 Section	Meaning
T1	500 milliseconds	17.1.1.1	RTT Estimate
T2	4 seconds	17.1.2.2	Maximum retransmit interval for non-`INVITE` requests and `INVITE` responses
T4	5 seconds	17.1.2.2	Maximum duration that a message remains in the network
Timer A	T1	17.1.1.2	`INVITE` request retransmit interval, for UDP only
Timber B	64*T1	17.1.1.2	`INVITE` transaction timeout timer
Timer C	> 3 minutes	16.6 (bullet 11)	Proxy `INVITE` transaction timeout
Timer D	> 32 seconds for UDP, 0 for TCP/SCTP	17.1.1.2	Wait time for response retransmits
Timer E	T1	17.1.2.2	Non-`INVITE` request retransmit interval, for UDP only
Timer F	64 * T1	17.1.2.2	Non-`INVITE` transaction timeout timer
Timer G	T4 for UDP, 0 for TCP/SCTP	17.2.1	`INVITE` response retransmit interval

TABLE A–1 SIP Transaction Timers *(Continued)*

Timer	Default Value	RFC 3261 Section	Meaning
Timer H	64 * T1	17.2.1	Wait time for `ACK` receipt
Timer I	T4 for UDP, 0 for TCP/SCTP	17.2.1	Wait time for `ACK` retransmits
Timer J	64 * T1 for UDP, 0 for TCP/SCTP	17.2.2	Wait time for non-`INVITE` request retransmits
Timer K	T4 for UDP, 0 for TCP/SCTP	17.1.2.2	Wait time for response retransmits