

SIP API for Solaris

sip_api_team@sun.com
Network Approachability
Sun Microsystems Inc.

Revision 1.1
Oct. 29, 2005

Table of Contents

I. Introduction	2
II. Overview	2
III. SIP API Internals	6
SIP stack initialization	6
Header Management Layer	8
Transaction Management Layer	13
Dialog Management Layer	16
Message Formatting Layer	19
Connection Manager	20
Timer Management Layer	23
Generating Call-ID, From and To tags, Branch-ID and Cseq	24
IV. Multithreading	24
V. URI support	25
VI. Open Issues	26
Appendix I. Stack Configurations	27
Appendix II. External Interfaces	31
Appendix III. External Data structures	35
Appendix IV. Examples	39
Appendix V. Transaction Timers	44

List of Figures

Transaction Call Stateful	27
Stateless	28
Transaction Stateful	29
Call Stateful	30

I. Introduction

Session Initiation Protocol (SIP) is a signaling protocol used to set up and tear down multimedia sessions like Voice over IP (VoIP) and Instant Messaging (IM). This document discusses the design and relevant implementation details of SIP API for Solaris.

II. Overview

The objective is to define an extensible API to enable development of simple and involved SIP applications. The underlying SIP stack should be modular and should be flexible enough for a user to provide his/her own functionality, if needed.

The API is implemented as a passive user library that requires applications to register certain mandatory and optional functions. The mandatory functions are necessary for sending messages to the peer, receiving messages, and functions for performing connection object related operations. The optional functions include providing timeout/un-timeout functionality, callbacks for error notification etc. If the application does not register any of the optional functions, the library uses the built-in one, if present.

The library consists of the following distinct operational components:

Header Management Layer

The Header Management layer provides the interfaces required to build, parse, examine and validate SIP headers.

Transaction Management Layer

SIP is based on a HTTP-like request/response transaction model. A transaction includes a request sent by a client transaction (using the transport layer) to a server transaction, and all responses to that request sent back from the server transaction. The Transaction Management layer handles application-level retransmissions, matching responses to requests, and application-level timeouts. Any task that a user agent client (UAC) accomplishes takes place using a series of transactions.

Dialog Management Layer

A dialog is a peer-to-peer SIP relationship between two user agents that persists for some time. A dialog facilitates sequencing of messages and proper routing of requests between the user agents.

Use of dialogs is optional. Dialogs are not needed for setting up a SIP session. Dialogs, when present, hold state information, which can be used to construct a request within a session. An application has the option of implementing its own Dialog Management layer instead of using the one provided by the library. If the library is configured to manage dialog it automatically creates and updates dialogs for incoming and outgoing SIP messages, and also delivers the matching dialog (to the application) along with the incoming message.

Message Formatting Layer

For an incoming message, the Message Formatting layer constructs a SIP message before delivering it to the application. If the underlying transport is TCP, this layer also breaks the byte stream at message boundaries using the Content-Length header (from the SIP message). If a transaction exists, the message is passed to the Transaction layer for further processing before sending it to the Dialog Management layer, if present, and the application.

On the outbound side, the library adds a Content-Length header followed by an empty line (as per RFC 3261). As with an incoming message, this message is passed to the Transaction layer and Dialog layer, if present. The message is then sent to the application's send function after copying all 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 has an option of implementing its own timeout/un-timeout routines and registering them with the library, instead of using the ones provided by the library.

Connection Manager

The Connection Manager provides I/O functionality. It is not part of the SIP library, but interacts with the library using well defined interfaces. A connection, identified by local and remote endpoints and the transport, is represented by a connection object. The library does not impose any structure on the connection object, except that the first member of the connection object must be a void pointer. The library uses this first member to store library specific data for a connection. To this end the application must initialize the connection object using a library provided function. The application is responsible for the following:

Compliance with RFC 3263

The library expects the application to supply the address of the next recipient of the SIP message and handle network errors. Therefore, it is up to the application to follow the procedures defined in RFC 3263 for locating SIP servers.

Sent-By and Received parameters

RFC 3261 (section 18.1.1) states that the Transport layer must add a “sent-by” parameter to the topmost VIA header in every request. The application should include this information when constructing the VIA header. The application must register all values it could use for the sent-by parameter with the library, so that the library can validate incoming responses as per RFC 3261 (section 18.1.2).

RFC 3261 (section 18.2.1) lists conditions under which the Transport layer must add a “received” parameter to the top VIA header of incoming message indicating the source of the message (section 18.2.2 discusses the use of this information). The library does not add a received parameter as it caches the connection object (in the transaction) on which the original transaction-creating request was received. To satisfy the requirement of the RFC, retransmissions within a transaction are first

attempted using the cached connection object before using the connection object that is passed in (the fallback occurs in case of network error).

TTL and maddr parameters

The application should add the TTL and maddr parameters in the VIA header when required.

Different stack configurations are shown in **Appendix I**.

The library supports the following:

Multithreading:

The library is MT-safe. Multiple threads can simultaneously perform operations on headers of a message. Each message maintains a reference count and is only freed when the message is deleted and the reference count drops to 0. The library provides no protection against an application thread reducing the reference count to 0 and the message being deleted when other threads are still using it.

URI:

The library supports all URI types defined in section 25 of RFC 3261. Access functions are made available for individual components of a URI.

Logging/Tracing:

The library currently does not include a logging/tracing mechanism, however the design will be updated to include this in subsequent revisions.

III. Solaris SIP API Internals

The following sections describe various components of the library in detail. The interfaces described in this section are internal to the library unless otherwise specified. For a list of external interfaces and data structures, refer to **Appendix II** and **Appendix III** respectively.

SIP Stack Initialization

An application initializes the stack before anything else. The initialization parameters can be broadly subdivided into:

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

Generic stack parameters.

SIP version

Currently – `SIP_STACK_VERSION_1` (which is defined as 1)

Stack flags

The only currently defined flag is `SIP_STACK_DIALOGS`, which instructs the library to maintain dialogs. If this flag is not set, the library does not maintain any dialog information.

SIP Timers

Application can provide values for the following timers (see **Appendix V** for details). A value of 0 for any of these means the library will use default values.

`sip_timer_T1`

`sip_timer_T2`

`sip_timer_T4`

`sip_timer_TD`

Upper layer registrations

Upper layer receive routine

The library delivers SIP messages to the application using this receive function. It is mandatory to register this.

Application specific timeout and un-timeout routines

The application can provide its own timeout and un-timeout routines for the library to use. If one is provided, so must be the other. It is optional to provide these. If the application does not provide these, the library will use the built-in timeout routines.

Transactions error notifications

The application can optionally register a routine that the library will invoke when the Transaction layer encounters a network error when sending a SIP message. If the application does not provide this, there will be no notifications sent in case of a network error.

Dialog delete notification

If the library is configured to maintain dialogs, the application can optionally provide a callback routine, which the library will invoke when a dialog is deleted. If the application does not provide this, there will be no notifications sent when a dialog is deleted.

Connection Manager Interfaces

It is mandatory for the application to register all the following:

Send routine

The library calls this to send the SIP message out.

Hold/Release functions

Since the library caches connection objects, the application must provide functions to increment/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` : is connection byte-stream (e.g. TCP).
- `is_reliable` : is connection reliable (e.g. TCP, SCTP)
- remote address: peer endpoint information.
- local address : local endpoint information.
- transport : type of transport (TCP/UDP/SCTP)

Custom header table

An application can optionally register a table of custom headers along with parsing functions for the same. The application can also include standard headers in the table, in which case the user supplied parsing functions will be used instead of the built-in ones.

The stack initialization structure is shown in **Appendix III**.

After initializing the stack, an application can create requests using the interfaces provided by the Header Management layer and send them out using the interface provided by the Message Formatting layer. Likewise, the application can receive incoming request/response after initializing the stack and pass them to the library for processing. These are described in the following sub-sections.

Header Management Layer

This layer provides interfaces that allow an application to create, parse, modify and examine SIP messages. A SIP message consists of a start line, a variable number of headers and, optionally, a message body. The start line and headers are terminated by a single Carriage Return/Line Feed (CRLF), while the message body is preceded by an empty line containing only a CRLF. SIP allows combining multiple headers of the same (name) type under one header, so a single header can have multiple values. A header value consists of a number of descriptive parameters and an optional name-value pair. E.g. the Backus-Naur Form (BNF) for a VIA header is defined as (RFC 3261, section 25):

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

via-parm are descriptive parameters and *via-params* are name-value pairs. The combination of *via-parm* and *via-params* constitute the value of the header. The library maintains a *sip_<header_name>_value_t* for each SIP header (e.g. for VIA it would be *sip_via_value_t*). A header may have multiple comma separated values, in which case

the values are a list of *sip_<header_name>_value_t* for that header; in the VIA header example, each *sip_via_value_t* structure would correspond to a *via-parm*.

When a message is received, the Message Formatting layer transforms it into a SIP message, *_sip_msg_t*, defined as:

```
typedef struct sip_message {
    char          *sip_msg_buf;
    char          *sip_msg_oldbuf;
    boolean_t     sip_msg_modified;
    boolean_t     sip_msg_cannot_be_modified;
    int           sip_msg_len;
    size_t        sip_msg_content_len;
    sip_content_t sip_msg_content;
    pthread_mutex_t sip_msg_mutex;
    _sip_header_t *sip_msg_headers_start;
    _sip_header_t *sip_msg_headers_end;
    _sip_header_t *sip_msg_start_line;
    sip_message_type_t *sip_msg_req_res;
    int           sip_msg_ref_cnt;
}_sip_msg_t;
```

The SIP message contains a list of all the headers and the content. This SIP message is then passed to the application using the application-registered receive function. The application can process the message by querying for headers, values in any header, generating a response (if it is a request) etc.

Internally, the SIP message, *_sip_msg_t*, forms the topmost structure in the header hierarchy. SIP headers are stored in a linked list of *sip_header_t* structures given by *sip_msg_headers_start*. Functions working on *_sip_msg_t* use *sip_msg_mutex* to synchronize access to any member of the SIP message, including headers and their values. *sip_msg_ref_cnt* is used to track the users of the SIP message; when deleted, the reference count is decremented and the message is destroyed only if this count falls to 0.

The library allows a received SIP message to be forwarded to a SIP entity with or without modifications. Once a message has been sent, it cannot be modified because it is required for future reference (*sip_msg_modified* and *sip_msg_cannot_be_modified* are used to enforce this).

A SIP message consists of various SIP headers, each of which is represented by *_sip_header_t*, defined as:

```
typedef struct sip_header {
    sip_hdr_general_t    sip_hdr_general;
    int                  sip_header_state;
    struct sip_header    *sip_hdr_next;
    struct sip_header    *sip_hdr_prev;
    struct sip_message   *sip_hdr_sipmsg;
    boolean_t            sip_hdr_allocated;
    sip_header_function_t *sip_header_functions;
}_sip_header_t;

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*, that indicates the start, end and the current position (during processing) for the header . *sip_hdr_allocated* indicates that the header was allocated and needs to be freed, this will be the case when a header is added and was not part of a received message. The header also contains a back-pointer to the SIP message. *sip_header_functions* points to a table that has an entry for every header defined in RFC 3261 as well as an entry for type “unknown” for unrecognized headers. Once parsed, the parsed header is cached in *sip_hdr_parsed* for future reference.

SIP allows inclusion of custom message headers not defined in RFC 3261. As mentioned in the previous section (“SIP Stack Initialization”), an application can provide its own table of custom headers and supporting parsing functions. The application provides a table/array of *sip_header_function_t* for this purpose. The application provided function table entries have precedence over the same header entries

in the default function table, if present.

When a header is deleted by the application, *sip_header_state* is set to *SIP_HEADER_DELETED* to indicate this. Existing references to the header remain, but new lookups will not return the deleted header. All headers are freed when the message is freed.

A parsed SIP header is represented by:

```
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*, which is returned as a result of parsing a *_sip_header_t*. Regardless of the *<header_name>*, the first field of that structure will always be a *sip_value_t*, defined as:

```
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;
```

param_list is the list of parameters for this value. *sip_param_t* is defined as:

```
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;
```

As mentioned earlier, each header in RFC 3261 has an associated value defined, e.g., *sip_via_value_t* is defined as:

```

typedef struct via_value {
    sip_value_t      sip_value;
    sip_proto_version_t sent_protocol;
    sip_str_t        sent_by_host;
    int              sent_by_port;
}sip_via_value_t;

typedef struct sip_proto_version_s {
    sip_str_t        name;
    sip_str_t        version;
    sip_str_t        transport;
}sip_proto_version_t;

```

When a value is deleted from a header, *sip_header_state* is set to *SIP_HEADER_DELETED_VAL* to indicate this in addition to setting *value_state* to *SIP_VALUE_DELETED*. Existing references to the deleted value remain, while new lookups ignore the deleted value.

The hierarchy to traverse in order to get to a value from the SIP message is as follows:

```

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

```

Freeing any structure using its corresponding free function will result in all the underlying structures being freed.

Note that the library performs lazy parsing of headers. Only headers whose value(s) are accessed by the application are parsed and checked for compliance. This approach is adopted to comply with section 16.3 in RFC 3261 - “Request Validation”, “Reasonable syntax check”. If a value is invalid, *value_state* is set to *SIP_VALUE_BAD* indicate this.

Writing Parsers for Custom Headers

An application can provide custom (or standard) headers and register supporting parsing functions. Parsing functions can be written using the following:

sip_header_general_t

Passed in to provide the start and end of a header within a message. The parsed header element is set by the parser to point to *sip_parsed_header_t* that it allocates.

sip_parsed_header_t

The parser allocates this structure and assigns it. This structure is passed back to the caller.

sip_value_t

The application defines the value depending on the header. The library only requires that the first element of the structure be *sip_value_t*. The parser creates a linked list of values, if there are multiple values, and sets the value field in the parsed header to the first value. The application provides any access function, if required, for value members.

Transaction Management Layer

The Transaction Management layer is responsible for creating and maintaining transaction states for both clients and servers as defined in RFC 3261, section 17. This section describes the design and related details of the Transaction layer.

The library maintains a hash table of transactions that are indexed by the MD5 hash of either the branch ID (for SIP messages compliant with RFC 3261) or a combination of the branch ID, Call-ID, From, To, and Cseq fields (for SIP messages compliant with RFC 2543). The branch ID in the topmost VIA header is used to identify if a message complies with RFC 3261 or RFC 2543 (for a message compliant with RFC 3261, the branch ID has a prefix of “z9hG4bK”).

Internally, a transaction is defined as:

```
typedef struct sip_xaction_s {
    char                *sip_xaction_branch_id; /* Transaction id */
    uint16_t            sip_xaction_hash_digest[8];

    _sip_msg_t          *sip_xaction_orig_msg; /* orig request msg. */
    _sip_msg_t          *sip_xaction_last_msg; /* last msg sent */
    sip_conn_object_t   sip_xaction_conn_obj;
    int                 sip_xaction_state; /* Transaction State */
    sip_method_t        sip_xaction_method;

    uint32_t            sip_xaction_ref_cnt;

    pthread_mutex_t     sip_xaction_mutex;

    /* Timer Information */
    sip_timer_t         sip_xaction_TA; /* Initial T1 = 500 ms */
    sip_timer_t         sip_xaction_TB; /* Default 64 * T1 */
    sip_timer_t         sip_xaction_TD; /* Default 32s - UDP */
    sip_timer_t         sip_xaction_TE; /* Initial T1 */
    sip_timer_t         sip_xaction_TF; /* Default 64 * T1 */
    sip_timer_t         sip_xaction_TG; /* Initial T1 */
    sip_timer_t         sip_xaction_TH; /* Default 64 T1 */
    sip_timer_t         sip_xaction_TI; /* Default T4 = 5 s - UDP */
    sip_timer_t         sip_xaction_TJ; /* Default 64 * T1 */
    sip_timer_t         sip_xaction_TK; /* Default T4 - UDP */
    void                *sip_xaction_ctxt; /* currently unused */
} sip_xaction_t;

/* Structure for SIP timers */
typedef struct sip_timer_s {
    int                 sip_timerid;
    struct timeval      sip_timeout_val;
} sip_timer_t;
```

Functions working with transactions use `sip_xaction_mutex` to synchronize access to members. `sip_xaction_ref_cnt` tracks the users of a transaction. A successful lookup for a transaction always increments the reference count, it is the responsibility of the caller to decrement the reference count after use.

The Transaction layer is initialized, as part of the stack initialization, with the transaction error callback, if provided by the application.

Transaction creation and maintenance

For incoming requests and responses the transaction hash table is scanned for an existing transaction. If a transaction does not exist for an incoming response, the transaction is handled statelessly; while for an incoming request, the application will inform the stack whether to create a transaction when sending the response. In either case the message is passed to the application. If a request matches a transaction, it is considered a retransmission and the last response is retransmitted by the Transaction layer. The incoming request is dropped (i.e. not passed to the application). The exceptions to this are a CANCEL request and an ACK for non-2xx response. A CANCEL request will match the INVITE transaction it is cancelling and an ACK for a non-2xx response will also match the INVITE request whose response it is acknowledging. In these two cases, the request is not a retransmission and will be sent up to the application. If a response matches a transaction, the response is processed statefully and the message is passed to the application.

For outgoing responses a transaction is created only if the application indicates so when sending the response using *sip_sendmsg()* (discussed in the *Message Formatting Layer* sub-section). Similarly, for outgoing requests, a transaction is created only if the application indicates so using *sip_sendmsg()*. The outgoing message for which the transaction is created is saved in the transaction to facilitate retransmission or to respond to retransmitted requests. For a newly created transaction, timers: A, B, D, E, F, G, H, I, J and K are initialized (see **Appendix V**).

Transaction Layer and Ack generation

A SIP entity needs to send an ACK for every final response it receives to an INVITE request. The procedure for sending the ACK depends on the type of response. For final responses between 300 and 699, the ACK processing is done by the Transaction layer. For 2xx responses, the ACK is generated by the application. In all cases the application receives the response.

Transaction Deletion

The MD5 hash stored in the transaction is used to lookup and delete it from the hash table. When the reference count falls to 0 and the transaction is in one of the terminated states (i.e. client invite terminated, client non-invite terminated, server invite terminated, server non-invite terminated), it is deleted and removed from the hash table; if the reference count is greater than 0, the transaction is not removed from the hash table, but it will not be returned in lookups.

Transaction Lookup

Internally, a transaction is retrieved by providing the SIP message. As mentioned earlier, the branch ID from the message is used to locate a transaction for messages conforming to RFC 3261 and a combination of different fields is used for messages conforming to RFC 2543. A successful lookup always increments the reference count for the transaction. A transaction that is marked as deleted is not returned via a lookup.

Transaction Timers

The Transaction layer maintains a set of timers for timing out transactions or sending retransmissions. These timers are listed in **Appendix V**. When a transaction times-out, the Transaction layer notifies the application if it has been initialized to do so.

Transaction and Network errors

When the Transaction layer has to send a message, such as retransmitting a request or a response, it uses a cached connection object (discussed in *Connection Manager* subsection). If there is a network error the Transaction layer releases its hold on the connection object and calls the application registered callback function, if provided. If the return value from the callback function is not 0 or no callback function is provided, the transaction is terminated and resources freed.

Dialog Management Layer

A Dialog represents a peer to peer relationship between two user agents that persists for some time. A dialog facilitates sequencing of messages between the user agents and proper routing of requests between them. 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.

Use of dialogs are option; examples of dialog creating methods include INVITE and SUBSCRIBE. For an INVITE, a 2xx or 101-199 response with a To tag creates a dialog. For SUBSCRIBE, a 2xx response or NOTIFY request creates a dialog. Dialogs created due to provisional responses are called EARLY dialogs, while those created by a 2xx final response are called CONFIRMED dialogs. An EARLY dialog transitions to the confirmed state when a subsequent 2xx response is received.

An application can do its own Dialog Management or delegate it to the stack. If configured to manage dialogs, the library automatically creates and maintains dialogs and also delivers any dialog, matching incoming messages, to the application. If the library is not configured to manage dialogs, there is no interaction between the application and the library with respect to dialogs.

Internally, a dialog is defined as:

```
typedef struct sip_dialog
{
    _sip_header_t          *sip_dlg_remote_uri_tag;
    _sip_header_t          *sip_dlg_local_uri_tag;
    _sip_header_t          *sip_dlg_remote_target;
    _sip_header_t          *sip_dlg_route_set;
    sip_str_t              sip_dlg_rset;
    sip_str_t              sip_dlg_req_uri;
    _sip_header_t          *sip_dlg_call_id;
    uint32_t                sip_dlg_local_cseq;
    uint32_t                sip_dlg_remote_cseq;
    uint16_t                sip_dlg_id[8];
    boolean_t              sip_dlg_j;
    dialog_state_t         sip_dlg_state;
    int                    sip_dlg_type; /* CALLEE or CALLER */
    pthread_mutex_t        sip_dlg_mutex;
    uint32_t                sip_dlg_ref_cnt;
    sip_timer_t            sip_dlg_timer; /* to delete partial dialogs */
    void                   *sip_dlg_ctxt; /* currently unused */
} _sip_dialog_t;
```

UAC dialog creation

The library creates a dialog when it receives a 1xx response with a To tag or a 2xx response to a dialog creating request for which a dialog does not already exist. The response contains all the information to create the dialog. Note that a request sent by an UAC can be forked resulting in multiple responses, which means multiple dialogs could be created as a result of sending a request. The response is then sent upstream along with the newly created dialog.

UAS dialog creation

When the library receives a dialog creating request, it creates a partial dialog using the request. This partial dialog is sent upstream along with the request. The library needs to create a partial dialog because the response (from the application) will not contain all the information needed to create the dialog. When the application sends a response downstream, the library completes the dialog. Note that the partial dialog is not inserted into any hash table, and thus will not be found as a result of a lookup. It is possible that the UAS may not respond to the request, for this reason a timer is started when the partial dialog is created. The timer is set to the duration of an INVITE transaction timeout, as specified in RFC 3261. If the UAS does not respond during this time interval, the partial dialog is deleted after calling the dialog delete callback function, if registered. The partial dialog is also destroyed if the response is not 1xx or 2xx.

Dialog Caching

As soon as the application's receive function returns or a dialog terminating request is received, the dialog is released or freed. An application can cache a dialog after incrementing the reference count on it. The library provides *sip_hold_dialog()* and *sip_release_dialog()* for this purpose. The application should also register a callback function, which will be called when the dialog is being deleted, so that it can decrement the reference count after taking appropriate action.

Dialog termination, deletion and notification

Processing a request/response may result in termination of a dialog. An early dialog terminates if the final response is not a 2xx response or if no response arrives. The

mechanism for terminating confirmed dialogs are method specific. The BYE method terminates an INVITE dialog and the session associated with it.

When a dialog is terminated, the callback function, if registered, is invoked and the dialog deleted. If the reference count on the dialog is not 0, it is marked deleted, but not destroyed. When the reference count falls to 0, the dialog is destroyed. Dialogs marked deleted are not returned by lookup functions. An application can also delete a dialog using *sip_delete_dialog()* if it does not want dialogs to be maintained for certain requests.

Message Formatting Layer

The Message Formatting layer is responsible for representing the message in a form required by the next component/layer.

If the incoming message arrives over TCP, the Message Formatting layer breaks the byte stream at message boundaries. It parses the message so that it can be represented as *_sip_msg_t*. The message is then passed on to the next layer, which could be the Transaction layer, Dialog layer or the application.

On the sending side, the message is either received from the application or the Transaction layer (e.g. retransmissions) and all headers and content are copied into a contiguous buffer before delivering it to the application's send function (provided by the Connection Manager.)

Receiving Messages

The Connection Manager delivers messages to the SIP library by calling *sip_process_new_packet()* along with the connection object. If the transport is TCP, the Message Formatting layer breaks the byte stream at message boundaries. The boundary is determined by the Content-Length header in the SIP message. A Content-Length header must be present in every message delivered over TCP (according to RFC 3261). The behavior of the library is undefined if a message is received over TCP that does not contain a Content-Length header.

The Message Formatting layer holds any excess data, that is not part of the current message, to be used with the next packet (thus, when a connection is closed or reused the Connection Manager has to inform the library so that it can free the data and resources allocated for this purpose). After parsing the message, i.e. converting it into a `_sip_msg_t`, it is delivered to the next layer for processing. The message is eventually delivered to the application by calling the receive function registered during stack initialization.

Sending Messages

An application sends a SIP message using `sip_sendmsg()` along with the connection object and any message specific flags. The library adds a Content-Length and an empty line (CRLF) to the message and delivers the packet to the next layer. The Message Formatting layer then forms a contiguous buffer using the contents of the SIP message and delivers it to the send routine provided by the Connection Manager.

Connection Manager

The Connection Manager provides I/O functionality. It is not part of the library but interacts with the library using well defined interfaces. This section describes the usage model of the Connection Manager, its interface with the library, and the requirements imposed by the library. The Connection Manager must register the following mandatory interfaces with the library 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,
                                     struct sockaddr *, socklen_t *);
int          sip_conn_local_address(sip_conn_object_t,
                                    struct sockaddr *, socklen_t *);
int          sip_conn_transport(sip_conn_object_t);
```

A connection, identified by local and remote end points and the transport, is represented by a connection object.

The only requirement the library imposes on the connection object is that the first element, a void pointer, should be reserved for use by the library. Every connection

object must be initialized by calling *sip_init_conn_object()* before use. The connection object, itself, is opaque to the library.

Connection object

An example definition of the connection object could be:

```
typedef struct my_conn_obj {
    void                *my_conn_resv;
    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;
} my_conn_obj_t;
```

As part of initializing a connection object, the library sets the first element to point to a structure to track transactions that have cached this connection object. This structure is also used for breaking a TCP stream at message boundaries.

The library does not interact with listening endpoints, so it does not impose any restriction on creating/maintaining listening endpoints.

The application calls a Connection Manager specific routine to get a connection. It is the responsibility of the Connection Manager to ensure that there is a unique connection object for every remote address, local address, and transport tuple. This is important for UDP where the underlying endpoint does not uniquely identify a local/remote endpoint pair.

Caching connection object

Internally, the library caches connection objects for use by the Transaction layer. The library increments the reference count on the connection object so that it is not freed when in use by the library. In the case of TCP, the library also adds a reference count on the connection object when it allocates resources to split the TCP stream at message boundaries. The Connection Manager registers *sip_hold_conn_object()/sip_rel_conn_object()* for this purpose.

Freeing connection object

The Connection manager can close a connection any time it wants, but it cannot free the connection object as long as there are existing references to it. When the Connection Manager wants to free the connection object and there are references on it, it calls a library provided function *sip_conn_destroyed()* so that the library can take appropriate action before releasing the references it holds. When *sip_conn_destroyed()* is called, the library locates the transaction(s) caching the connection object and terminates them; any resources allocated for TCP handling are also freed at this time.

Connection Lookup

The Connection Manager does not register any lookup function with the library as there is no need for the library to lookup connection objects. The library caches connection objects that are passed in when a message is delivered to the library.

Sending Messages

When an application sends a message using *sip_sendmsg()*, the library internally calls the Connection Manager registered function, *sip_conn_send()*, after processing the outbound message. In case of a network error, the Connection Manager is free to resolve the issue in any manner it deems fit, including returning an error. The only requirement is that the behavior be consistent for all subsequent calls using the same connection object. One possible solution might be to create a new connection and update the existing one. However, for a TCP connection, there might be data in the connection object as a result of breaking TCP stream at message boundaries. This stale data must be flushed by calling a library provided routine *sip_clear_stale_data()*. The Connection Manager must not change the transport type of a connection object.

Receiving Messages

The Connection Manager delivers new packets to the library by calling *sip_process_new_packet()*. After processing the message, the library calls the application registered receive callback function to pass the message to the application. The message and connection object are freed/released upon return from the application's receive function. If the application queues the message it should manage the reference

counts on the connection object, as described in “*Caching connection objects*” above, and the message, using *sip_hold_msg()/sip_free_msg()*.

Transaction layer and I/O errors

When the Transaction layer has to send a message, such as retransmitting a request or a response, it uses a cached connection object. If there is a network error, the Transaction layer releases its hold on the connection object and calls the application callback function, if registered, for transaction error notification. If there is no callback function registered or if the callback returns a non-zero value, the transaction is terminated. If the return from the callback function is 0, the cached object is not released assuming everything is fine.

Timer Management Layer

The Timer Management facility mimics the timeout/un-timeout functionality of the Unix kernel. A thread maintains a time sorted list of timer objects and calls the callback function at the specified interval. The timeout object is defined as:

```
typedef struct timeout {
    struct timeout    *sip_timeout_next;
    hrtime_t          sip_timeout_val;
    void              (*sip_timeout_callback_func)(void *);
    void              *sip_timeout_callback_func_arg;
    int               sip_timeout_id;
} sip_timeout_t;
```

As part of the stack initialization, the Timer layer is initialized which results in starting the timer thread.

A timeout can be scheduled by calling *sip_timeout()*, with the callback function, the time interval after which the callback needs to be invoked and the arg to be passed to the callback function. The return value is a timeout id, which can be used to cancel the timeout.

A timeout can be cancelled by invoking *sip_untimeout()* and specifying the timeout id of the timeout to be cancelled.

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

The library provides *sip_guid()* to generate unique ids for Call-ID and tags. The id is generated using the combination of *gethrtime()* and random number obtained from */dev/urandom*. A 32-bit random number obtained from */dev/urandom* is concatenated with the upper 32-bits from *gethrtime()*. Then randomly, alphabets (lower and upper case) are used to replace numbers in the resulting string. The application is responsible for freeing the string returned by *sip_guid()*.

For generating a Branch-ID, the library provides *sip_branchid()*. If *sip_branchid()* is invoked without a SIP message or with a SIP message without a VIA header, the branch-id is formed using *sip_guid()*. If a SIP message with a VIA header is provided then the branch-id is generated using the MD5 hash of the To tag, From tag, Call-ID, Request URI, topmost VIA header, sequence number from Cseq header, and any Proxy-Require and Proxy-Authorization header fields that may be present. All branch-ids are prefixed with “z9hG4bK” to conform to RFC 3261. The application is responsible for freeing the string returned by *sip_branchid()*.

An application can use *sip_get_cseq()* to obtain the initial sequence number. *sip_get_cseq()* uses the most significant 31 bits of the value returned by *time(2)*.

IV. Multithreading

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 a header or one of its value is deleted it is marked as deleted and not available via subsequent lookups. Threads already holding references are not affected.

However, care must be taken before calling *sip_free_msg()* to free a SIP message, as it will result in deleting the message if the reference count for the message falls to 0. The reference count on a message can be incremented using *sip_hold_msg()*.

V. URI support

The library supports all the URI types defined in section 25 of RFC 3261. Internally, a URI is represented as:

```
typedef struct sip_uri_sip {
    sip_param_t    *params;
    sip_str_t      headers;
} sip_uri_sip_t;
```

```
typedef struct sip_uri_abs {
    sip_str_t      opaque;
    sip_str_t      query;
    sip_str_t      path;
    sip_str_t      regname;
} sip_uri_abs_t;
```

```
typedef struct sip_uri {
    sip_str_t      scheme;
    sip_str_t      user;
    sip_str_t      password;
    sip_str_t      host;
    uint_t        port;
    uint_t        errflags;
    boolean_t     issip;
    boolean_t     isteluser;
    union {
        sip_uri_sip_t sip;
        sip_uri_abs_t abs;
    } specific;
} _sip_uri_t;
```

Access functions are provided to obtain a URI from a SIP header and also various URI components. A SIP URI can be obtained by getting the URI value from a SIP header and then parsing it into its components using *sip_get_uri_parsed()*. For a complete list of access functions refer to **Appendix II**.

VI. Open Issues

The following are currently not addressed in this design. We are still evaluating the need for (except for logging/tracing) and/or the correct design approach for these. We are hoping to get comments on these as part of the review process.

- The library does not provide any functions to build a URI. We believe this should be part of a standard library and should not be embedded in the SIP library.
- The library does not provide any error checking on headers being added. We expect the application to know what it is adding.
- The library does not provide a mapping mechanism between a message and application context. When invoking the transaction and dialog callbacks, the library will provide the appropriate connection object as a parameter.
- Once the application registers transaction timeout/dialog delete callbacks, the library does not provide any mechanism to unregister.
- The library, as noted earlier, does not provide any logging/tracing mechanism.

Appendix I. Stack configurations

Figure 1. Transaction and Call stateful: when the stack is initialized to maintain dialogs and the request/response is sent with the SIP_SEND_STATEFUL flags set.

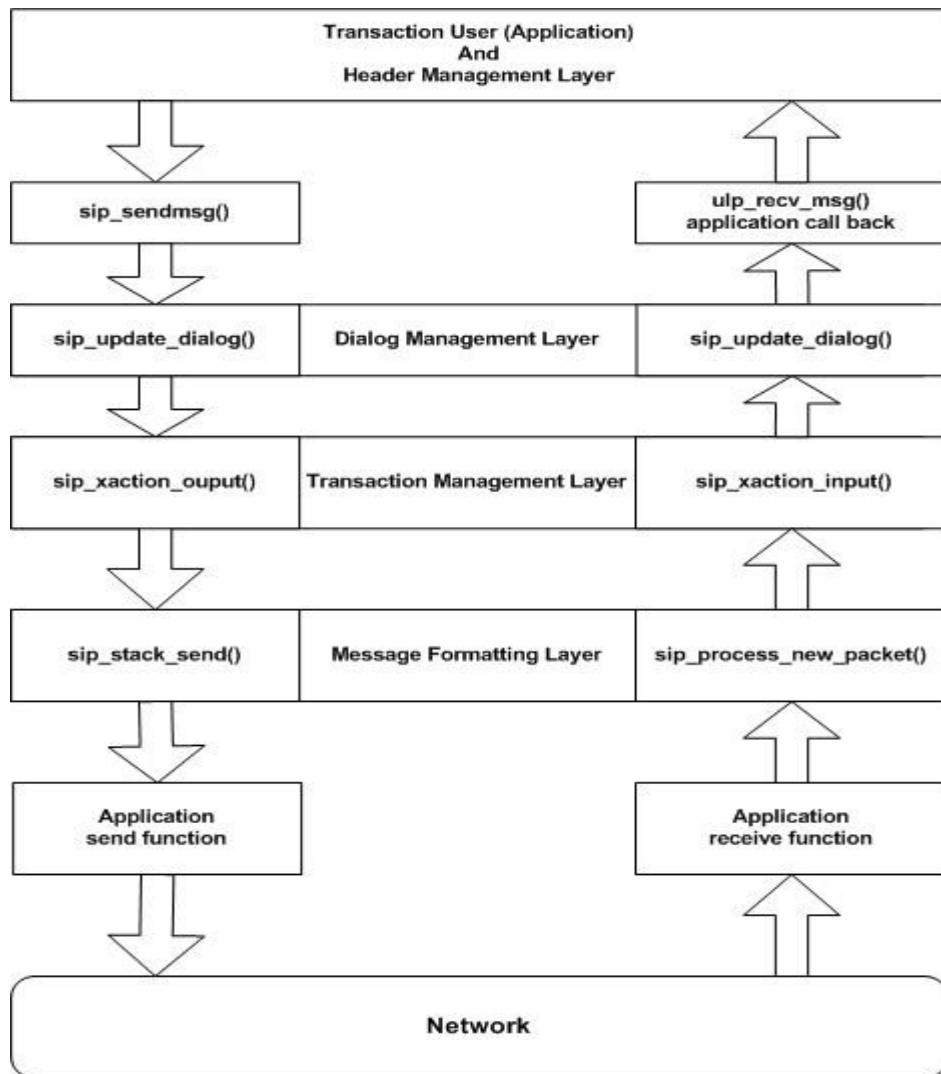


Figure 2. Stateless: When the stack does not maintain dialogs and a request/response is sent without setting the SIP_SEND_STATEFUL flag:

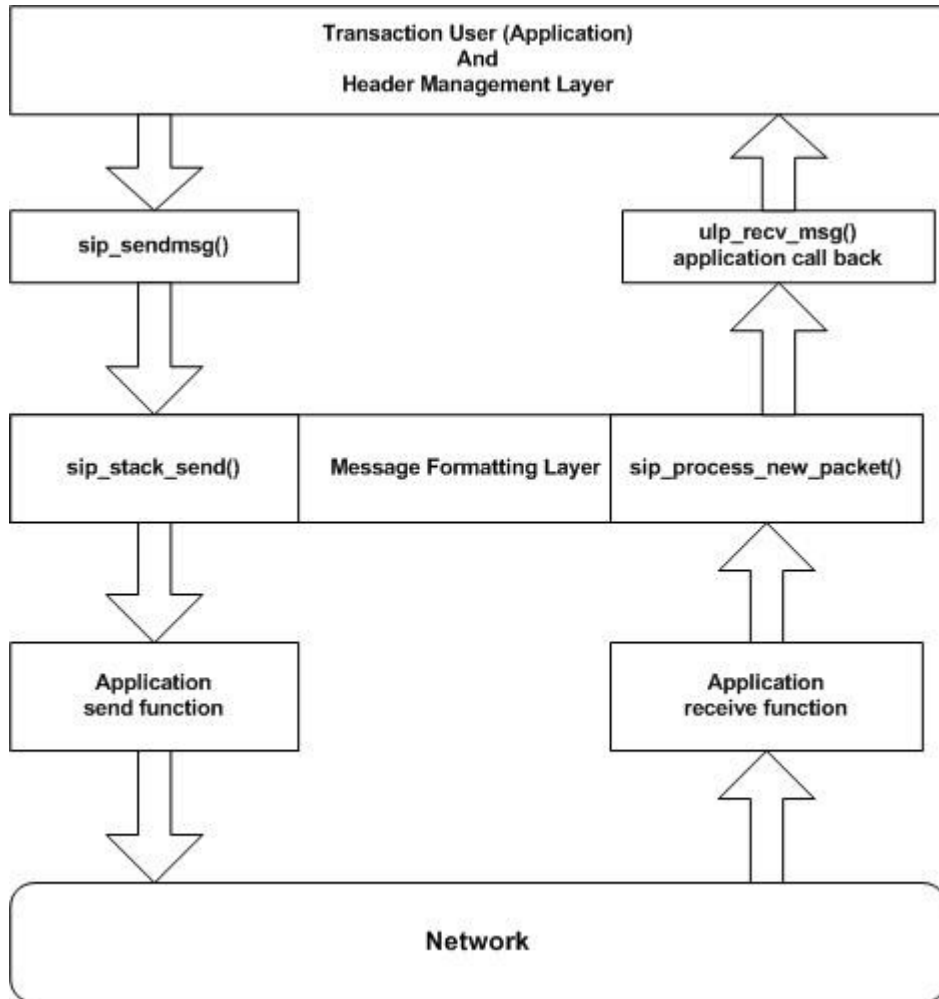


Figure 3. Transaction stateful: When the stack does not maintain dialogs and a request/response is sent with the SIP_SEND_STATEFUL flags set.

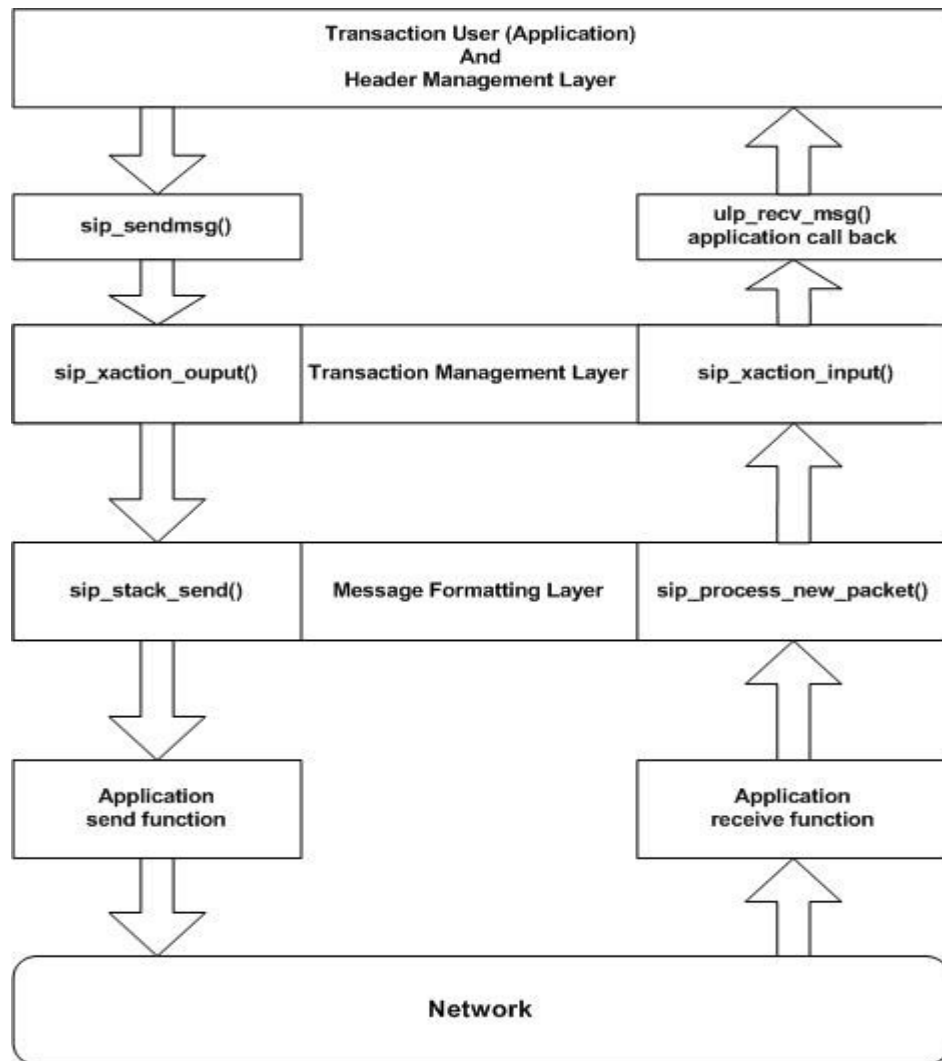
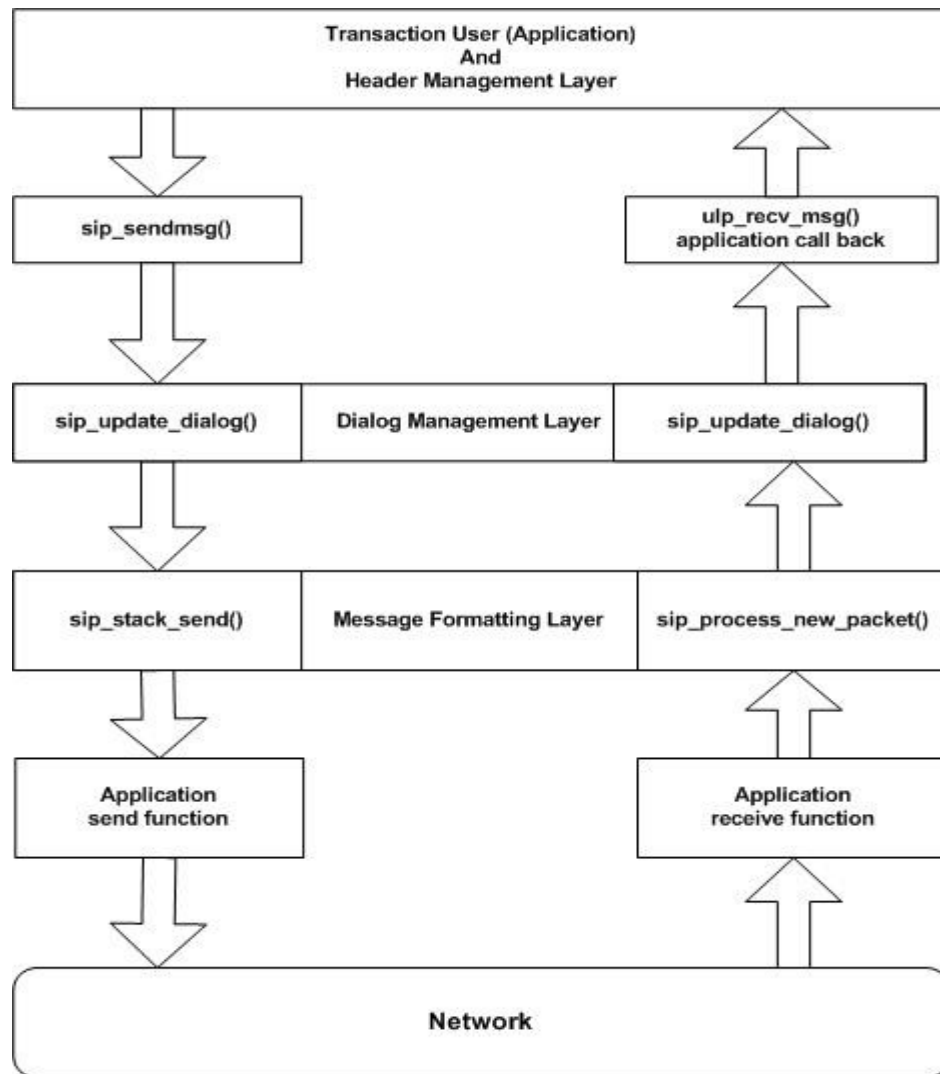


Figure 4. Call stateful: When the stack maintains dialogs and a request/response is sent without setting the SIP_SEND_STATEFUL flag.



Appendix II. External interfaces.

Stack Initialization

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

Message allocation

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

Adding SIP headers

```
int sip_add_header(sip_msg_t sip_msg, char *header_string);  
sip_header_t sip_add_param(sip_header_t sip_header, char *param,  
int *error);  
int sip_add_from(sip_msg_t sip_msg, char *display_name, char *from_uri,  
char *fromtags, boolean_t add_aquot);  
int sip_add_to(sip_msg_t sip_msg, char *display_name, char *to_uri,  
char *totags, boolean_t add_aquot);  
int sip_add_via(sip_msg_t sip_msg, char *sent_protocol_name,  
char *sent_protocol_version, char *sent_protocol_transport,  
char *sent_by_host, int sent_by_port, char *via_params);  
int sip_add_branchid_to_via(sip_msg_t sip_msg, char *branchid);  
int sip_add_maxforward(sip_msg_t sip_msg, uint_t maxforward);  
int sip_add_callid(sip_msg_t sip_msg, char *callid);  
int sip_add_cseq(sip_msg_t sip_msg, sip_method_t method, uint32_t cseq);  
int sip_add_content_type(sip_msg_t sip_msg, char *, char *);  
int sip_add_content(sip_msg_t sip_msg, char *contents);  
int sip_add_contact(sip_msg_t sip_msg, char *display_name, char  
*contact_uri, char *tag, boolean_t add_aquot);  
int sip_add_accept(sip_msg_t sip_msg, char *type, char *subtype, char  
*mparams, char *params);  
int sip_add_accept_enc(sip_msg_t sip_msg, char *code, char *params);  
int sip_add_accept_lang(sip_msg_t sip_msg, char *lang, char *params);  
int sip_add_alert_info(sip_msg_t sip_msg, char *alert, char *params);  
int sip_add_allow(sip_msg_t sip_msg, char *method_name);  
int sip_add_call_info(sip_msg_t sip_msg, char *uri, char *params);  
int sip_add_content_disp(sip_msg_t sip_msg, char *dis_type, char *params);  
int sip_add_content_enc(sip_msg_t sip_msg, char *code);  
int sip_add_content_lang(sip_msg_t sip_msg, char *lang);  
int sip_add_date(sip_msg_t sip_msg, char *date);  
int sip_add_error_info(sip_msg_t sip_msg, char *uri, char *params);  
int sip_add_expires(sip_msg_t sip_msg, int secs);  
int sip_add_in_reply_to(sip_msg_t sip_msg, char *reply_id);  
int sip_add_mime_version(sip_msg_t sip_msg, char *version);  
int sip_add_min_expires(sip_msg_t sip_msg, int secs);  
int sip_add_org(sip_msg_t sip_msg char *org);  
int sip_add_priority(sip_msg_t sip_msg, char *prio);  
int sip_add_reply_to(sip_msg_t sip_msg, char *uname, char *addr, char  
*params);  
int sip_add_require(sip_msg_t sip_msg, char *req);
```

```
int sip_add_retry_after(sip_msg_t sip_msg, int secs, char *cmt, char *params);
int sip_add_server(sip_msg_t sip_msg, char *svr_val);
int sip_add_subject(sip_msg_t sip_msg, char *subject);
int sip_add_supported(sip_msg_t sip_msg, char *support);
int sip_add_tstamp(sip_msg_t sip_msg, char *time);
int sip_add_unsupported(sip_msg_t sip_msg, char *unsupported);
int sip_add_user_agent(sip_msg_t sip_msg, char *srv_val);
int sip_add_warning(sip_msg_t sip_msg, int code, char *addr, char *msg);
```

SIP Request/Response creation

```
int sip_add_response_line(sip_msg_t sip_response, int response,
    char *response_code);
int sip_add_request_line(sip_msg_t sip_request, sip_method_t method,
    char *request_uri);
sip_msg_t sip_create_response(sip_msg_t, int, char *, char *,
    char *);
sip_msg_t sip_create_dialog_req(sip_method_t method, sip_dialog_t dialog,
    char *protocol, char *version, char *transport, char *via_param,
    char *nexthop, uint32_t maxforward, uint_t cseq);
```

Copying headers/messages

```
int sip_copy_start_line(sip_msg_t from, sip_msg_t to);
int sip_copy_header(sip_msg_t sip_msg, sip_header_t sip_header,
    char *param);
int sip_copy_header_by_name(sip_msg_t from, sip_msg_t to, char
    *header_name, char *param);
(header_name can be either the long or compact name)
int sip_copy_all_headers(sip_msg_t from, sip_msg_t to);
sip_msg_t sip_clone_msg(const sip_msg_t sip_msg);
```

Deleting SIP headers/values

```
void sip_delete_start_line(sip_msg_t sip_msg);
int sip_delete_header(sip_header_t sip_header);
int sip_delete_header_by_name(sip_msg_t sip_msg, char *header_name);
int sip_delete_value(sip_header_t header, sip_header_value_t value);
```

SIP Headers lookups

```
const sip_header_t sip_get_header(sip_msg_t sip_msg, char *header_name,
    sip_header_t old_header);
(header_name can be either the long or compact name)
```

Getting header/param values and response description

```
const struct sip_value *sip_get_header_value(const struct sip_header *
    sip_header, int *error);
const struct sip_value *sip_get_next_value(sip_header_value_t
    old_value, int *error);
const sip_str_t *sip_get_param_value(sip_header_value_t header_value,
    char *param_name);
const sip_param_t *sip_get_params(sip_header_value_t header_value);
```

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

SIP ID generation

```
char *sip_branchid(sip_msg_t sip_msg);  
char *sip_guid();  
uint32_t sip_get_cseq();
```

VIA functions

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

Sending SIP messages

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

Transaction layer functions

```
const struct sip_xaction *sip_get_trans(sip_msg_t sip_msg, int which);  
sip_method_t sip_get_trans_method(sip_transaction_t sip_trans);  
int sip_get_trans_state(sip_transaction_t sip_trans);  
const struct sip_message *sip_get_trans_orig_msg(sip_transaction_t  
sip_trans);  
const struct sip_conn_object *sip_get_trans_conn_obj(sip_transaction_t  
sip_trans);  
void sip_hold_trans(sip_transaction_t sip_trans);  
void sip_release_trans(sip_transaction_t sip_trans);
```

Dialog layer functions

```
int sip_get_dialog_state(sip_dialog_t dialog);  
const sip_str_t *sip_get_dialog_callid(sip_dialog_t dialog);  
const sip_str_t *sip_get_dialog_local_tag(sip_dialog_t dialog);  
const sip_str_t *sip_get_dialog_remote_tag(sip_dialog_t dialog);  
const struct sip_uri *sip_get_dialog_local_uri(sip_dialog_t dialog);  
const struct sip_uri *sip_get_dialog_remote_uri(sip_dialog_t dialog);  
const struct sip_uri *sip_get_dialog_remote_target_uri(sip_dialog_t dialog);  
const sip_str_t *sip_get_dialog_route_set(sip_dialog_t dialog);  
uint32_t sip_get_dialog_local_cseq(sip_dialog_t dialog);  
uint32_t sip_get_dialog_remote_cseq(sip_dialog_t dialog);  
int sip_get_dialog_type(sip_dialog_t dialog);  
void sip_hold_dialog(sip_dialog_t dialog);  
void sip_release_dialog(sip_dialog_t dialog);  
void sip_delete_dialog(sip_dialog_t dialog);
```

URI functions

```
const struct sip_uri *sip_get_uri_parsed(sip_header_value_t value, int *error);  
const struct sip_uri *sip_get_request_uri(sip_msg_t sip_msg, int *error);  
const struct sip_uri *sip_parse_uri(sip_str_t *sip_uri, int *error);  
boolean_t sip_is_sipuri(sip_uri_t sip_uri);  
const sip_str_t *sip_uri_scheme(const struct sip_uri *uri, int *error);  
const sip_str_t *sip_get_uri_user(const struct sip_uri *uri, int *error);
```

```
const sip_str_t *sip_get_uri_password(const struct sip_uri *uri, int *error);
const sip_str_t *sip_get_uri_host(const struct sip_uri *uri, int *error);
uint_t sip_get_uri_port(sip_uri_t sip_uri, int *error);
const sip_param_t *sip_get_sip_uri_params(sip_uri_t sip_uri, int *error);
const sip_str_t *sip_get_sip_uri_headers(sip_uri_t sip_uri, int *error);
const sip_str_t *sip_get_abs_uri_opaque(sip_uri_t sip_uri, int *error);
const sip_str_t *sip_get_abs_uri_query(sip_uri_t sip_uri, int *error);
const sip_str_t *sip_get_abs_uri_path(sip_uri_t sip_uri, int *error);
const sip_str_t *sip_get_abs_uri_regname(sip_uri_t sip_uri, int *error);
boolean_t sip_uri_is_tel_user(sip_uri_t sip_uri);
char *sip_get_uri_error(uint_t errflags);
```

Getting SIP header values

```
boolean_t sip_msg_is_request(const sip_msg_t sip_msg);
boolean_t sip_msg_is_response(const sip_msg_t sip_msg);
sip_method_t sip_get_request_method(const sip_msg_t sip_msg);
int sip_get_response_code(sip_msg_t sip_msg);
const sip_str_t *sip_get_response_phrase(sip_msg_t sip_msg);
const sip_str_t *sip_get_sip_version(sip_msg_t sip_msg);
int sip_get_msg_len(sip_msg_t sip_msg);
const sip_str_t *sip_get_contact_display_name(sip_header_value_t value);
const sip_str_t *sip_get_from_display_name(sip_msg_t sip_msg, int *error);
const sip_str_t *sip_get_from_tag(sip_msg_t sip_msg, int *error);
const sip_str_t *sip_get_to_display_name(sip_msg_t sip_msg, int *error);
const sip_str_t *sip_get_to_tag(sip_msg_t sip_msg, int *error);
const sip_str_t *sip_get_callid(sip_msg_t sip_msg, int *error);
int sip_get_callseq_num(sip_msg_t sip_msg, int *error);
sip_method_t sip_get_callseq_method(sip_msg_t sip_msg, int *error);
const sip_str_t *sip_get_via_sent_by_host(sip_header_value_t value);
int sip_get_via_sent_by_port(sip_header_value_t value);
const sip_str_t *sip_get_via_sent_protocol_version(sip_header_value_t value);
const sip_str_t *sip_get_via_sent_protocol_name(sip_header_value_t value);
const sip_str_t *sip_get_via_sent_transport(sip_header_value_t value);
int sip_get_maxforward(sip_msg_t sip_msg, int *error);
int sip_get_content_length(sip_msg_t sip_msg, int *error);
const sip_str_t *sip_get_content_type(sip_msg_t sip_msg, int *error);
const sip_str_t *sip_get_content_sub_type(sip_msg_t sip_msg, int *error);
char *sip_get_content(sip_msg_t sip_msg, int *error);
const sip_str_t *sip_get_accept_type(sip_header_value_t value, int *error);
const sip_str_t *sip_get_accept_sub_type(sip_header_value_t value, int *error);
const sip_str_t *sip_get_accept_enc(sip_header_value_t value, int *error);
const sip_str_t *sip_get_accept_lang(sip_header_value_t value, int *error);
const sip_str_t *sip_get_alert_info_uri(sip_header_value_t value, int *error);
const sip_str_t *sip_get_allow_method(sip_header_value_t value, int *error);
int sip_get_min_expires(sip_msg_t sip_msg, int *error);
const sip_str_t *sip_get_mime_version(sip_msg_t sip_msg, int *error);
const sip_str_t *sip_get_org(sip_msg_t sip_msg, int *error);
const sip_str_t *sip_get_priority(sip_msg_t sip_msg, int *error);
const sip_str_t *sip_get_replyto_display_name(sip_header_value_t value, int *error);
const sip_str_t *sip_get_replyto_uri(sip_header_value_t value, int *error);
```

```

const sip_str_t *sip_get_date_time(sip_msg_t sip_msg, int *error);
int sip_get_date_day(sip_msg_t sip_msg, int *error);
const sip_str_t *sip_get_date_month(sip_msg_t sip_msg, int *error);
const sip_str_t *sip_get_date_wkday(sip_msg_t sip_msg, int *error);
int sip_get_date_year(sip_msg_t sip_msg, int *error);
const sip_str_t *sip_get_date_timezone(sip_msg_t sip_msg, int *error);
const sip_str_t *sip_get_content_disp(sip_msg_t sip_msg, int *error);
const sip_str_t *sip_get_content_enc(sip_header_value_t value, int *error);
const sip_str_t *sip_get_error_info_uri(sip_header_value_t value, int *error);
int sip_get_expires(sip_msg_t sip_msg, int *error);
const sip_str_t *sip_get_require(sip_header_value_t value, int *error);
const sip_str_t *sip_get_subject(sip_msg_t sip_msg, int *error);
const sip_str_t *sip_get_supported(sip_header_value_t value, int *error);
const sip_str_t *sip_get_tstamp_delay(sip_msg_t, int *);
const sip_str_t *sip_get_unsupported(sip_header_value_t value, int *error);
const sip_str_t *sip_get_server(sip_msg_t sip_msg, int *error);
const sip_str_t *sip_get_user_agent(sip_msg_t sip_msg, int *error);
int sip_get_warning_code(sip_header_value_t value, int *error);
const sip_str_t *sip_get_warning_agent(sip_header_value_t value, int *error);
const sip_str_t *sip_get_warning_text(sip_header_value_t value, int *error);
const sip_str_t *sip_get_call_info_uri(sip_header_value_t value, int *error);
const sip_str_t *sip_get_in_reply_to(sip_header_value_t value, int *error);
int sip_get_retry_after_time(sip_msg_t sip_msg, int *error);
const sip_str_t *sip_get_retry_after_cmts(sip_msg_t sip_msg, int *error);

```

Connection object functions

```

int sip_conn_send(const sip_conn_object_t conn_obj, char *msg,
int msglen);
void sip_hold_conn_object(sip_conn_object_t conn_obj);
void sip_rel_conn_object(sip_conn_object_t conn_obj);
boolean_t sip_conn_is_reliable(sip_conn_object_t conn_obj);
boolean_t sip_conn_is_stream(sip_conn_object_t conn_obj);
int sip_conn_remote_address(sip_conn_object_t conn_obj,
struct sockaddr *name, socklen_t *namelen);
int sip_conn_local_address(sip_conn_object_t conn_obj,
struct sockaddr *name, socklen_t *namelen);
int sip_conn_transport(sip_conn_object_t conn_obj);

```

Miscellaneous functions

```

char *sip_msg_to_str(sip_msg_t sip_msg, int *error);
char *sip_hdr_to_str(sip_header_t sip_header, int *error);
char *sip_reqline_to_str(sip_msg_t sip_msg, int *error);
char *sip_respline_to_str(sip_msg_t sip_msg, int *error);
int sip_init_conn_object(sip_conn_object_t conn_obj);
void sip_clear_stale_data(sip_conn_object_t conn_obj);
void sip_conn_destroyed(sip_conn_object_t conn_obj);

```

Appendix III. External data structures

SIP message

```
typedef struct sip_message    *sip_msg_t;
```

SIP header

```
typedef struct sip_header    *sip_header_t;

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

Value of a SIP header

```
typedef struct sip_value    *sip_header_value_t;
```

SIP transactions

```
typedef struct sip_xaction    *sip_transaction_t;
```

SIP dialog

```
typedef struct sip_dialog    *sip_dialog_t;
```

SIP URI

```
typedef struct sip_uri    *sip_uri_t;
```

SIP string

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

SIP param

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

SIP methods

```
typedef enum {
    UNKNOWN = 0,
    INVITE,
    ACK,
    OPTIONS,
    BYE,
    CANCEL,
    REGISTER,
    REFER,
    INFO
} sip_method_t;
```

SIP initialization structure

```
typedef struct sip_stack_init_s {
    int sip_version;
    uint32_t sip_stack_flags;
    int sip_timer_T1;
    int sip_timer_T2;
    int sip_timer_T4;
    int sip_timer_TD;
    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;
```

```
#define SIP_STACK_VERSION_1 1
```

```
typedef struct sip_io_pointers_s {
    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_stream)(sip_conn_object_t);
    boolean_t (*sip_conn_is_reliable)(sip_conn_object_t);
    int (*sip_conn_remote_address)(sip_conn_object_t,
        struct sockaddr *, socklen_t *);
    int (*sip_conn_local_address)(sip_conn_object_t,
        struct sockaddr *, socklen_t *);
    int (*sip_conn_transport)(sip_conn_object_t);
} sip_io_pointers_t;
```

```
typedef struct sip_ulp_pointers_s {
    void (*sip_ulp_rcv)(const sip_conn_object_t,
        sip_msg_t, const sip_dialog_t);
    uint_t (*sip_ulp_timeout)(void *, void (*func)(void *),
        struct timeval *);
    boolean_t (*sip_ulp_untimeout)(uint_t);
    int (*sip_ulp_trans_error)(sip_transaction_t, int, void *);
    void (*sip_ulp_dlg_del)(sip_dialog_t, sip_msg_t, void *);
} sip_ulp_pointers_t;
```

SIP header function table

```
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;
```

SIP parsed header

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

```
#define SIP_PARSED_HEADER_VERSION_1 1
```

SIP header value

```
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;
```

```
#define SIP_VALUE_VERSION_1 1
```

SIP flags

SIP_SEND_STATEFUL flag to sip_sendmsg() to send a request or response statefully.

SIP_STACK_DIALOGS flag for sip_stack_flags when initializing the stack to indicate that the library must maintain dialogs.

Error values returned by URI access functions

<i>#define URLERR_SCHEME</i>	<i>0x00000001</i> /* invalid URL SCHEME name */
<i>#define URLERR_USER</i>	<i>0x00000002</i> /* invalid user name */
<i>#define URLERR_PASS</i>	<i>0x00000004</i> /* invalid password */
<i>#define URLERR_HOST</i>	<i>0x00000008</i> /* invalid domain name */
<i>#define URLERR_PORT</i>	<i>0x00000010</i> /* invalid port number */
<i>#define URLERR_PARAM</i>	<i>0x00000020</i> /* parameter specific error */
<i>#define URLERR_HEADER</i>	<i>0x00000040</i> /* headers specific error */
<i>#define URLERR_OPAQUE</i>	<i>0x00000080</i> /* opaque specific error */
<i>#define URLERR_QUERY</i>	<i>0x00000100</i> /* query specific error */
<i>#define URLERR_PATH</i>	<i>0x00000200</i> /* path specific error */
<i>#define URLERR_REGNAME</i>	<i>0x00000400</i> /* reg-name specific error */
<i>#define URLERR_NOURI</i>	<i>0x00000800</i> /* No URI */

Appendix IV. Examples

Note, in the following examples we don't always check for the return values.

Stack Initialization

```
sip_stack_init_t    sip_init[1];
sip_io_pointers_t  sip_io[1];
sip_ulp_pointers_t sip_ulp;

/* Initialize connection manager – all my_conn* functions should be defined */
bzero(sip_init, sizeof(sip_stack_init_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;
sip_io->sip_conn_transport = my_conn_transport;

/* Initialize callback registrations – my_ulp_rcv and ulp_dialog_del should be defined */
bzero(&sip_ulp, sizeof(sip_ulp_pointers_t));
sip_ulp.sip_ulp_rcv = my_ulp_rcv;
sip_ulp.sip_ulp_dlg_del = ulp_dialog_del;

sip_init->sip_version = SIP_STACK_VERSION_1;
sip_init->sip_io_pointers = sip_io;
/* library will manage dialogs */
sip_init->sip_stack_flags |= SIP_STACK_DIALOGS;
sip_init->sip_ulp_pointers = &sip_ulp;

if (sip_stack_init(sip_init) != 0) {
    /* error */
}
```

Creating and sending requests on a connection – conn_object.

```
sip_msg_t    sip_msg;
int          port = 5060;

sip_msg = sip_new_msg();
if (sip_msg == NULL) {
    /* error */
}
sip_add_request_line(sip_msg, INVITE, "sip:user@example.com");
sip_add_from(sip_msg, "me", "sip:me@mydomain.com", "tag-from-01", B_TRUE);
sip_add_to(sip_msg, "you", "sip:you@yourdomain.com", NULL, B_TRUE);
sip_add_contact(sip_msg, NULL, "sip:myhome.host.com", NULL, B_TRUE);
sip_add_via(sip_msg, "SIP", "2.0", "UDP", "192.0.0.1", port, "branch=z9hG4bK-via-01");
sip_add_maxforward(sip_msg, 70);
```

```

sip_add_callid(sip_msg, NULL);
sip_add_cseq(sip_msg, INVITE, sip_get_cseq());

if (sip_sendmsg(conn_object, sip_msg, NULL, SIP_SEND_STATEFUL) != 0) {
    /* error */
}

```

Creating and sending in-dialog requests on conn_object, dialog is given by sip_dialog.

```

sip_msg_t    sip_msg;

/* -1 as the last arg indicates that the cseq # will be filled in by sip_create_dialog_req() */
sip_msg = sip_create_dialog_req (BYE, sip_dialog, "SIP", "2.0", "UDP",
    ";branch=x9hG4bK-via-02", "192.0.0.2", 70, -1);
if (sip_msg == NULL) {
    /* error */
}
if (sip_sendmsg(conn_object, sip_msg, sip_dialog, SIP_SEND_STATEFUL) != 0) {
    /* error */
}

```

Creating and sending responses for request sip_request on connection conn_object

```

int          resp_code = SIP_OK;
char         *ttag;
sip_msg_t    sip_msg_resp;

/* generate to tag */
ttag = sip_guid();

/* create response specifying contact info as "me@192.0.2.4" */
sip_msg_resp = sip_create_response(sip_request, resp_code, sip_get_resp_desc(resp_code),
    ttag, "me@192.0.2.4");
free(ttag);
if (sip_msg_resp == NULL) {
    /* error */
}
if (sip_sendmsg(conn_object, sip_msg_resp, NULL, SIP_SEND_STATEFUL) != 0) {
    /* error */
}

```

Manipulating headers

```

/* find FROM header and copy to another message */
sip_copy_header_by_name(sip_msg, new_msg, "FROM", NULL);

/* delete contact header */
sip_delete_header_by_name(sip_msg, "CONTACT");

```

```

/* get topmost VIA header */
sip_header_t  via_hdr;

via_hdr = sip_get_header(sip_msg, "VIA", NULL);

/* get all VIA headers */
via_hdr = sip_get_header(sip_msg, "VIA", NULL);
while (via_hdr != NULL)
    via_hdr = sip_get_header(sip_msg, "VIA", via_hdr);

/* print all headers to stdout */
sip_header_t  header;
char          *hdr_str;

header = sip_get_header(sip_msg, NULL, NULL);
while (header != NULL) {
    hdr_str = hdr_to_str(header, NULL);
    printf("%s\n", hdr_str);
    free(hdr_str);
    header = sip_get_header(sip_msg, NULL, header);
}

/* get value from a header */
sip_header_t      sip_header;
sip_header_value_t via_value;
int               error;
sip_str_t         *via_sent_by;
const sip_param_t *params;
const sip_str_t   *branchid;

sip_header = sip_get_header(sip_msg, "VIA", NULL);
if (sip_header == NULL) {
    /* error */
}
if ((via_value = sip_get_header_value(header, &error)) == NULL) {
    /* error */
}

/* get param list */
params = sip_get_params(via_value);

/* get value of a specific param */
branchid = sip_get_param_value(via_value, "branch");

```

Getting URI components

```

sip_msg_t          sip_msg = NULL;
sip_uri_t          sip_uri = NULL;
const sip_str_t    *sip_str = NULL;
const sip_header_t sip_header = NULL;
const sip_header_value_t sip_value = NULL;
const sip_param_t  *sip_param = NULL;
uint_t            port = 0;
int               error = 0;

sip_msg = sip_new_msg();
if (sip_msg == NULL) {
    /* error */
}
sip_add_from(sip_msg, "me", "sip:alice:secret@atlanta.com:123;actor=jude"
    ";lr?movie=titanic&siph=", "From-tag-01", B_TRUE);
header = sip_get_header(sip_msg, "FROM", NULL);
sip_value = sip_get_header_value(sip_header, &error);
sip_uri = sip_get_uri_parsed(sip_value, &error);

/* error handling should be a separate function, it is shown here for clarity */
if (error != 0) {
    /* bad URI */
    if (error & URLERR_SCHEME) {
        /* scheme error */
    }
    if (error & URLERR_USER) {
        /* user error */
    }
    if (error & URLERR_PASS) {
        /* password error */
    }
    if (error & URLERR_HOST) {
        /* host error */
    }
    if (error & URLERR_PORT) {
        /* port error */
    }
    if (error & URLERR_PARAM) {
        /* param error */
    }
    if (error & URLERR_HEADER) {
        /* header error */
    }
}

/* check if it is a SIP URI */
if (!sip_is_sipuri(sip_uri)) {
    /* error */
}

/* get user name, password, host, port, params and headers */
sip_str = sip_uri_scheme(sip_uri, &error);
```

```
sip_str = sip_get_uri_user(sip_uri, &error);
sip_str = sip_get_uri_password(sip_uri, &error);
sip_str = sip_get_uri_host(sip_uri, &error);
port = sip_get_uri_port(sip_uri, &error);
sip_param = sip_get_sip_uri_params(sip_uri, &error);
sip_str = sip_get_sip_uri_headers(sip_uri, &error);
sip_free_msg(sip_msg);
```

Check header for compliance

```
sip_header_t via_hdr;

/* get topmost VIA header */
via_hdr = sip_get_header (sip_msg, "VIA", NULL);
if (via_hdr == NULL) {
    /* error */
}

/* check for compliance */
if (!sip_header_is_compliant(via_hdr)) {
    /* not compliant */
}
```

Appendix V. SIP Transaction timers

<i>Timer</i>	<i>Default Value</i>	<i>Section in RFC 3261</i>	<i>Meaning</i>
T1	500 msec.	17.1.1.1	RTT estimate
T2	4 sec.	17.1.2.2	Max. retransmit interval for non-INVITE requests and INVITE responses
T4	5 sec.	17.1.2.2	Max. duration a message will remain in the network
Timer A	T1	17.1.1.2	INVITE request retransmit interval, for UDP only.
Timer B	64 * T1	17.1.1.2	INVITE transaction timeout timer.
Timer C	> 3 min	16.6 (bullet 11)	Proxy INVITE transaction timeout
Timer D	> 32 sec 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, 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.
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.