SIP API for Solaris

Revision 1.3 - June, 2006

Network Approachability
Sun Microsystems Inc.
network-sip-discuss@opensolaris.org

Table of Contents

List of Figures

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 easy to use and extensible set of API to enable development of SIP applications. The underlying SIP stack should be modular and flexible enough for a user to provide his/her own functionality, if needed.

The API and stack are implemented as a passive user library that requires applications to register certain mandatory and optional functions. The mandatory functions are provided by the application for performing I/O and receiving incoming messages from the stack. Optional functions include providing timeout/un-timeout functionality, callbacks for error notification, and transaction/dialog state transitions among others. 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 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 is initiated by a request sent by a client transaction to a server transaction, and terminated by a final response (for that request) from the server transaction, there could be intermediate responses from the server. The Transaction Management layer handles retransmissions, matching responses to requests, and 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. The stack can be configured to manage dialogs, in which case the stack automatically creates and maintains dialogs for incoming and outgoing SIP messages. Additionally, the stack also delivers the matching dialog along with the incoming message.

Message Formatting Layer

For an incoming message, the Message Formatting layer constructs a SIP message and delivers it to the application. If the underlying transport is TCP, this layer also breaks the byte stream at SIP message boundaries using the Content-Length header. If there is a matching transaction for the incoming message, the Transaction Management Layer processes the message before passing it on to the Dialog Management layer, if required. The SIP message is subsequently delivered to the application.

On the outbound side, the stack 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 and Dialog Management layers, if required. Finally, the stack constructs the outgoing message by copying all the headers from the SIP message into a contiguous buffer and invoking the application registered send function.

Timer Subsystem

The SIP stack uses several timers. The Timer subsystem 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 stack. If he application is single threaded, it should register its own timer routines. These routines are internal to the SIP 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 stack uses this first member to store connection object specific data for its use. For this reason applications must initialize a connection object using a library provided function.

Further, an 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 stack 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 below:

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.

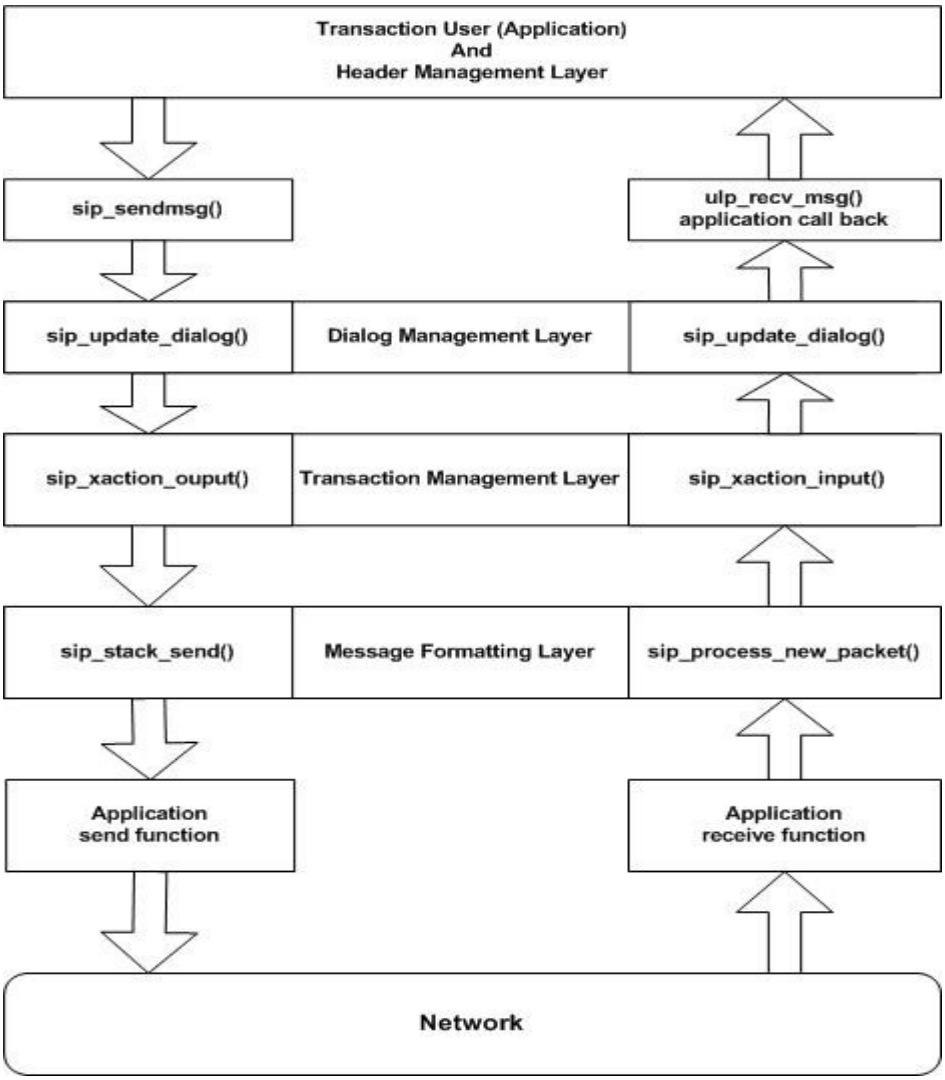


Figure 2. Stateless: When the stack does not maintain dialogs and a request/response is sent without setting the SIP_SEND_STATEFUL flag:

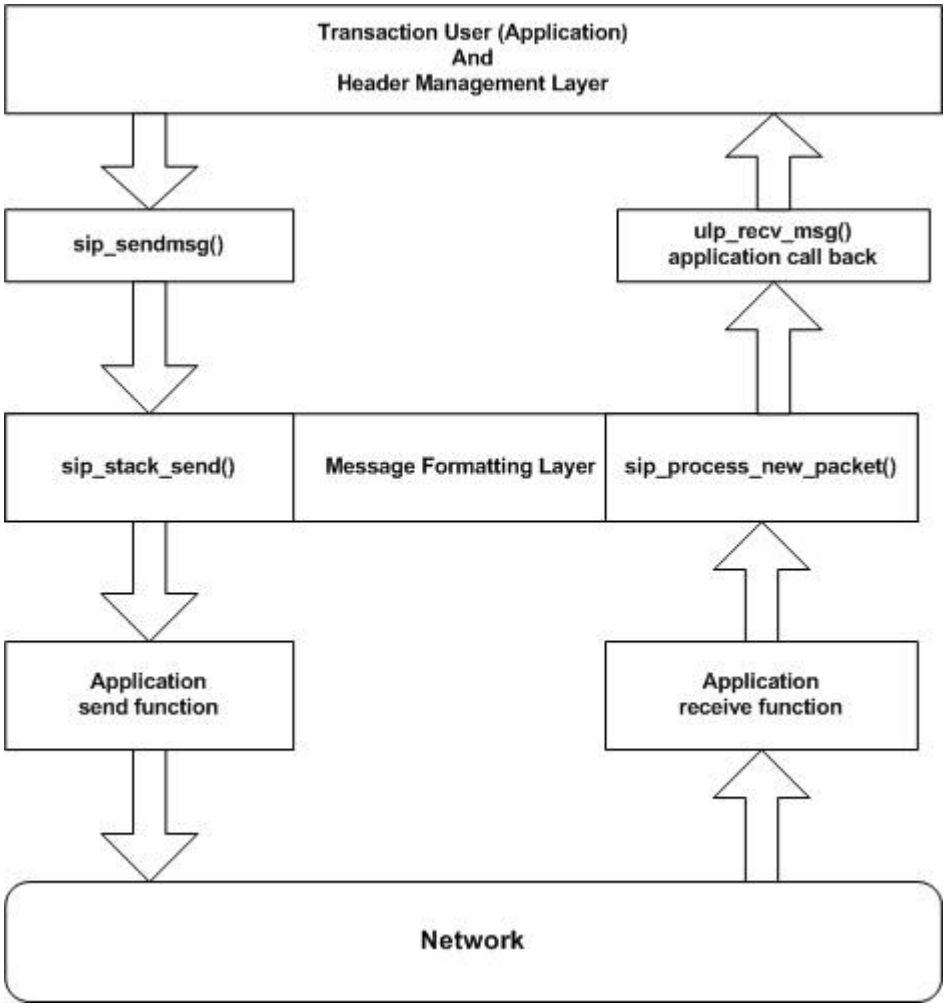


Figure 3. Transaction stateful: When the stack does not maintain dialogs and a request/response is sent with the SIP_SEND_STATEFUL flags set.

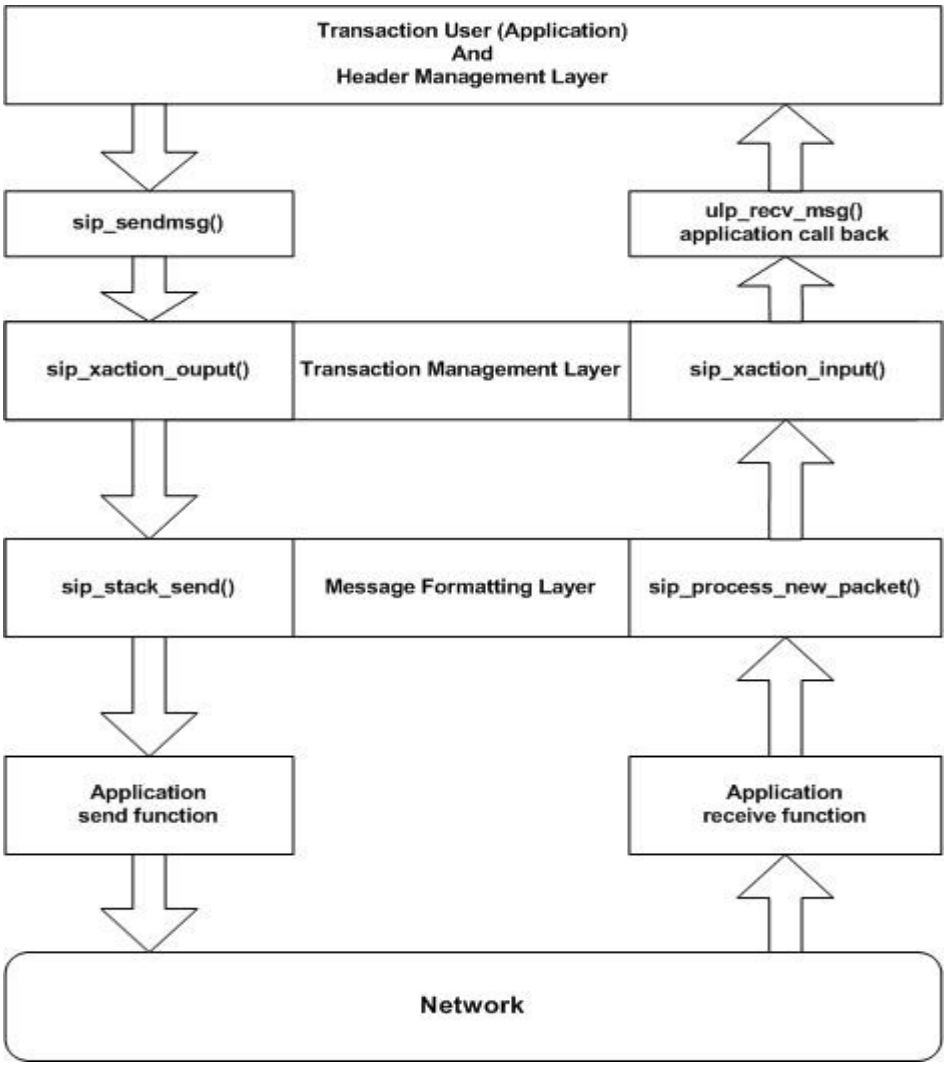
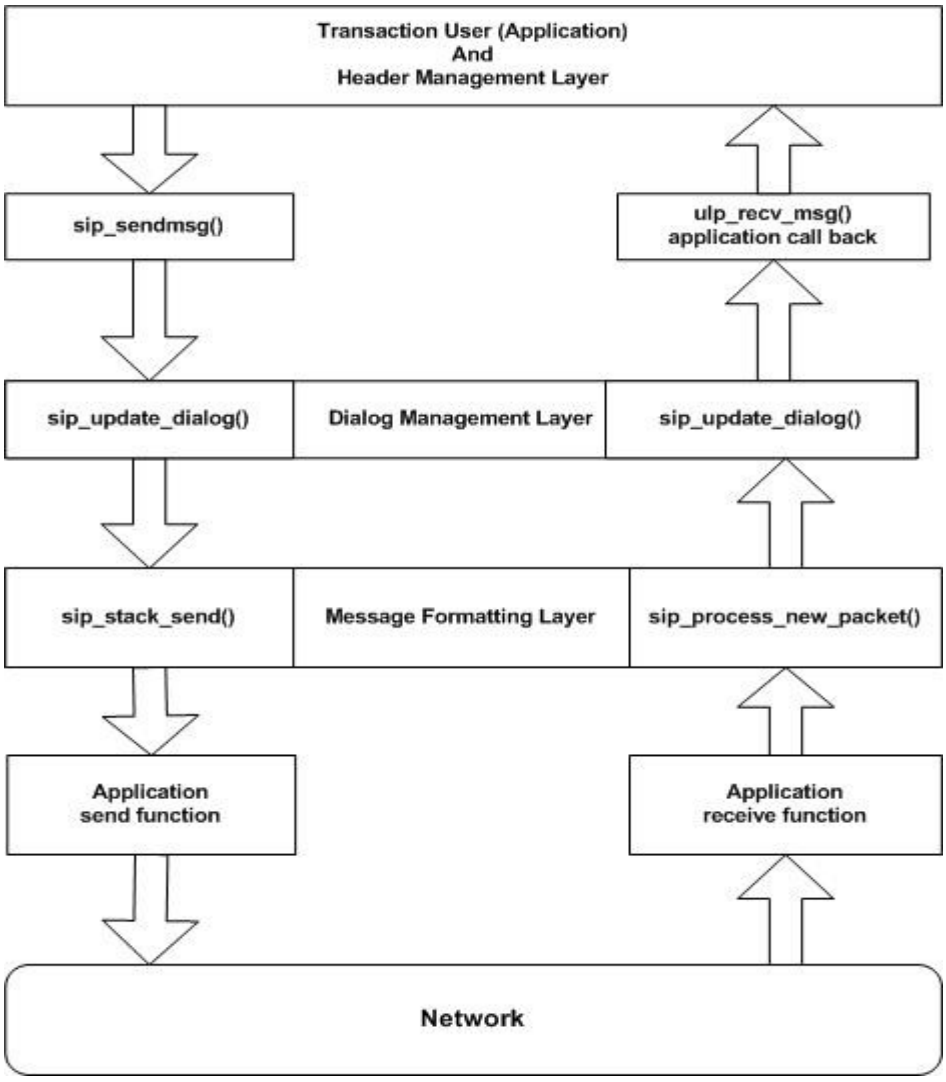


Figure 4. Call stateful: When the stack maintains dialogs and a request/response is sent without setting the SIP_SEND_STATEFUL flag.



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.

Logging/Tracing:

The library currently does not include a logging/tracing mechanism.

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 I** and **Appendix II** 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 (I/O) interfaces
- Custom header table.

Generic stack parameters:

SIP version

*S*et to *SIP_STACK_VERSION***.**

Stack flags

If set to *SIP_STACK_DIALOGS,* instructs the library to maintain dialogs; if this flag is not set, the library does not maintain any dialog information.

Upper layer registrations

Upper layer receive routine

The stack delivers SIP messages to the application using this 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. If the application does not provide these, the library will use the built-in timeout and un-timeout routines.

Transactions error notifications

The application can optionally register a routine that the stack 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 in case of a network error.

Dialog delete notification

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

Transaction state transition notifications

An application can register a routine that will be invoked when a transaction changes states. The routine will be invoked with the transaction handle, the message resulting in the transition and the previous and current transaction states. If an application does not register this callback, there will be no notifications when a transaction changes states.

Dialog state transition notifications

If the stack is configured to maintain dialogs, the application can optionally provide a callback routine, which the stack will invoke when a dialog changes states. The callback will be invoked with the dialog handle, the message resulting in the transition and the previous and current state. If an application does not register this callback, there will be no notifications when a dialog changes states.

Connection Manager Interfaces

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

Send routine

The stack 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 transport byte-stream (e.g. TCP).
is_reliable : is transport reliable (e.g. TCP, SCTP)
remote address: peer endpoint information.
local address : local endpoint information.
transport : type of transport (TCP/UDP/SCTP)

Optionally, the application can register the following for the stack to obtain timer values for a connection object:

Timer1 : RTT (Round trip time) estimate.
Timer2 : Maximum retransmit interval for non-INVITE requests and INVITE responses.
Timer4 : Max. duration a message will remain in the network.
TimerD : Wait time for response retransmits.

If the application does not register these, default values for the timers are used (see **APPENDIX IV**).

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 II**.

After initializing the stack, an application can create requests and send them out using the interfaces provided by the Header Management and Message Formatting layers. Likewise, an application can receive incoming request/response after initializing the stack and pass them to the stack for processing.

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 a 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; e.g. for the VIA

header, each *sip_via_value_t* structure would correspond to a *via-parm*.

When an incoming message is received by the stack from the application, 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.

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;
```

sip_header_general_t provides the starting (*sip_hdr_start*) and ending (*sip_start_end*) positions for the header, *sip_hdr_current* is used when processing the header (i.e. to indicate the current position being processed). *sip_hdr_allocated* states if the header was allocated and, hence, 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.

```
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 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 registers a table 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. *header_check_compliance* and *header_is_equal* are currently not supported in the default function table.

When a header is deleted by the application, *sip_header_state* is set to *SIP_HEADER_DELETE*. 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;
```

value is a pointer to a *sip_<header_name>_value_t*, which is returned as a result of parsing a SIP header. 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 defined as:

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

where sip_str_t is defined as:

```
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 *sip_<header_name>_value_t* 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* and *value_state* to *SIP_VALUE_DELETED*. Existing references to the deleted value remain, while new lookups ignore the deleted value.

The hierarchy to obtain a value from the SIP message is:

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 determined to be invalid, *value_state* is set to *SIP_VALUE_BAD* .

Writing Parsers for Custom Headers

An application can provide custom headers along with 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 the parser allocates.

sip_parsed_header_t

The parser allocates this structure and assigns it as a result of parsing the header.

sip_value_t

The application defines the value for an 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 functions for accessing members of value.

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 stack 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 hash function will salted to take avoid bucket-collison. The branch ID in the topmost VIA header is used to identify whether a message complies with RFC 3261 or not (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;
    sip_timer_t         sip_xaction_TB;
    sip_timer_t         sip_xaction_TD;
    sip_timer_t         sip_xaction_TE;
    sip_timer_t         sip_xaction_TF;
    sip_timer_t         sip_xaction_TG;
    sip_timer_t         sip_xaction_TH;
    sip_timer_t         sip_xaction_TI;
    sip_timer_t         sip_xaction_TJ;
    sip_timer_t         sip_xaction_TK;

    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 and state transition callbacks, if provided by the application.

Transaction creation and maintenance

For incoming requests and responses the transaction hash table is scanned for an existing transaction. For an incoming response if a transaction does not exist the response is handled statelessly; 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 an incoming request matches a transaction, it is considered a retransmission and the last response is retransmitted by the Transaction layer, the request is 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 an incoming response matches a transaction, the response is processed statefully and the message is passed to the application.

For outgoing responses and requests 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). 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 (timer names from RFC 3261) are initialized (see **Appendix IV**).

When a message causes a transaction to change states, a callback function, if registered, is invoked.

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.

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 further lookups. When the last reference to the terminated transaction is released (i.e the reference count drops to 0), it will be removed from the hash table.

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 header fields are used for messages conforming to RFC 2543. A successful lookup always increments the reference count for the transaction, the caller is responsible for releasing the reference.

Transaction Timers

The Transaction layer maintains a set of timers for timing out transactions or sending retransmissions. These timers are listed in **Appendix IV**. When a transaction times-out, the Transaction layer notifies the application if it has registered a callback for this purpose.

Transaction and Network errors

When the Transaction layer sends a message, either retransmitting a request or a response, it uses a cached connection object (discussed in *Connection Manager* sub-section). 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 is optional; 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 a corresponding NOTIFY request creates a dialog. Dialogs created due to provisional responses are EARLY dialogs, while those created by a 2xx final response are CONFIRMED dialogs. An EARLY dialog transitions to the confirmed state when a subsequent 2xx response is received. When a dialog changes state a callback function, if registered, is invoked.

An application can do its own Dialog Management or delegate it to the stack. If configured to manage dialogs, the stack automatically creates and maintains dialogs and also delivers any dialog, matching incoming messages, to the application. If the stack is not configured to manage dialogs, there is no interaction between the application and the stack 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_header_t           *sip_dlg_event;
    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_secure;
    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 */
    boolean_t               sip_dlg_on_fork;
    sip_method_t            sip_dlg_method;
    void                    *sip_dlg_ctxt;  /* currently unused */
} _sip_dialog_t;
```

UAC dialog creation

The library creates a partial dialog when an application sends out a dialog creating request (INVITE or SUBSCRIBE). When the dialog completing response is received, the partial dialog is completed. If the request is sent by setting the *SIP_DIALOG_ON_FORK* flag in *sip_sendmsg()*, the partial dialog is retained for any further dialog completing responses (due to forking). The partial dialog times out after a time duration that is set to the INVITE transaction timeout, after which further dialogs will not be created. If *SIP_DIALOG_ON_FORK* is not set when sending the request, the first dialog completing response creates a dialog, further responses to that request will not create any dialog. After completing a dialog, the response is sent upstream along with the newly completed dialog.

UAS dialog creation

When the stack receives a dialog creating request, it creates a partial dialog using the request. This partial dialog is sent upstream along with the request. The stack 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 stack 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. 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 respectively. An application can cache a dialog after incrementing its reference count. The library provides *sip_hold_dialog()* and *sip_release_dialog()* for this purpose. The application should also register a callback function, which will be invoked 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()*.

Message Formatting Layer

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

If the incoming message arrives over TCP, the Message Formatting layer breaks the byte stream at message boundaries. It parses the message into a SIP message. The message is then passed on to the next layer.

On the sending side, the SIP message is received from the application. The Message formatting Layer adds a Content-Length header, copies all headers and content into a contiguous buffer and passes it on to the application registered send routine.

Receiving Messages

The application delivers incoming messages, along with the connection object, to the stack using *sip_process_new_packet()*. 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 stack 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 message (thus, when a connection is closed or reused the application has to inform the stack so that it can free the data and resources allocated for this purpose). After parsing the message, i.e. converting it into a SIP message, 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, along with the connection object, using *sip_sendmsg()* with message specific flags. The Message Formatting Layer adds a Content-Length and an empty line (CRLF) to the message and delivers the packet to the next layer. The Message Formatting layer also forms a contiguous buffer using the

contents of the SIP message and delivers it to the send routine provided by the application.

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 stack, 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);
```

and the following optional ones to provide connection object specific values for Timer 1, Timer 2, Timer 4 and Timer D (timer names are from RFC 3261):

```
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, 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 be a void pointer, for use by the stack. 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; /* for use by the stack */
    int                     my_conn_fd;    /* socket 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_timer1; /* in msecs */
    int                     my_conn_timer2;/* in msecs */
    int                     my_conn_timer4;/* in msecs */
    int                     my_conn_timerD;/* in msecs */
} my_conn_obj_t;
```

As part of initializing a connection object, the stack 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.

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

Freeing connection object

The application 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 application wants to free the connection object and there are references to it, it calls a library provided function *sip_conn_destroyed()* so that the stack can take appropriate action before releasing the references it holds. When *sip_conn_destroyed()* is called, the stack locates the transaction(s) caching the connection object and terminates them; any resources allocated for TCP handling are also freed at this time.

Sending Messages

When an application sends a message using *sip_sendmsg()*, the stack internally calls *sip_conn_send()* after processing the outbound message. In case of a network error, the application 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 application must not change the transport type of a connection object.

Receiving Messages

The application delivers incoming messages to the library using *sip_process_new_packet()*. After processing the message, the stack calls the application registered receive function to pass the SIP 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 sends a message, such as retransmitting a request or a response, it uses the 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 Subsystem

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()* with the timeout id of the appropriate timeout id.

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, From, Call-ID, Request URI, topmost VIA header and the sequence number from Cseq header. The returned branch-ID is 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()* or sip_get_rseq() to obtain the initial sequence number. *sip_get_cseq()* and *sip_get_rseq()* uses the most significant 31 bits of the value returned by *time(2)*.

IV. User agent crashes and restarts/recovery

In a SIP network call and session state resides in the user agents such as gateways, conferencing and media servers etc. SIP proxies do not maintain call state, as a result failure of a SIP proxy mid-call has no effect on in-progress calls.

The SIP library does not maintain any state for the media session. The SIP library, if configured to manage dialogs, maintains a list of dialogs (call-legs) active for a user agent. However, if a media session is interrupted (say peer fails), it is the user agent that re-establishes the session using the session paramerters it had used to establish the session in the first place. The user agent can terminate the dialog, if any, for a failed session using *sip_delete_dialog()*.

If a SIP transaction is interrupted (i.e the peer fails mid-way during a transaction), then the user agent on detecting a failure can re-establish another connection, if required, and calls *sip_conn_destroyed()* with the failed connection object which will result in terminating any outstanding transaction on that connection. Retransmitted request (say when a user agent fails and comes up) are handled as per the transaction state transition diagram in secction 17, RFC 3261. If SIP messages are sent statelessly, then the user agent handles these cases as appropriate. If an user agent crashes, any state that it maintains (session/call) is lost unless the user agent backs it up and there is a mechanism to failover to another user agent that can take over; this is outside the scope of this project.

V. 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()*.

VI. URI support

The SIP library supports all URI types defined in section 25 of RFC 3261. The SIP library provides access functions to obtain a URI from SIP header, and also access functions to obtain various components from a URI. A SIP URI can be obtained by getting the URI value from a SIP header and then parsed into its components.

A URI in a SIP message can be a SIP[S] URI or an absolute URI, a URI is internally represented as:

```
/*
 * params     sip uri params
 * headers    sip uri headers
 */
typedef struct sip_uri_sip {
        sip_param_t         *sip_params;
        sip_str_t           sip_headers;
} sip_uri_sip_t;

/*
 * opaque     absolute uri opaque part
 * query      absolute uri query
 * path       absolute uri path
 * regname    absolute uri reg-name
 */
typedef struct sip_uri_abs_s {
        sip_str_t           sip_uri_opaque;
        sip_str_t           sip_uri_query;
        sip_str_t           sip_uri_path;
        sip_str_t           sip_uri_regname;
} sip_uri_abs_t;

/*
 * structure for a parsed URI
 *   scheme     URI scheme
 *   user       user name
 *   password   password for the user
 *   host       host name
 *   port       port number for the host (0 = none specified)
 *   errflags   error flags
 *   issip      B_FALSE means absolute URI, B_TRUE means SIP URI
 *   isteluser  user is a telephone-subscriber
 */
typedef struct sip_uri {
        sip_str_t           sip_uri_scheme;
        sip_str_t           sip_uri_user;
        sip_str_t           sip_uri_password;
        sip_str_t           sip_uri_host;
        uint_t              sip_uri_port;
        uint_t              sip_uri_errflags;
        boolean_t           sip_uri_issip;
        boolean_t           sip_uri_isteluser;
        union {
              sip_uri_sip_t sip_sipuri;     /* SIP URI */
              sip_uri_abs_t sip_absuri;     /* Absolute URI */
        } specific;
}_sip_uri_t;
```

errflags is a bitmask which is set when *sip_parse_uri()* parses a SIP URI string when creating the parsed SIP structure – *_sip_uri_t*. The possible values for *errflags* is listed in **Appendix II.**

VII. 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.

- 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 can provide the appropriate connection object as a parameter (currently it passes NULL).
- 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. 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 header

int *sip_add_header(sip_msg_t sip_msg, char *header_str);*
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 *from_tag, boolean_t add_quote, char *param);*
int *sip_add_to(sip_msg_t sip_msg, char *display_name, char *to_uri, char *to_tag, boolean_t add_quote, char *param);*
int *sip_add_via(sip_msg_t sip_msg, char *sent_protocol_transport, char *sent_by_host, int sent_by_port, char *via_param);*
int *sip_add_maxforward(sip_msg_t sip_msg, uint_t max_forward);*
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_num);*
int *sip_add_content_type(sip_msg_t sip_msg, char *type , char *subtype);*
int *sip_add_content(sip_msg_t sip_msg, char *content);*
int *sip_add_contact(sip_msg_t sip_msg, char *display_name, char *contact_uri, boolean_t add_quote, char *param);*
int *sip_add_route(sip_msg_t sip_msg, char *display_name, char *route_uri, char *route_param);*
int *sip_add_record_route(sip_msg_t sip_msg, char *display_name, char *route_uri, char *route_param);*
int *sip_add_branchid_to_via(sip_msg_t sip_msg, char *branchid);*
int *sip_add_accept(sip_msg_t sip_msg, char *type, char *subtype, char *media_param, char *accept_param);*
int *sip_add_author(sip_msg_t sip_msg, char *scheme, char *param);*
int *sip_add_authen_info(sip_msg_t sip_msg, char *authinfo);*
int *sip_add_proxy_authen(sip_msg_t sip_msg, char *scheme, char *param);*
int *sip_add_proxy_author(sip_msg_t sip_msg, char *scheme, char *param);*
int *sip_add_proxy_require(sip_msg_t sip_msg, char *opt);*
int *sip_add_www_authen(sip_msg_t sip_msg, char *scheme, char *param);*
int *sip_add_accept_enc(sip_msg_t sip_msg, char *code, char *param);*
int *sip_add_accept_lang(sip_msg_t sip_msg, char *lang, char *param);*
int *sip_add_alert_info(sip_msg_t sip_msg, char *alert, char *param);*
int *sip_add_allow(sip_msg_t sip_msg, char *method);*
int *sip_add_call_info(sip_msg_t sip_msg, char *uri, char *param);*
int *sip_add_content_disp(sip_msg_t sip_msg, char *display_type, char *param);*
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 *param);*
int *sip_add_expires(sip_msg_t sip_msg, int sec);*
int *sip_add_in_reply_to(sip_msg_t sip_msg, char *replyid);*
int *sip_add_mime_version(sip_msg_t sip_msg, char *mime_vers);*
int *sip_add_min_expires(sip_msg_t sip_msg, int sec);*
int *sip_add_org(sip_msg_t sip_msg, char *org);*
int *sip_add_priority(sip_msg_t sip_msg, char *priority);*

```
int         sip_add_reply_to(sip_msg_t sip_msg, char *uname, char *addr, char *param,
               boolean_t add_aquot);
int         sip_add_require(sip_msg_t sip_msg, char *require);
int         sip_add_retry_after(sip_msg_t sip_msg, int sec, char *comment,
               char *param);
int         sip_add_server(sip_msg_t sip_msg, char *server);
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, char *delay);
int         sip_add_unsupported(sip_msg_t sip_msg, char *unsupport);
int         sip_add_user_agent(sip_msg_t sip_msg, char *usr);
int         sip_add_warning(sip_msg_t sip_msg, int code, char *addr, char *msg);
int         sip_add_allow_events(sip_msg_t sip_msg, sip_method_t method;
int         sip_add_event(sip_msg_t sip_msg, char *event, char *param);
int         sip_add_substate(sip_msg_t sip_msg, char *sub, char *param);
int         sip_add_privacy(sip_msg_t sip_msg, char *priv_val);
int         sip_add_passertedid(sip_msg_t sip_msg, char *display_name, char *addr,
               boolean_t add_aquot);
int         sip_add_ppreferredid(sip_msg_t sip_msg, char *display_name, char *addr,
               boolean_t add_aquot);
int         sip_add_rack(sip_msg_t sip_msg, int resp_num, int cseq,
               sip_method_t method);
int         sip_add_rseq(sip_msg_t sip_msg, int rseq);
```

Getting SIP header values

```
const sip_str_t   *sip_get_author_scheme(sip_msg_t sip_msg, int *error);
const sip_str_t   *sip_get_author_param(sip_msg_t sip_msg, char *name, int *error);
const sip_str_t   *sip_get_authen_info(sip_header_value_t ainfo_hdr, int *error);
const sip_str_t   *sip_get_proxy_authen_scheme(sip_msg_t sip_msg, int *error);
const sip_str_t   *sip_get_proxy_authen_param(sip_msg_t sip_msg, char *name, int *error);
const sip_str_t   *sip_get_proxy_author_scheme(sip_msg_t sip_msg, int *error);
const sip_str_t   *sip_get_proxy_author_param(sip_msg_t sip_msg, char *name, int *error);
const sip_str_t   *sip_get_proxy_require(sip_header_value_t preqval, int *error);
const sip_str_t   *sip_get_www_authen_scheme(sip_msg_t sip_msg, int *error);
const sip_str_t   *sip_get_www_authen_param(sip_msg_t sip_msg, char *name, int *error);
const sip_str_t   *sip_get_allow_events(sip_header_value_t alloweval, int *error);
const sip_str_t   *sip_get_event(sip_msg_t sip_msg, int *error);
const sip_str_t   *sip_get_substate(sip_msg_t sip_msg, int *error);
const sip_str_t   *sip_get_accept_type(sip_header_value_t accval, int *error);
const sip_str_t   *sip_get_accept_sub_type(sip_header_value_t accval, int *error);
const sip_str_t   *sip_get_accept_enc(sip_header_value_t aencval, int *error);
const sip_str_t   *sip_get_accept_lang(sip_header_value_t alanval, int *error);
const sip_str_t   *sip_get_alert_info_uri(sip_header_value_t ainfoval, int *error);
sip_method_t      sip_get_allow_method(sip_header_value_t allowval, 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_msg_t sip_msg, int *error);
const sip_str_t   *sip_get_replyto_uri_str(sip_msg_t sip_msg, 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_lang(sip_header_value_t clanval, int *error);
const sip_str_t   *sip_get_content_enc(sip_header_value_t cencval, int *error);
const sip_str_t   *sip_get_error_info_uri(sip_header_value_t einfoval, int *error);
int               sip_get_expires(sip_msg_t sip_msg, int *error);
const sip_str_t   *sip_get_require(sip_header_value_t reqval, 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 supval, int *error);
const sip_str_t   *sip_get_tstamp_delay(sip_msg_t sip_msg, int *error);
const sip_str_t   *sip_get_tstamp_value(sip_msg_t sip_msg, int *error);
const sip_str_t   *sip_get_unsupported(sip_header_value_t usupval, 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 warnval, int *error);
const sip_str_t   *sip_get_warning_agent(sip_header_value_t warnval, int *error);
const sip_str_t   *sip_get_warning_text(sip_header_value_t warnval, int *error);
const sip_str_t   *sip_get_call_info_uri(sip_header_value_t cinfoval, int *error);
const sip_str_t   *sip_get_in_reply_to(sip_header_value_t irplytoval, 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);
const sip_str_t   *sip_get_passertedid_display_name(sip_header_value_t paidval, int *error);
const sip_str_t   *sip_get_passertedid_uri_str(sip_header_value_t paidval, int *error);
const sip_str_t   *sip_get_ppreferredid_display_name(sip_header_value_t ppidval, int *error);
const sip_str_t   *sip_get_ppreferredid_uri_str(sip_header_value_t ppidval, int *error);
const sip_str_t   *sip_get_priv_value(sip_header_value_t prival, int *error);
int               sip_get_rack_resp_num(sip_msg_t sip_msg, int *error);
int               sip_get_rack_cseq_num(sip_msg_t sip_msg, int *error);
sip_method_t      sip_get_rack_method(sip_msg_t sip_msg, int *error);
int               sip_get_rseq_resp_num(sip_msg_t sip_msg, int *error);
const sip_str_t   *sip_get_route_uri_str(sip_header_value_t routeval, int *error);
const sip_str_t   *sip_get_route_display_name(sip_header_value_t routeval, int *error);
const sip_str_t   *sip_get_contact_uri_str(sip_header_value_t cval, int *error);
const sip_str_t   *sip_get_contact_display_name(sip_header_value_t cval, int *error);
const sip_str_t   *sip_get_from_uri_str(sip_msg_t sip_msg, int *error);
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_uri_str(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 viaval, int *error);
int               sip_get_via_sent_by_port(sip_header_value_t viaval, int *error);
const sip_str_t   *sip_get_via_sent_protocol_version(sip_header_value_t viaval, int *error);
const sip_str_t   *sip_get_via_sent_protocol_name(sip_header_value_t, int *error);
const sip_str_t   *sip_get_via_sent_transport(sip_header_value_t viaval, int *error);
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);
```

SIP Request/Response functions

```
int               sip_add_response_line(sip_msg_t sip_msg, int code, char *phrase);
int               sip_add_request_line(sip_msg_t sip_msg, sip_method_t method, char *uri);
sip_msg_t         sip_create_response(const sip_msg_t request, int code, char *phrase,
                    char *to_tag, char *contact_uri);
```

```
sip_msg_t          sip_create_dialog_req(sip_method_t method, sip_dialog_t dialog,
                       char *sent_protocol_transport, char *sent_by_host,
                       int sent_by_port, char *via_param, uint32_t maxforward, int cseq);
int                sip_create_OKack(const sip_msg_t response, sip_msg_t ack_msg,
                       char *sent_protocol_transport, char *sent_by_host, int sent_by_port,
                       char *via_param);
boolean_t          sip_msg_is_request(const sip_msg_t sip_msg, int *error);
boolean_t          sip_msg_is_response(const sip_msg_t sip_msg, int *error);
sip_method_t       sip_get_request_method(const sip_msg_t sip_msg, int *error);
const sip_str_t    *sip_get_request_uri_str(sip_msg_t sip_msg, int *error);
int                sip_get_response_code(sip_msg_t sip_msg, int *error);
const sip_str_t    *sip_get_response_phrase(sip_msg_t sip_msg, int *error);
```

Copying headers/messages

```
int                sip_copy_start_line(sip_msg_t from_msg, sip_msg_t to_msg);
int                sip_copy_header(sip_msg_t sip_msg, sip_header_t hdr, char *param);
int                sip_copy_header_by_name(sip_msg_t from_msg, sip_msg_t to_msg,
                       char *hdr_name, char *param);
int                sip_copy_all_headers(sip_msg_t from_msg, sip_msg_t to_msg);
sip_msg_t          sip_clone_msg(const sip_msg_t sip_msg);
```

Deleting SIP headers/values

```
int                sip_delete_start_line(sip_msg_t sip_msg);
int                sip_delete_header(sip_header_t sip_hdr);
int                sip_delete_header_by_name(sip_msg_t sip_msg, char *hdr_name);
int                sip_delete_value(sip_header_t sip_hdr, sip_header_value_t value);
```

SIP Headers lookups

```
const struct sip_header    *sip_get_header(sip_msg_t sip_msg, char *hdr_name,
                               sip_header_t prev_hdr, int *error);
(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_hdr, int *error);
const struct sip_value     *sip_get_next_value(sip_header_value_t value, int *error);
const sip_str_t            *sip_get_param_value(sip_header_value_t value, char *param_name,
                               int *error);
const sip_param_t          *sip_get_params(sip_header_value_t value, int *error);
boolean_t                  sip_is_param_present(const sip_param_t *paramlist,
                               char *param_name, int param_len);
char                       *sip_get_resp_desc(int resp_code);
```

SIP ID generation

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

VIA functions

```
int                sip_get_num_via(sip_msg_t);
char               *sip_get_branchid(const sip_msg_t, int *);
```

Sending SIP messages

int sip_sendmsg(sip_conn_object_t cobj, sip_msg_t sip_msg, sip_dialog_t dialog, uint32_t send_flags);

Processing incoming message

*void sip_process_new_packet(sip_conn_object_t cobj, void *message, size_t msglen);*

Transaction layer functions

*const struct sip_xaction *sip_get_trans(sip_msg_t sip_msg, int type, int *error);*
*char *sip_get_trans_branchid(sip_transaction_t trans, int *error);*
*sip_method_t sip_get_trans_method(sip_transaction_t trans, int *error);*
*int sip_get_trans_state(sip_transaction_t trans, int *error);*
*const struct sip_message *sip_get_trans_resp_msg(sip_transaction_t trans, int *error);*
*const struct sip_message *sip_get_trans_orig_msg(sip_transaction_t trans, int *error);*
void sip_hold_trans(sip_transaction_t trans);
void sip_release_trans(sip_transaction_t trans);
*const struct sip_conn_object *sip_get_trans_conn_obj(sip_transaction_t trans, int *error);*

Dialog layer functions

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

URI functions

*const struct sip_uri *sip_get_request_uri(sip_msg_t sip_msg, int *error);*
*const struct sip_uri sip_get_uri_parsed(sip_header_value_t value, int *error);*
*sip_uri_t sip_parse_uri(sip_str_t *uri_str, int *error);*
void sip_free_parsed_uri(sip_uri_t sip_uri);
*const sip_str_t *sip_get_uri_headers(const struct sip_uri *sip_uri, int *error);*
*const sip_param_t *sip_get_uri_params(const strucr sip_uri *sip_uri, int *error);*
*boolean_t sip_is_sipuri(const struct sip_uri *sip_uri);*
*const sip_str_t *sip_get_uri_scheme(const struct sip_uri *sip_uri, int *error);*
*const sip_str_t *sip_get_uri_user(const struct sip_uri *sip_uri , int *error);*
*const sip_str_t *sip_get_uri_password(const struct sip_uri *sip_uri , int *error);*
*const sip_str_t *sip_get_uri_host(const struct sip_uri *sip_uri , int *error);*
*int sip_get_uri_port(const struct sip_uri *sip_uri , int *error);*
*const sip_str_t *sip_get_uri_opaque(const struct sip_uri *sip_uri , int *error);*
*const sip_str_t *sip_get_uri_query(const struct sip_uri *sip_uri , int *error);*

```
const sip_str_t     *sip_get_uri_path(const struct sip_uri *sip_uri , int *error);
const sip_str_t     *sip_get_uri_regname(const struct sip_uri *sip_uri , int *error);
boolean_t           sip_is_uri_teluser(const struct sip_uri *sip_uri );
int                 sip_get_uri_errflags(const struct sip_uri *sip_uri , int *error);
char                *sip_uri_errflags_to_str(int uri_errflags);
```

Connection object functions

```
int         sip_conn_send(const sip_conn_object_t cobj, char *msg, int msglen);
void        sip_hold_conn_object(sip_conn_object_t cobj);
void        sip_rel_conn_object(sip_conn_object_t cobj);
boolean_t   sip_conn_is_reliable(sip_conn_object_t cobj);
boolean_t   sip_conn_is_stream(sip_conn_object_t cobj);
int         sip_conn_remote_address(sip_conn_object_t cobj,
              struct sockaddr *name, socklen_t *namelen);
int         sip_conn_local_address(sip_conn_object_t cobj,
              struct sockaddr *name, socklen_t *namelen);
int         sip_conn_transport(sip_conn_object_t cobj);
int         sip_conn_timer1(sip_conn_object_t cobj);
int         sip_conn_timer2(sip_conn_object_t cobj);
int         sip_conn_timer4(sip_conn_object_t cobj);
int         sip_conn_timerd(sip_conn_object_t cobj);
```

Miscellaneous functions

```
char            *sip_msg_to_str(sip_msg_t sip_msg, int *error);
char            *sip_hdr_to_str(sip_header_t 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);
const sip_str_t *sip_get_sip_version(sip_msg_t sip_msg, int *error);
int             sip_get_msg_len(sip_msg_t sip_msg, int *error);
int             sip_init_conn_object(sip_conn_object_t cobj);
void            sip_clear_stale_data(sip_conn_object_t cobj);
void            sip_conn_destroyed(sip_conn_object_t cobj);
char            *sip_sent_by_to_str(int *error);
int             sip_register_sent_by(char *sent_by);
void            sip_unregister_sent_by(char *sent_by);
void            sip_unregister_all_sent_by();
```

Appendix II. 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,
    SUBSCRIBE,
    NOTIFY,
    PRACK
}sip_method_t;
```

SIP 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;

#define SIP_STACK_VERSION   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);
    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);
}sip_io_pointers_t;

typedef struct sip_ulp_pointers_s {
    void                    (*sip_ulp_recv)(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 *);
    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);
}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_11
```

SIP flags

SIP_SEND_STATEFUL	flag to *sip_sendmsg()* to send a request or response statefully.
SIP_DIALOG_ON_FORK	flag to *sip_sendmsg()* instructs the library if multiple dialogs should be created for a dialog creating request (as a result of forking).
SIP_STACK_DIALOGS	flag for *sip_stack_flags* when initializing the stack to indicate that the library must maintain dialogs.

SIP URI Errors

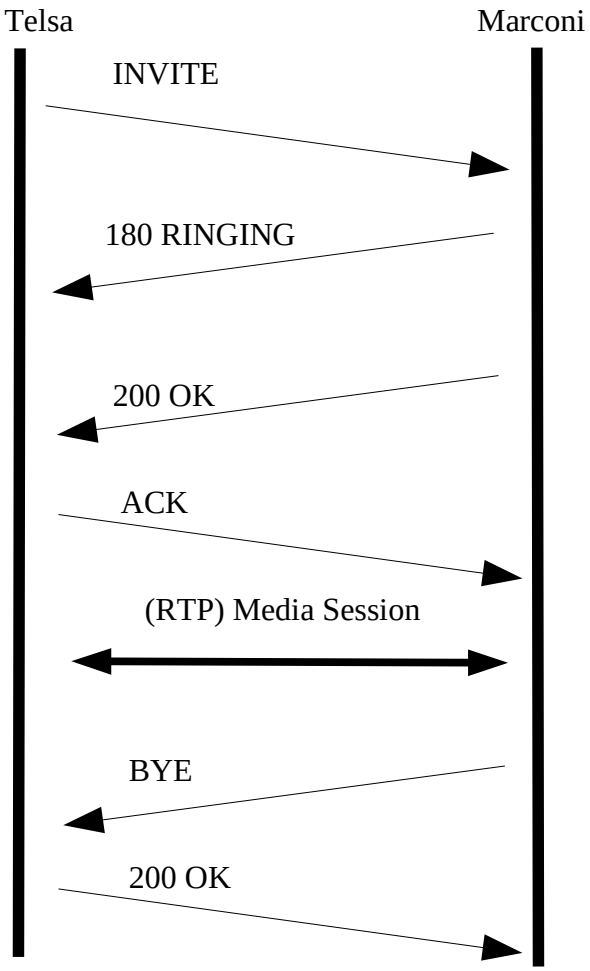
```
/* URI parse errors */

#define SIP_URIERR_SCHEME       0x00000001
#define SIP_URIERR_USER         0x00000002
#define SIP_URIERR_PASS         0x00000004
#define SIP_URIERR_HOST         0x00000008
#define SIP_URIERR_PORT         0x00000010
#define SIP_URIERR_PARAM        0x00000020
#define SIP_URIERR_HEADER       0x00000040
#define SIP_URIERR_OPAQUE       0x00000080
```

```
#define SIP_URIERR_QUERY        0x00000100
#define SIP_URIERR_PATH         0x00000200
#define SIP_URIERR_REGNAME      0x00000400
#define SIP_URIERR_NOURI        0x00000800
```

Appendix III. Example usage Scenario

The following (taken from “SIP Understanding the Session Initiation Protocol” “ by Alan Johnston) shows message exchange between two SIP devices. The following is a call flow diagram for this example, each arrow in the diagram is a SIP message, with the arrowhead indicating the direction of the transmission. It is assumed that both parties are connected to an IP network and know each other's IP address.



Application at both ends initialize the SIP stack as follows:

```
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 – see Connection Manager,
 *  Page 22 - should be defined
 */
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;
sip_io->sip_conn_transport = my_conn_transport;

/*
 *  Initialize callback registrations - my_ulp_recv should be defined.  Not registering
 *  optional callback functions.
 */
bzero(&sip_ulp, sizeof (sip_ulp_pointers_t));
sip_ulp.sip_ulp_recv = my_ulp_recv;

/* stack does not manage dialogs */
sip_init->sip_version = SIP_STACK_VERSION;
sip_init->sip_io_pointers = sip_io;

sip_init->sip_ulp_pointers = &sip_ulp;

if (sip_stack_init(sip_init) != 0) {
        exit(0);
}
```

Connection Establishment (assuming TCP)

Application on Tesla establishes a TCP connection with Marconi on port 5060. The connection is represented by a connection object – conn_obj. The connection object must be initialized using:

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

Tesla: Sending the INVITE request

Assuming the following SIP INVITE message is to be sent (not all fields in the example from the book shown):

```
INVITE sip:marconi@radio.org SIP/2.0
Via: SIP/2.0/TCP lab.high-voltage.org:5060;branch=z9hG4bK-1
From: Nik Tesla <sip:n.tesla@high-voltage.org>;tag=from1
To: G. Marconi <sip:marconi@radio.org>
Call-ID: 123456789@lab.high-voltage.org
CSeq: 1 INVITE
Contact: sip:n.tesla@high-voltage.org
Content-Type: application/sdp
Content-Length: 158

v=0
o=Tesla 2890844526 2890844526 IN IP4 lab.high-voltage.org
s=Phone Call
c=IN IP4 100.101.102.103
t=0 0
m=audio 49170 RTP/AVP 0
a=rtpmap:0 P8000
```

```
sip_msg = sip_new_msg();
if (sip_msg == NULL)
        goto error_exit;
if (sip_add_request_line(sip_msg, INVITE, "<sip:n.tesla@high-voltage.org>") != 0)
        goto error_exit;
if (sip_add_via(sip_msg, "TCP", "lab.high-voltage.org", 5060, "branch=z9hG4bK-1") != 0)
        goto error_exit;
if (sip_add_from(sip_msg, "Nik Tesla", "sip:n.tesla@high-voltage.org", "from1", B_TRUE, NULL) !=
0)
        goto error_exit;
if (sip_add_to(sip_msg, "G. Marconi", "sip:Marconi@radio.org", NULL, B_TRUE, NULL) != 0)
        goto error_exit;
if (sip_add_callid(sip_msg, "123456789@lab.high-voltage.org") != 0)
        goto error_exit;
if (sip_add_cseq(sip_msg, INVITE, 1) != 0)
        goto error_exit;
if (sip_add_contact(sip_msg, NULL, "sip:n.tesla@high-voltage.org", B_TRUE, NULL) != 0)
        goto error_exit;
if (sip_add_content_type(sip_msg, "application", "sdp") != 0)
        goto error_exit;

/* assume the string 'content' contains the message body i.e. "v=0....a=rtpmap:0 P8000" */
if (sip_add_content(sip_msg, content) != 0)
        goto error_exit;

/* Send the INVITE message */
if (sip_sendmsg((sip_conn_object_t)conn_obj, sip_msg, NULL, SIP_SEND_STATEFUL) != 0)
        goto error_exit;
sip_freemsg(sip_msg);
return (success);
error_exit:
        /*
         * If we can never send the INVITE, then there is no session established. There is no
         * clean-up required in the stack
         */
```

```
/* free the SIP message */
if (sip_msg != NULL)
        sip_free_msg(sip_msg);
/* free connection object */
sip_conn_destroyed((sip_conn_object_t)conn_obj);
close(connection);
return (error);
```

Marconi: Sending a 180 response followed by a 200 response

```
/* Assuming sip_msg is the request delivered by the stack */

sip_msg_t sip_msg_resp;

/* Create a 180 response with the To tag set to to1 and no contact header */
```

SIP/2.0 180 Ringing
VIA: SIP/2.0/TCP lab.high-voltage.org:5060;branch==z9hG4bK-1
FROM: Nik Tesla <sip.n.tesla@high-voltage.org>;tag=from1
TO: G. Marconi <sip:marconi@radio.org>
CALL-ID: 123456789@lab.high-voltage.org
CSEQ: 1 INVITE
Content-Length : 0

```
sip_msg_resp = sip_create_response(sip_msg, SIP_RINGING,
   sip_get_resp_desc(SIP_RINGING), "to1", NULL);

/*
 * Close connection, this might cause the peer to destoy the connection object at the other end, resulting
 * in the INVITE transaction being terminated.
 */
if (sip_msg_resp == NULL) {
        sip_conn_destroyed(sip_conn_object_t)conn_obj);
        close(connection);
        return;
}

/* Send the 180 response */
if (sip_sendmsg((sip_conn_object_t)conn_obj, sip_msg_resp, NULL, SIP_SEND_STATEFUL) !
   = 0) {
        sip_free_msg(sip_msg_resp);
        sip_conn_destroyed((sip_conn_object_t)conn_obj);
        close(connection);
        return;
}
/* free the response */
sip_free_msg(sip_msg_resp);

/*
 * Create a 200 OK response with the To tag set to to1 and contact set
 * to "sip:marconi@radio.org".
 * If this message is to be sent later, then the request message – sip_msg – will need to be held
 * using sip_hold_msg() and saved some place. After the response is sent if the request is no
 * longer needed, it must be released using sip_free_msg().
 */
```

SIP/2.0 200 OK
VIA: SIP/2.0/UDP lab.high-voltage.org:5060;branch==z9hG4bK-1
FROM: Nik Tesla <sip.n.tesla@high-voltage.org>;tag=from1

TO: G. Marconi <sip:marconi@radio.org>;tag=to1
CALL-ID: 123456789@lab.high-voltage.org
CSEQ: 1 INVITE
Contact : sip:marconi@radio.org
Content-Type : application/sdp
Content-Length: 155

v=0
o=Marconi 2890844528 2890844528 IN IP4 tower.radio.org
s=Phone Call
c=IN IPV4 200.201.202.203
t=0 0
m=audio 60000 RTP/AVP 0
a=rtpmap:0 PCMU/8000

```
sip_msg_resp = sip_create_response(sip_msg, SIP_OK, sip_get_resp_desc(SIP_OK),
   "to1", "sip:marconi@radio.org");
if (sip_msg_resp == NULL) {
        sip_conn_destroyed((sip_conn_object_t)conn_obj);
        close(connection);
        return;
}
/* Add additional fields */
if (sip_add_content_type(sip_msg_resp, "application", "sdp") != 0) {
        sip_free_msg(sip_msg_resp);
        sip_conn_destroyed((sip_conn_object_t)conn_obj);
        close(connection);
        return;
}

/* assuming 'content' contains the message body, i.e. "v=0...a=rtpmap:0 PCMU/8000" */
if (sip_add_content(sip_msg_resp, content) != 0) {
        sip_free_msg(sip_msg_resp);
        sip_conn_destroyed((sip_conn_object_t)conn_obj);
        close(connection);
        return;
}

/* Send the 200 OK response */
if (sip_sendmsg((sip_conn_object_t)conn_obj, sip_msg_resp, NULL, SIP_SEND_STATEFUL) != 0) {
        sip_free_msg(sip_msg_resp);
        sip_conn_destroyed((sip_conn_object_t)conn_obj);
        close(connection);
        return;
}

/* free the response */
sip_free_msg(sip_msg_resp);
return;
```

Tesla: Sending the ACK request

```
/* Assume sip_msg is the 200 OK response sent up by the stack */

sip_msg_t sip_ack_msg;

/* Create the OK ack */
        ACK sip:marconi@radio.org SIP/2.0
        VIA: SIP/2.0/TCP lab.high-voltage.org:5060;branch=z9hG4bK-ack
        FROM: Nik Tesla <sip:n.tesla@high-voltage.org>;tag=from1
        TO: G. Marconi <sip:marconi@radio.org>;tag=to1
        CALL-ID: 123456789@lab.high-voltage.org
        CSEQ: 1 ACK
        Content-Length : 0

sip_ack_msg = sip_new_msg();
if (sip_ack_msg == NULL) {
        /*
         * If we don't send the ACK, the UAS will, per RFC 3261, keep retransmitting the
         * the response. If we can never send the ACK, the UAS will time out and consider
         * the sesssion terminated. If we had a dialog associated with this session, we will
         * also delete it here.
         */
        sip_conn_destroyed((sip_conn_object_t)conn_obj);
        close(connection);
        return;
}

if (sip_create_OKack(sip_msg, sip_ack_msg, "TCP", "lab.high-voltage.org", 5060,
   "branch=z9hG4bK-ack") != 0) {
        sip_free_msg(sip_ack_msg);
        sip_conn_destroyed((sip_conn_object_t)conn_obj);
        close(connection);
        return;
}

/* Send the ACK message */
if (sip_sendmsg((sip_conn_obj_t)conn_obj, sip_ack_msg, NULL, SIP_SEND_STATEFUL) != 0) {
        sip_free_msg(sip_ack_msg);
        sip_conn_destroyed((sip_conn_object_t)conn_obj);
        close(connection);
        return;
}

/* free the ack message */
sip_free_msg(sip_ack_msg);
```

At this point the media session begins using the media information carried in the SIP messages. The media session takes place using another protocol, typically RTP.

Marconi: Terminating the media session

/ Send a BYE request to terminate the session */*

BYE: sip:n.tesla@high-voltage.org SIP/2.0
VIA: SIP/2.0/TCP tower.radio.org:5060;branch=z9hG4bK-2
FROM: G. Marconi <sip:marconi@radio.org>;tag=to1
TO: Nik Tesla<sip:n.tesla@high-voltage.org>
Call-ID : 123456789@lab.high-voltage.org
CSEQ : 1 BYE
Content-Length : 0

```
sip_msg = sip_new_msg();
if (sip_msg == NULL)
        goto error_exit;
if (sip_add_request_line(sip_msg, BYE, "sip:n.tesla@high-voltage.org") != 0)
        got error_exit;
if (sip_add_via(sip_msg, "TCP", "tower.radio.org", 5060, "branch=z9hG4bK-2") != 0)
        goto error_exit;
if (sip_add_from(sip_msg, "G. Marconi", "sip:marconi@radio.org", "to1", B_TRUE, NULL) != 0)
        goto error_exit;
if (sip_add_to(sip_msg, "Nik Tesla", "sip:n.tesla@high-voltage.org", NULL, B_TRUE, NULL) != 0)
        goto error_exit;
if (sip_add_callid(sip_msg, "123456789@lab.high-voltage.org") != 0)
        goto error_exit;
if (sip_add_cseq(sip_msg, BYE, 1) != 0)
        goto error_exit;

/* send the BYE request */
if (sip_sendmsg((sip_conn_object_t)conn_obj, sip_msg, NULL, SIP_SEND_STAEFUL) != 0)
        goto error_exit;

/* free the SIP message */
sip_free_msg(sip_msg);

return(success);
error_exit:
        /*
         * If we never send the BYE, then the peer will detect that the media session is down and
         * do any clean-up, if necessary.
         */
        if (sip_msg != NULL)
                sip_free_msg(sip_msg);
        sip_conn_destroyed(sip_conn_object_t)obj);
        close(connection);
        return (failure);
```

Tesla: Confirmation response to the BYE request

/ Assuming sip_msg is the BYE request delivered by the stack */*

/ Create a 200 OK response with the To tag set to from1 and no contact header */*

SIP/2.0 200 OK
VIA: SIP/2.0/TCP tower.radio.org:5060;branch==z9hG4bK-2
TO: Nik Tesla <sip.n.tesla@high-voltage.org>;tag=from1
FROM: G. Marconi <sip:marconi@radio.org>;tag=to1
CALL-ID: 123456789@lab.high-voltage.org
CSEQ: 1 BYE

Content-Length : 0

```
sip_msg_resp = sip_create_response(sip_msg, SIP_OK, sip_get_resp_desc(SIP_OK), “from1”, NULL);
if (sip_msg_resp == NULL) {
        /*
         * If we never send the response than the BYE transaction at the peer will time out and
         * result in any required clean-up.
         */
        sip_conn_destroyed((sip_conn_object_t)conn_obj);
        close(connection);
        return;
}

/* Send the 200 response */
if (sip_sendmsg((sip_conn_object_t)conn_obj, sip_msg_resp, NULL, SIP_SEND_STATEFUL) ! = 0) {
        sip_free_msg(sip_msg_resp);
        sip_conn_destroyed((sip_conn_object_t)conn_obj);
        close(connection);
        return;
}


/* free the response */
sip_free_msg(sip_msg_resp);
```

Appendix IV. SIP Transaction timers

Timer	*Default Value*	*Section in RFC 3261*	*Meaning*
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.