Aculab Extended SIP API guide

MAN1762 Revision 6.7.0

PROPRIETARY INFORMATION

Document Revision

Rev	Date	By	Detail
1.0.0	23.06.05	DJL	Initial controlled release
1.0.1	31.11.05	DJL	Updates following reviews/feedback
1.0.2	18.01.06	DJL	Additional small changes
1.0.3	21.02.06	DJL	`sip_openout cnf` changed for `CNF_TSVIRTUAL`.
1.0.4	15.05.06	DJL	Updates to sections 2.17 and 2.18
1.0.5	18.05.06	DJL	Additional clarification added to section 1.2
1.1.0	06.07.06	SP	Call redirect features added.
1.2.0	05.10.06	MG	TLS, Out of dialog, session timer features added. T.38 attributes removed from `ACU_PAYLOAD`: now in `ACU_MISCELLANEOUS_MEDIA_ATTRIBUTES`.
1.2.1	09.01.07	MG	Configuration of distributed operation information added.
6.4.0	17.01.07	DJL	Updated revision status to reflect current telephony software
6.4.100	05.09.08	SP	Updated with new sipserv.cfg options sip_send_response added.
6.4.126	07.04.09	SP	Added sip_load_tls_configuration. Removed TLS plugin. Added sip_send_invite_response. New configuration options.
6.4.152	21.01.10	NC	Added DRSS section
6.4.153	11/03/10	EBJ	Updated to corporate fonts.
6.4.154	01/04/10	NC	Added extra features to sip_disconnect, fixed errors in sip_incoming_ringing and sip_send_out_of_dialog_request.
6.4.161.8	12/08/10	NC	Updated DRSS information.
6.4.161.13	20.10.10	DF	Changed name of document

Rev	Date	By	Detail
6.4.161.14	17.12.10	NC	Extra features to `sip_disconnect()`, fixes and updates to configuration section, aligned with current cl_lib.h
6.4.163	08.03.11	NC	Extended the `sip_send_request()` and `sip_send_response()` structures to include `ACU_MEDIA_OFFER_ANSWER` for supporting `UPDATE` with SDP body. Updated SIP specific events for `EV_MEDIA_PROPOSE`
6.4.166	15.07.11	NC	Update – no MHP compatibility with DRSS. Update – DRSS for ringing calls Clarify procedure when EV_MEDIA_PROPOSE is received.
6.4.167	23.07.12	NMC	Added `ACU_DO_NOT_USE_PROXY_IF_SET` to `ACU_SIP_CALL_OPTIONS` in sip_openout()
6.5.0	26.09.12	NC	APIs for `SUBSCRIBE` & `NOTIFY`, IPv6 clarifications
6.5.1	15.02.13	NC	How to use session level attributes in SDP
6.5.2	16.01.14	NC	Removed reference to unsupported early media option. Expanded description of configuration options to include MHP New event for SUBSCRIBE refresh failure, clarified what happens if subscriptions refresh was not successful
6.5.3	04.07.14	NC	Config options, sip_openout() options and updates for async calls. Updated IPv6 listening addresses. Clarified send_early_media flag in sip_incoming_ringing.
6.5.4	06.10.15	NMC	Additional config options. Added custom_headers to sip_media_propose and sip_media_request_proposal
6.5.5	19.11.15	NMC	Added EV_SIP_RSS_IPTAKEOVER_SUCCESSFUL event to indicate when the floating address has been successfully acquired
6.5.6	21.03.16	NMC	DelayTrying is an option added to the sipserv.cfg When set it disables the automatic sending of ‘100 trying’ response
6.5.7	19.10.16	NMC	Callid added to SIP_OUT_PARMS in sip_openout()
6.5.8	26.10.16	NMC	SSL sip options to remove support for various SSL and TLS types configurable by sipserv.cfg
6.5.9	03.01.17	NMC	ACU_RAISE_MEDIA_EVENT_FOR_HOLD introduced to sip_openout() and sip_openin() in call_options
6.5.10	08.02.17	NMC	New function sip_set_global_tos()

Rev	Date	By	Detail
6.7.0	10.03.17	NMC	ACU_RAISE_FORKED_EVENT introduced to sip_openout() in call_options and new event EV_FORKED

CONTENTS

1 Introduction

This guide describes API functions comprising an extended SIP API coexistent with the Aculab Generic call control API. The objective of the extended SIP API is to provide a set of SIP features that cannot be represented in a protocol independent manner.

The SIP service has been written to support basic call control and the following extra features:

- Third party call control using re-`INVITE`. Enabling the re-direction of RTP streams between endpoints.
- Fine control of SDP content in media negotiation; this can be used to facilitate third party call control, non-audio payload type, and multiple RTP streams.
- Presentation of raw SIP messages to applications; selected types of SIP messages may be passed to the application for bespoke parsing.
- Custom headers and message bodies in the call setup `INVITE`, adding greater flexibility to call setup and providing transparency for signalling links with IP legs.
- Mid-call signalling with custom headers and message bodies; providing for additional features to be implemented in applications such as forwarding PSTN information and MWI (message waiting indication).
- Unlimited sizes of data can be passed, and received as details, by the API; `char*` types and linked lists are used to represent types which can be repeated within messages.
- Authentication and security. HTTP style message digest authentication permitting controlled access to secure domains and TLS encryption, providing secrecy for signalling messages.
- SIP session timer; a mechanism, which detects lost signalling endpoints throughout call duration, facilitating preservation of signalling resource.
- Out of dialog message control. Permitting the fine control of messages not associated with calls, such as `REGISTER` and `OPTIONS`, to be handled by the SIP application.
- Support for SIP Specific Event Notification (RFC 3265)

CAUTION

This document is intended for use in conjunction with the SIP Specification RFC 3261.

Further guidance may also be found in the following SIP IETF documents:

RFC 2246 – The TLS protocol
RFC 2976 – INFO method
RFC 3204 – MIME media types for QSIG/ISUP
RFC 3262 – Reliability of Provisional Responses in SIP
RFC 3264 – Offer/Answer model with SDP
RFC 3265 – SIP Specific Event Notification
RFC 3311 – UPDATE method
RFC 3326 – The Reason Header Field for SIP
RFC 3428 – SIP Message Extension
RFC 3515 – The REFER method
RFC 3581 – rport parameter
RFC 3665 – Basic call flow examples
RFC 3666 – SIP/PSTN call flows
RFC 3725 – Third party call control best practices
RFC 3891 – Replaces header
RFC 3966 – Tel URI for telephone numbers
RFC 4028 – SIP session timer
RFC 4566 – Session Description Protocol (SDP)
RFC 5589 – Hold and attended transfer.
RFC 5589 – Blind transfer
RFC 6157 – IPv6 Transition

Other RFCs may be implemented using the functionality provided in this document.

Before using these features, the user should be familiar with the Call Control API Guide. For a more detailed description of the scenarios where this API may be used, please refer to the SIP Programmers Guide, both available at www.aculab.com. Some functionality which affects the SIP service is provided by the IP Telephony API Guide such as configuring listening ports and managing registrations.

This guide does not presume any particular environment, and is intended for use under various operating systems. As such, the functions are defined as library calls where isolation from the operating system is desired.

1.1 Scope

This guide is intended to be of use in the development of applications that make use of the various function calls.

This specification describes the initiation and control of an outgoing call, the reception and control of an incoming call, and support of various SIP features.

1.2 Overlap of the extended SIP API and generic call control API

In order to minimise the duplication of effort, there is some overlap in the use of APIs. For example, to hold a call opened using `sip_openout()`, it is necessary to use `call_hold()` as `sip_hold()` has not been implemented. Similarly, disconnect and release functionality is that provided by the generic API, there is no requirement to provide a SIP API specific function to achieve this task.

For functionality where there are both `call_` and `sip_` instances of the same function, for example, with `call/sip_feature_send()`, it is necessary to choose the routine consistent with the method of opening. So, if a call is created using `call_openout()` then use `call_feature_send()`, else if the call is created using `sip_openout()` then use `sip_feature_send()`.

Where an extended API call has no equivalent in the generic API, some extended API calls may be used with both APIs and others may not. These cases are dealt with individually in the following sections.

The extended SIP API functions may be categorised as follows:

1.2.1 Extended versions of existing generic API functions

`sip_openout()`
`sip_openin()`
`sip_accept()`
`sip_details()`
`sip_incoming_ringing()`
`sip_progress()`
`sip_feature_send()`

These routines play similar roles to those matching functions in the generic API, however they accept SIP specific parameters in particular structures used in the detailed configuration of SIP messages, SDP bodies, and other message body types.

These routines require that the *net* on which the call was opened be obtained from `sip_open_port()`.

1.2.2 SIP/SDP specific routines targeted towards third party call control and re-INVITE handling

`sip_media_propose()`
`sip_media_accept()`
`sip_media_request_proposal()`
`sip_media_reject_proposal()`

These routines have no analogues in the generic API and were added to assist applications implementing third party call control and therefore require that the call *handle* be opened using `sip_openout()`/`sip_openin()`.

1.2.3 Routines supporting the extended architecture

sip_open_port()
sip_free_details()

The routines in section 1.2.1 and 1.2.2 act on calls not directly associated with a media resource at the protocol stack level. A different port_id is required to facilitate this, not to be confused with those port_ids discovered using Aculab's standard hardware-centric port discovery mechanism (as used by the generic API). sip_open_port() provides this port_id.

As the size of the *details* information of a SIP call cannot be, generally, known in advance of a details call, a fully expressive sip_details() function is only possible if the implementation dynamically allocates the structures required after the query to the protocol stack. A function, sip_free_details() is therefore used to return the memory to the OS once the details have been processed.

1.2.4 Miscellaneous

sip_send_request()
sip_send_response()
sip_set_reason_phrase()
sip_add_answer_challenge_credentials()
sip_remove_answer_challenge_credentials()
sip_recall()
sip_disconnect()
sip_set_tls_private_key_password()
sip_send_invite_response()

sip_send_request(), sip_send_response() and sip_send_invite_response() may only be called on a call handle opened with the extended SIP API.

sip_set_reason_phrase(), sip_add_answer_challenge_credentials(),sip_remove_answer_challenge_credentials(), sip_load_tls_configuration() and sip_set_tls_private_key_password() are not related to call handles and may be called by either an extended or generic API based application.

sip_recall() and sip_disconnect() implement redirection in SIP. These may only be called on a call created by sip_openout() (in the case of sip_recall()) and sip_openin() (in the case of sip_disconnect()).

1.2.5 Out of dialog messages

sip_set_message_notification()

sip_send_out_of_dialog_request()

sip_send_out_of_dialog_response()

sip_read_request()

sip_read_response()

sip_read_timeout()
sip_free_message()

The above routines provide the application writer with facilities to control the transmission and reception of 'out of dialog' messages. 'Out of dialog' messages are not associated with telephony calls, nor are they distinguished by an Aculab call handle, instead a number representing the underlying SIP transaction identifies the relevant messages.

The transaction identifier field used in the extended SIP API is of type ACU_POINTER. For compatibility with 64-bit architectures, this field is a 64-bit integer. Application writers are warned to pay attention to formatting requirements when printing this type of field, for instance the Microsoft C/C++ compiler uses %I64, whereas the GCC uses %ll.

1.2.6 Subscription API

sip_sub_subscriber()

sip_sub_notifier()

sip_sub_accept()

sip_sub_notify()

sip_sub_cancel()

sip_sub_release()

sip_sub_fetch()

The above routines provided the application writer with the means to implement any SUBSCRIBE/NOTIFY event package based on RFC 3265, either as a subscriber or a notifier. Although subscriptions are very different to calls in protocol terms, except perhaps that they both create dialogs, they are analogous in many ways with regards to the API. A subscription is created with a call to either sip_sub_subscriber() or sip_sub_notifier() both of which return an ACU_CALL_HANDLE which is to be used in subsequent API calls relating to that particular subscription. sip_details() is called after events are raised and many of the existing API structures are re-used. In particular, ACU_RAW_MESSAGE_BODY is likely to get considerably more use through these APIs.

RFC 3265 is does not define any concrete event package, only the underlying subscription mechanism. As such it is not possible for the SIP service to collect meaningful information relating to a specific event package. For this reason, raw SIP messages are presented to the API through sip_details() so that the application may interpret the relevant headers and message bodies themselves. The maintenance of subscriptions is handled by the SIP service.

1.2.7 Transports

TCP, UDP, and TLS transports are all available using the Aculab SIP service.

1.2.8 IPv6

Beginning with SIP version 6.6.0 IPv6 is supported.

1.2.9 Asynchronous Calls

Beginning with SIP 6.5.14 it is possible to use `sip_openout()`, `sip_send_out_of_dialog_request()` and `sip_sub_subscriber()` in an asynchronous mode. When these calls are invoked normally (synchronously) delays in resolving the target URI can result in the call being held up for a number of seconds. If serious DNS issues occur then this delay may result in an unrecoverable `-507` error.

A new option called `ACU_USE_ASYNC_MODE` may be used with `sip_openout()` which returns control to the application immediately. The SIP service will continue to resolve the target URI without holding up the call. If the target cannot be resolved an `EV_IDLE` is raised instead of `EV_WAIT_FOR_OUTGOING`, and the call will fail. There will be no cause value available if the call fails in this way. If successful the event `EV_WAIT_FOR_OUTGOING` will be raised and the call will proceed as before.

Two new flags `AsyncOOD` and `AsyncSubscribe` can be used in sipserv.cfg.

If `AsyncOOD` is set to 1 then the call to `sip_send_out_of_dialog_request()` will return control to the application immediately. The SIP service will continue to resolve the target URI without holding up the call. If the target cannot be resolved the event `ACU_SIP_EV_REQUEST_FAILED` is raised instead.

If `AsyncSubscribe` is set to 1 then the call to `sip_sub_subscriber()` will return control to the application immediately. The SIP service will continue to resolve the target URI without holding up the call. If the target cannot be resolved the event `EV_SIP_SUBSCRIPTION_CANCELLED` is raised instead.

2 Configuring the SIP service

The SIP service may be configured to exhibit certain global behaviours by use of a configuration file. This file must be called *sipserv.cfg* and must be placed in the `$ACULAB_ROOT/cfg` directory. It is not provided with the distribution, but a default configuration file may be generated by running `sipserv -g` from the command line. This file will contain all of the possible parameters that may be passed to the SIP service. What follows is a description of each of these parameters:

2.1 SIP service logging configuration

```
Loglevel = n
```

This parameter sets the amount of diagnostic logging that the SIP service produces. It may take on a value from 0 (no logging) to 5 (maximum logging).

```
Logfile = <N>
```

This parameter determines whether the logging is to be written to a buffered file (`$ACULAB_ROOT/log/SIP_[TIME]_[DATE].log`) or to be collected by the v6trace utility. It may take either the value `0` (`v6trace`) or `some positive N` (`log file`). If it is non-zero, `v6trace` will not produce any logging and the service will create N log files before it overwrites the first one.

```
OutgoingTrace = <0 or 1>
```

```
IncomingTrace = <0 or 1>
```

These parameters determine whether incoming and/or outgoing SIP protocol messages are logged in either `sipserv.log` or `v6trace`. They may both take either the value `0` (no logging) or `1` (logging).

```
StackTrace = <0 or 1>
```

This parameter determines whether diagnostic logging from the protocol stack is enabled or disabled. It may take either the value `0` (disabled) or `1` (enabled). This trace is also logged either in `sipserv.log` or `v6trace`.

```
MaxLogfileSize = <NUM BYTES>
```

This parameter sets the maximum size of the log file. It is a buffered log file and when the maximum size is reached, the service will overwrite previous logging from the beginning.

```
OverwriteLogfile = <0 or 1>
```

The log file used to be written to `$ACULAB_ROOT/log/sipserv.log` and this file would be overwritten each time the service was started. This is not always desirable so the behaviour was changed to the above. Set this parameter to `1` to restore the original behaviour.

All of the preceding parameters should, in general, be disabled at all times. Their use is to be limited to development environments and/or under the direction of Aculab Support. It should be noted that the SIP service would run more slowly when logging is enabled.

The remaining parameters may be enabled or disabled at the user's discretion. This list will grow as more functionality is implemented within the SIP service.

2.2 Session timer configuration

The following parameters configure the session timer's use.

`EnableSessionTimer = <0 or 1>`

Set this parameter to 1 to enable the session timer.

`MinimumSessionInterval = n`

Set this parameter to a value in seconds that represents the minimum session interval that the application is willing to accept. This parameter corresponds to the SIP header, '`Min-SE`' and defaults to 90 seconds as the smallest value possible mandated by RFC 4028. Other user agents or proxies may raise this minimum during session set up.

`PreferredSessionInterval = n`

Set this parameter to a value in seconds that represents the interval the application would like to use between session refresh requests. This parameter corresponds to the SIP header, '`Session-Expires`' and defaults to 1800 as recommended by RFC 4028. Other user agents or proxies may lower this value during session set up.

`UseInviteForSessionRefresh = <0 or 1>`

If the remote party supports the `UPDATE` method, the SIP service will use `UPDATE` for the session refresh requests. Set this value to 1 if the application requires the use of `INVITE` instead.

`PreferredRefresher = uac / uas`

When an `INVITE` is received from a UAC containing a '`Session-Expires`' header and no '`refresher`' parameter, it is up to the UAS to decide which party is to perform the refreshes. Set this value to either `uac` or `uas` to make this decision. `uac` is used by default.

Note: The session timer may be enabled in the config file but if the remote end does not support timers it will not be utilised. This is to prevent the case where PreferredRefresher = uac and the server end expects a refresh and does not receive one in the allotted time and thus hangs up the call. The server end does not assume the role of refresher in this instance.

2.3 SIP header configuration

By default, the following SIP headers are added to relevant SIP messages:

Allow headers: `INVITE, ACK, BYE, CANCEL, OPTIONS, NOTIFY, REFER, PRACK, INFO, UPDATE` and `REGISTER`

Accept headers: `application/sdp, application/isup, application/qsig` and `multipart/mixed`

Supported headers: `replaces` and `100rel`. (`Supported: timer` is also added if the session timer is enabled.)

These defaults may be overridden using the following parameters.

```
AllowUseSupplied = <0 or 1>
```

If this is set to `1`, none of the default Allow headers will be used and the user must configure them with the following parameter:

```
Allow = <sip method>
```

This parameter may (and almost certainly will) be used multiple times.

```
Allow = INVITE
Allow = ACK
Allow = CANCEL
```

```
AcceptUseSupplied = <0 or 1>
```

If this is set to `1`, none of the default Accept headers will be used and the user must configure them with the following parameter:

```
Accept = <body-type[/sub-type]>
```

This parameter may be used multiple times.

```
Accept = application/sdp
Accept = application/qsig
```

```
SupportedUseSupplied = <0 or 1>
```

If this is set to `1`, none of the default Supported headers will be used and the user must configure them with the following parameter:

```
Supported = <option-tag>
```

This parameter may be used multiple times.

```
Supported = 100rel
Supported = replaces
```

```
OneLineHeaders = <0 or 1>
```

Certain headers occur multiple times, for example, the `Allow` header. By default these headers appear on one line in the SIP message for each instance of the header. Set this parameter to 1 if these headers should appear as a single header with comma separated values.

For Example

```
Allow: INVITE
```

```
Allow: CANCEL
```

becomes

```
Allow: INVITE, CANCEL
```

2.4 TLS configuration

By default, TLS support is disabled in the Aculab SIP service. Use of TLS by the application writer is not a trivial matter; the developer is advised to refer to Appendix D: and to the wealth of information on the public domain about TLS and IP security.

Briefly, the settings in the configuration file relevant to TLS are as follows:

```
UseTLS = <0 or 1>
```

Set the above flag to 1 to enable TLS support in the Aculab SIP service.

```
TLS_ServerCertificate = <absolute path to file>
```

This value should be an absolute path to a file containing the 'server certificate', which this application wishes to use. 'Server certificate' is a slight misnomer in that the contents of the file may be used by the client entity in a transaction; it is named as such since it is mandatory in a server application.

```
TLS_TrustedCertificates = <absolute path to file>
```

The value should be an absolute path to a file containing the certificates, which this application trusts. Typically, these certificates will be the public certificates of the 'Certification Authority' (CA) for the application. However, for test purposes, it is possible to generate these certificates using tools available on the Internet. There may be one or many of these certificates in the file, depending on the number of trust relationships held by the application. If not applicable, the RHS may be blank.

```
TLS_DH512File = <absolute path to file>
```

The value should be an absolute path to a 512-bit Diffie-Hellmann file. Please refer to Appendix D for details.

```
TLS_DH1024File = <absolute path to file>
```

The value should be an absolute path to a 1024-bit Diffie-Hellmann file. Please refer to Appendix D for details.

```
TLS_VerifyPeer = <0 or 1>
```

Setting the 'Verify peer' flag to `1` results in the SIP service enforcing a greater level of verification during the initial TLS handshake.

```
TLS_VerifyDepth = <n>
```

This value must be set using a non-negative integer. It specifies the maximum depth of 'chaining' permitted when verifying a public certificate against a candidate-trusted entity.

```
TLS_PostConnectionCheck = <0 or 1>
```

This value should be set if additional post connection verification routines should be performed. Please refer to Appendix D for details.

```
NoSSLv3 = <0 or 1>
NoTLSv1 = <0 or 1>
NoTLSv1_1 = <0 or 1>
NoTLSv1_2 = <0 or 1>
```

These values should be set if support for the SSL or TLS type is NOT required. If not defined then support is assumed. For example if 'NoSSLv3 = 1' and the others are not defined then support for SSLv3 will be removed and all other types are supported when SIPserv is started. Note that SSLv2 is NOT supported by default.

2.5 Distributed SIP settings

Although the SIP call control application and the SIP service are usually deployed on the same chassis, it is possible to arrange that the application and the SIP service be distributed. Such configuration requires parameterisation of the IPC (inter process communication) mechanism operating between the service and application. An IP port number must be selected by the system administrator and specified in the `sipserv.cfg` file.

`IPCListenPort = <n>`

This value must specify an IP port number. It should be greater than 0 and less than 65535, and should not be already in use by the operating system. It is also recommended that you avoid IP ports that could already be in use by other servers, for example, 0-1023, which are assigned by the Internet Assigned Numbers Authority (IANA), and registered ports 1024 to 49151.

When an "`IPC listen port`", above, is specified then the `sipserv` will use this port to listen for IPC on all potential network interfaces. Should the system administrator wish to restrict IPC to one particular interface, this can be achieved by supplying the field below:

`IPCListenAddress = <ip address>`

This value must be an IP address resident on the host in which the `sipserv` is deployed. Configuration of this field is optional for distribution.

In addition to the `sipserv` settings above, it is also necessary to configure the application to connect to the machine hosting the service. This is achieved by settings in a file (which will need to be created if it does not already exist) called `sipplugin.cfg` which should reside in `$ACULAB_ROOT/cfg` directory. For distributed operation both the settings below are mandatory:

`RemoteIPCAddress = <ip address or FQDN>:<port>`

This value specifies the host or an ip address on that host where the distributed SIP service resides. The port number for IPC should be same as that specified for the service in IPCListenPort.

IPv4 Example:

```
RemoteIPCAddress = 192.168.0.35:45234
```

IPv6 Example:

```
RemoteIPCAddress = [2001:db8::1]:45234
```

Certain considerations must be taken when deploying the distributed Aculab SIP service. Firstly, if a firewall exists between the SIP service and the application then the IPC port selected must be opened up. Secondly, the hardware specification of the machines hosting the service and application chassis must be of similar "`endian-ness`", that is, either both big-endian or both little endian.

`CommandTimeout = <N>`

Modify this parameter to set the round-trip in seconds before the application receives a `-507` error in response to an API call. The default is 16 seconds.

2.6 IPv6

IPv4 addresses are used by default. To allow the SIP service to use IPv6 addresses the following option should be set:

`UseIPv6` – set to 1 to enable IPv6, this is not enabled by default.

Note: if you install the SIP IPv6 package that UseIPv6=1 would be a default setting in the sipserv.cfg, otherwise if the SIP IPv4 is installed this setting 'UseIPv6' will not be present and all IPv6 related data such as the default IPv6 Listen Addresses being set to [::] would be absent.

If IPv6 is chosen an additional file is generated in the cfg area called sip_enable_ipv6.cfg which contains a single entry

enable = 1

if not present it's a simple case to generate such a file in text format.

Likewise, if you wish to disable IPv6 functions having installed the IPv6 package simply setting UseIPv6=0 and removing the sip_enable_ipv6.cfg should have the desired effect.

2.7 Listen Parameters

The listen parameters are fully configurable from the configuration file as follows:

`SIPListenPort`- 5060 by default. This value applies to UDP and TCP unless the `UDPListenPort` or `TCPListenPort` fields override this value.

`EnableUDP` - set to 1 to enable UDP, this is enabled by default.

`UDPListenPort` - 5060 by default. Only one of these may be present, all configured interfaces must listen on the same port.

`UDPListenAddress` - by default the SIP service listens on 0.0.0.0, i.e. all network interfaces. This parameter may be set multiple times to set specific interfaces for the service to listen for UDP.

`EnableTCP` - set to 1 to enable TCP, this is enabled by default

`TCPListenPort` - 5060 by default. Only one of these may be present, all configured interfaces must listen on the same port.

`TCPListenAddress` - by default the SIP service listens on 0.0.0.0, i.e. all network interfaces. This parameter may be set multiple times to set specific interfaces for the service to listen for TCP.

`TLSListenPort` - 5061 by default. Only one of these may be present, all configured interfaces must listen on the same port.

`TLSListenAddress` - by default the SIP service listens on 0.0.0.0, i.e. all network interfaces. This parameter may be set multiple times to set specific interfaces for the service to listen for TLS.

2.7.1 IPv6 Listen Addresses

To listen on all IPv6 interfaces configure the listen addresses for each transport as follows:

```
UDPListenAddress = [::]
TCPListenAddress = [::]
TLSListenAddress = [::]
```

To listen on a specific IPv6 interface (e.g. `2001:db8::1`) configure the listen addresses as follows:

```
UDPListenAddress = 2001:db8::1
TCPListenAddress = 2001:db8::1
TLSListenAddress = 2001:db8::1
```

If the SIP service is listening on an IPv6 address it must also listen on a separate IPv4 address to receive IPv4 traffic.

```
UDPListenAddress = 0.0.0.0
UDPListenAddress = [::]
```

2.8 Overload Monitor

SIP is highly dynamic in its memory usage, the more that is going on, the higher the memory usage. Without the overload monitor configured, this usage is essentially only bounded by hardware. The following parameters aim to restrict memory usage when under extreme load and allow existing calls to continue. There is no guide as to what these values should be, it is necessary to do some testing in order to determine what values are suitable for the system. It may be wise to configure these parameters if the system is unprotected from potentially massive amounts of SIP traffic, however, it is preferable to actually limit the traffic on the network itself.

`OLMMaxMemoryUsage` – The maximum number of bytes that the system is happy to allow for the SIP service. When in an overload situation, the memory will actually fluctuate around this value so it will reach a bit higher but not much. When this is reached for the first time, the service takes a snapshot of the number of current transactions and uses this value in future. An `EV_SIP_MEMORY_LIMIT_EXCEEDED` is raised when this limit is reached. Once this point is reached, all calls are rejected (with a 486 Busy Here) and no transaction is created for that `INVITE`. It is the persistence of the transaction timers that keep the memory usage high and if none are created, the memory usage does not increase.

`OLMMaxSafeMemoryUsage` – This is a lower memory limit that indicates that the system may be hitting excessive traffic. An `EV_SIP_MEMORY_LIMIT_WARNING` is raised when this limit is reached. An `EV_SIP_MEMORY_OK` is raised when the memory usage falls below this level. A transaction count is also taken the first time this limit is reached and used from then on to monitor the memory.

`OLMMemoryInterval` – the duration in milliseconds that the service waits between taking snapshots of the memory usage/transaction count.

`OLMRetryAfter` – If this is set (value in seconds) a Retry-After header containing this value will be added to all the failure responses.

`OLMCheckedAttribute` – Linux systems only. The SIP service will check the VmSize value for the current instance of the sipserv. Set this parameter to the desired field name if a change is required, e.g. VmRSS.

2.9 Miscellaneous global settings

`BracketiseReferTo = n`

There have been two headers in a `REFER` message that have been shown to have interoperability issues with the occasional 3rd party SIP User Agent (UA). These headers are 'Refer-To' and 'Referred-By'. By default, the SIP service will place angle brackets around the values of these headers and, although this is perfectly valid, some UAs do not understand the brackets. Setting this value to 0 will remove these brackets. Otherwise it should be commented out, equal to 1 or simply not present in the configuration file.

`DefaultResponse = xxx`

By default the SIP service will respond to an `INVITE` with a 486 Busy Here code when there are no incoming calls waiting. Set this parameter to a valid SIP response code to change this default response.

`AssumeNonUniqueBranchId` – Matching requests to responses in RFC 3261 is performed by matching the Via header branch parameter in the response to that in the request. In RFC 2543, the process was more complex. The presence of the 'magic cookie' (whose value is '`z9hG4bK`') in the branch parameter indicates that the UA supports RFC 3261 and the appropriate transaction matching is performed. Some UAs have been shown to break this rule.

If `AssumeNonUniqueBranchId` is set, transaction matching will always be performed according to RFC 2543. It is strongly recommended that this is only set under direction from Aculab support.

`DefaultRegExpiryTime` – The default registration duration is 3600 seconds. Set this parameter to another value, in seconds, to change this default.

`IgnoreSessionVersion` – An unchanged session version in the SDP body during a re-`INVITE` indicates that there is no change in the media session parameters. In this case the SIP service will not raise an EV_MEDIA so that it does not need to parse the rest of the SDP. Set this parameter if you want EV_MEDIAs to be raised regardless of the session version.

`ExtractUsernameParameters` – Some endpoints have been seen to put parameters in the user part of a URI. If this parameter is set, these parameters will not appear in sip_detail_parms. This was a workaround for a very specific installation and is highly unlikely to be required.

`TimerT1` – Most of the timers in the SIP service scale with timer T1 which by default is 500ms. It is best to look at RFC 3261 (section 17.1.1.2) for reasons why it might be necessary to change this.

`RaiseOutgoingProceeding` – If this is set an EV_OUTGOING_PROCEEDING will be raised on receipt of 100 Trying.

`RaiseEvDetailsForChallenge` – If an outgoing request is presented with an authentication challenge by the server (either a 401 or a 407) and notification of the SIP response in question has been requested by the application an EV_DETAILS will be raised. This represents a change in behaviour to previous versions of the SIP service. Set this parameter to 0 to restore the original behaviour.

`LocalContactForConnectedAddr` – The service places the remote Contact URI into sip_detail_parms.connected_addr. This represents a change in behaviour to previous versions of the SIP service. Set this parameter to 1 to restore the original behaviour which used the local contact as the connected_addr.

`NoSdpValidation` – Set this parameter to 1 if it is necessary for the application to

validate its own SDP. It is recommended that this value only be set under direction from Aculab support.

`AddDateHeader` – If set to 1 a Date header will be added to all SIP messages.

`UserAgentHeader` – If required, set to the string that is needed in a User-Agent header.

`SuppressPortInRURI` – Set to 1 if it is required that no port parameters appear in the request-URI of any SIP request.

`MatchAuthUserToAorURI` – By default the SIP service will attempt to match any authentication credentials to the user part of the address of record (for `REGISTER` requests) or the user part of the From header (for all other requests) where multiple credentials have been set for a given realm. Set this parameter to 1 if it is required that this match is performed using the name-address of the relevant header (i.e. user@domain) rather than just the user.

`RaiseAcceptForProgress` – By default, `EV_WAIT_FOR_ACCEPT` is not raised after `sip_progress()` has been called unless a reliable provisional response has been sent. Set this parameter to 1 if `EV_WAIT_FOR_ACCEPT` should always be raised.

`NoDisplayNameForURL` – By default the SIP service will create a display name for the To and From URLs. To disable this behaviour set this parameter to 1.

`SIPLicenceListenPort = <N>`
The SIP service licence manager will listen on this port for new licences to be installed. The default value is 13579. Currently the only licensed feature that the SIP service supports is the Dual Redundant SIP Service.

`ErrorOnInvalidCSeq = <N>`
If a SIP request is received with an invalid CSeq number the sipserv will by default respond with a `400` or `500` error code. To disable this behaviour set this parameter to 0.

`RawCauseOnTimeout = <N>`
For an outgoing SIP call, if the `INVITE` does not receive a `200 OK` response the call will timeout. When `xcall_getcause()` is called no raw cause value (i.e. SIP response code) is returned. To force a raw cause value to be returned in this situation set this parameter to the required value which should be a SIP response code.

`DelayStart = <N>`
If required networking resources are not available the SIP service will not start. In these situations the `acuresmgr` will automatically restart the SIP service. In certain circumstances it is possible that the automatic restart will fail. To work around this situation delaying the start up of the SIP service by 3 (or more) seconds will fix this situation. To do this set this parameter to the required number of seconds.

`KeepToHeaderParms = <N>`
`KeepFromHeaderParms = <N>`
The SIP service will remove some parameters from the `To` and `From` headers. To disable this behavious set these parameters to 1.

`RaiseCollisionEvent = <N>`
In the event of a re-`INVITE` collision, an `EV_MEDIA_REJECT_PROPOSAL` or `EV_MEDIA_REJECT_REQUEST_PROPOSAL` will be raised. These events are also raised when the remote end rejects an earlier proposal. To distinguish between a collision and rejection by the remote end a new event `EV_MEDIA_REJECT_COLLISION` can be raised instead. To enable this behaviour set the parameter to 1.

`OutboundThreads = <N>`
Outgoing calls, Out Of Dialog (OOD) requests and Subscriptions can make use of an asynchronous mode which uses separate threads to resolve remote addresses. This

results in less blocking in the associated API calls in the application. This parameter determines the number of threads (default 64). If this is set to 0, the asynchronous mode will not be used.

`AsyncOOD` – By default the SIP service will not use the asynchronous mode for Out Of Dialog requests. To enable this behaviour set this parameter to 1.

`AsyncSubscribe` – By default the SIP service will not use the asynchronous mode for `SUBSCRIBE` requests. To enable this behaviour set this parameter to 1.

`AsyncRegister` – By default the SIP service will not use the asynchronous mode for `REGISTER` requests. To enable this behaviour set this parameter to 1. This enables ipt_add_alias() in asynchronous mode. See ipt_add_alias() in IPT telephony API guide for more details.

`TryDelayStart = <N>`
This parameter has a function similar to `DelayStart`. In Linux builds part of the initialisation process involves a call to `getaddrinfo()` at which point if the returned value is in error a call to `res_init()` may need to be performed and another call to `getaddrinfo()` performed. The value entered into `TryDelayStart` represents a time in seconds in which to perform this action. If it has not performed this action successfully sipserv will fail and exit. This action is necessary in order to establish a Socket Monitor which indicates if proper socket connection is possible. A value of 10 seconds is recommended. If `TryDelayStart` is not included only one call to `getaddrinfo()` is attempted.

`DebugSocketMon` – By default the SIP service will not use this. By including the setting `DebugSocketMon` = 1 in the sipserv.cfg enables the logging of the amount of attempts at acquiring the Socket Monitor should a loss of socket communication occur. The results are passed to a file called sockmon.log.

`RemovePendingTransactions` – By default the SIP service will not use this. If an application should die (or is removed), if `RemovePendingTransactions` = 1 is included in the sipserv.cfg, any pending out-of-dialog transactions that are waiting for a response from the user/application will be removed. May be useful for resolving possible memory leaks. Caution is advised when sipserv is in a multi-application environment.

`TimerC` – TimerC is responsible for the time between an initial INVITE being sent and the sending of a CANCEL request should a response, other than a provisional response, not be received. The default value is set to 3 minutes. TimerC is a value in milliseconds thus by including it you can change this duration, for example, a time of 5 minutes is TimerC=300000.

`DelayTrying` – DelayTrying when set (DelayTrying = 1) disables sipserv from automatically sending the ‘100 Trying’ response. Once disabled the application will be responsible for sending this response using sip_send_invite_response() with sip_code = 100. All other parameters are zero. The default is equivalent to DelayTrying = 0 or sipserv sends ‘100 Trying’ response automatically upon reception of an INVITE.

`OutgoingOptions = 1024` – has a similar effect to setting ACU_NO_HOLD_EVENTS in call_options of sip_openout(). This is to prevent any HOLD events from being raised to conform with IETF RFC 6337. Default is to allow SIP service to raise HOLD events when appropriate based on the simplistic recognition of the ‘a=sendonly’ and ‘a=recvonly’ within the SDP bodies of the offer and answer respectively.

`IncomingOptions = 1024` – has a similar effect to setting ACU_NO_HOLD_EVENTS in call_options of sip_openin().This is to prevent any HOLD events from being raised

to conform with IETF RFC 6337. Default is to allow SIP service to raise HOLD events when appropriate based on the simplistic recognition of the 'a=sendonly' and 'a=recvonly' within the SDP bodies of the offer and answer respectively.

2.10 Media Handler Plugin Settings

The media handler plugin is the component which allows the Generic Call Control API to use the SIP service. To modify any of the settings edit the file (which will need to be created if it does not already exist) called `mhp.cfg` which should reside in `$ACULAB_ROOT/cfg` directory.

There following settings can be applied if necessary:

`TiNGTrace = <N>` - If this is set to a non-zero value a log file called `ACU_MHP_TiNGTrace.log` will be created in the `$ACULAB_ROOT/log` directory. The TiNG documentation describes suitable values for <N>.

`LocalRtpSymmetric` - If this is set to 1 then the TX RTP port will be set to the same port being used for receiving RTP data. This enables allows symmetric which helps when traversing NATs.

`NoRFC4040` – If this flag is set to 1 then no support is offered for RFC 4040.

`PayloadTelephoneEvent, PayloadG726_16, PayloadG726_24, PayloadG726_32, PayloadG726_40, PayloadGSM_EFR, PayloadILBC, PayloadRFC4040, PayloadAMR_NB, PayloadAMR_WB, PayloadEVRC, PayloadEVRC0` – If any of these are set to a number between 96 and 127 then that value will be used instead of the default dynamic codec number. This may be necessary where some vendor equipment requires specific payload numbers dynamic codecs.

3 Interface Definition (APIs)

The following section describes the interface of the library functions and the SIP service. Each function is described in terms of its calling parameters and the values that the function will return. No particular operating system is assumed.

Enhancements to the Aculab API often require extension of the structures used as parameters to Aculab API calls. To eliminate problems associated with this, the following steps must be performed:

```
memset(&structure, 0, sizeof(structure));
structure.size = sizeof(structure);
```

In C and C++ programs, these steps can be replaced with the following macro, defined in `acu_type.h`:

```
INIT_ACU_STRUCT(&structure);
```

SIP feature support uses a set of library function calls provided in addition to the generic call control library.

The SIP function library enables the application to send and receive instructions/information required to support the features specified at the start of this document.

With the exception of those explicitly stated as being **mandatory,** all the field elements of the structures used in the API functions documented below are optional.

The additional library function calls are shown below:

API	Description
`sip_open_port()`	Open a port for use
`sip_openout()`	Open for an outgoing call
`sip_openin()`	Open for an incoming call
`sip_accept()`	Accept the call after an incoming call has been indicated
`sip_details()`	Gather the details of the current call
`sip_free_details()`	Free any memory that may have been dynamically allocated
`sip_incoming_ringing()`	Send the ringing message to the network causing the caller to hear the ring tone
`sip_progress()`	Send call progress information to the network
`sip_send_invite_response()`	Send generic response to an initial `INVITE` with a high degree of control over the content of the response.
`sip_feature_send()`	Acknowledge or reject hold, reconnect or transfer requests
`sip_media_propose()`	Send a new media proposal (a new offer in an offer-answer exchange), to the remote party in a SIP call
`sip_media_accept()`	Accept a media proposal received from the remote party in a SIP call

API	Description
sip_media_request_proposal()	Request the remote party to send a new media offer
sip_media_reject_proposal()	Reject a media proposal received from the remote party
sip_send_request()	Send a mid call SIP request message to the remote party
sip_send_response()	Send a mid call SIP response message to the remote party
sip_set_reason_phrase()	Set the human readable string in a response message
sip_add_answer_challenge_credentials()	Used by the application to pass authentication credentials to the protocol stack
sip_remove_answer_challenge_credentials()	Used in order that an application can remove credentials previously added by sip_add_answer_challenge_credentials().
sip_disconnect()	Disconnect with 3xx response.
sip_recall()	Call an alternative address.
sip_set_tls_private_key_password()	Pass a password for a TLS private key.
sip_load_tls_configuration()	Load a new set of TLS certificates.
sip_set_message_notification()	Declare an interest in out of dialog messages.
sip_send_out_of_dialog_request()	Send an out of dialog request.
sip_send_out_of_dialog_response()	Send an out of dialog response.
sip_read_request()	Collect an out of dialog request.
sip_read_response()	Collect an out of dialog response.
sip_read_timeout()	Collect out of dialog timeout notification.
sip_free_message()	Free memory associated with out of dialog notification.
sip_sub_subscriber()	SUBSCRIBE to a specific event package.
sip_sub_notifier()	Wait for SUBCRIBE requests for a specific event package.
sip_sub_accept()	Accept a SUBSCRIBE request for a specific event package.
sip_sub_notify()	NOTIFY a subscriber of a state change in a specific event package.
sip_sub_cancel()	Cancel an existing subscription or reject a SUBSCRIBE request.
sip_sub_release()	Release the internal resources associated with a particular subscription.

API	Description
`sip_sub_fetch()`	Request an immediate fetch of subscription state.

3.1 sip_open_port() – open a SIP call control port

This routine enables the application to open the SIP call control port in a hardware independent manner. It is not necessary to have previously opened a card for call control with SIP when calling this function. This provides a framework for applications that control media resource acquisition independently of the call control library to be developed. The port obtained by this call can then be used in subsequent functions to open calls, that is, `sip_openout()` and `sip_openin()`.

A port obtained previously using `sip_open_port()` may be closed using `call_close_port()`.

Synopsis

```
ACU_ERR sip_open_port(ACU_PORT_ID* sip_port);              /* OUT */
```

Input parameters

The `sip_open_port()` function takes a pointer, *`sip_port`*, to an `ACU_PORT_ID` variable. On successful execution, this variable is populated with a `port_id` referring to a SIP call control port.

Return values

sip_port

The *`sip_port`* field will contain the port id to be used when making calls with this port.

On successful completion, a value of zero is returned. Otherwise, a negative value will be returned indicating the type of error.

Example usage

```
ACU_PORT_ID sipPort;

ACU_ERR rc = sip_open_port(&sipPort);
if(ERR_NO_ERROR==rc)
{
     SIP_OUT_PARMS outx;
     INIT_ACU_STRUCT(&outx);

     outx.net = sipPort;

     // setup SIP_OUT_PARMS

     rc = sip_openout(&outx);
}
```

3.2 sip_openout() - open for outgoing call

This function allows an application to initiate an outgoing call by sending an `INVITE` message to a remote party. The function registers the outgoing call requirement with the SIP service, which if satisfied with the calling parameters, will return a unique call identifier, the `handle`. The net field supplied to this function must be allocated using `sip_open_port()`.

In the generic call control API the implementation of `call_openout()` for SIP makes many assumptions on behalf of the application in the setup of the outbound `INVITE` message. In the extended API, the application is able to affect the format of the `INVITE` to a greater extent:

- Custom SIP headers can be added to the message
- Custom message bodies can be added to the message
- SDP configuration is now wholly the application's responsibility
- No size restrictions are placed on the amount of the data passed

Additionally the call can now be selected to collect certain types of SIP messages for later presentation to the application.

Outgoing calls may be made over the TCP transport. To effect this the string "`;transport=tcp`" must be appended to the destination URI. For further details, please refer to the URI example below.

Outgoing calls may be made over the TLS transport. To effect this a full SIPS URI must be used for the destination address. For further details, please refer to the URI example below. A brief discussion of the usage of TLS in the Aculab SIP service is given in Appendix D:.

This call `handle` may be used in any appropriate extended SIP API functions and in those generic API functions to which there is not an extended API equivalent. For example, a handle acquired by `sip_openout()` may be supplied to `sip_feature_send()` but not `call_feature_send()`, and additionally to `call_hold()`, `call_disconnect()` and `call_release()` (as the latter functions have no extended API equivalent).

Synopsis

```
ACU_ERR sip_openout(SIP_OUT_PARMS* out_parms);

  typedef struct sip_out_parms
  {
    ACU_ULONG                 size;
    ACU_CALL_HANDLE           handle;                          /* OUT */
    ACU_PORT_ID               net;                             /* IN */
    ACU_INT                   cnf;                             /* IN */
    char*                     destination_addr;                /* IN */
    char*                     originating_addr;                /* IN */
    char*                     contact_address;                 /* IN */
    char*                     destination_display_name;        /* IN */
    char*                     originating_display_name;        /* IN */
    ACU_ACT                   app_context_token;               /* IN */
    ACU_EVENT_QUEUE           queue_id;                        /* IN */
    ACU_UINT                  request_notification_mask;       /* IN */
    ACU_UINT                  response_notification_mask;      /* IN */
     ACU_UINT                 enable_midcall_response_mask;    /* IN */
    char*                     request_uri;                     /* IN */
    char*                     custom_headers;                  /* IN */
```

```
    ACU_RAW_MESSAGE_BODY*      message_bodies;                /* IN */
    ACU_MEDIA_OFFER_ANSWER*    media_offer_answer;            /* IN */
    ACU_UINT                   call_options;                  /* IN */
    char                       transport_type;                /* IN */
    char*                      callid;                        /* IN */
  } SIP_OUT_PARMS;
```

Input parameters

The `sip_openout()` function takes a pointer, `out_parms`, to a structure `SIP_OUT_PARMS`.

`SIP_OUT_PARMS` has similar format as the `OUT_XPARMS` structure described in the Generic Call Control specifications, but with some additional parameters.

net

Specifies the `port_id` referring to the protocol stack on which the call is to be made, as returned from `sip_open_port()`. This is a **mandatory** field.

cnf

The `CNF_TSVIRTUAL` option for the *`cnf`* flag is supported. When this option is set a virtual outgoing call is returned by the function. The URI supplied, as the *`destination_addr`* should refer to the target of a transfer operation. The call can be subsequently used as *`handlec`* in `call_transfer`. A call created with this option emits no signalling and raises only the `EV_IDLE` event, after which the call may be released.

destination_addr

This is a pointer to an ASCII null terminated string. This field should be a URI style of address.

A URI contains the name of the scheme being used (`<scheme>`) followed by a colon and then a string (the `<scheme-specific-part>`) whose interpretation depends on the scheme.

It must be noted that if the address supplied does not conform exactly to the URI format, for example if the `<scheme>: section missing`, the SIP service will prepend the sip: scheme specifier to the supplied address and attempt to place the call using that. The exception is when an IPv6 address forms part of the address In this case if the supplied address does not form a valid URI an error (ERR_PARM) will be returned.

The destination address will be used to construct the SIP or TEL To: header and the Request-URI for the initial `INVITE` message.

Any valid URI parameters may be supplied in the `destination_addr` field. For example, here are some valid SIP URIs:

```
sip:alice@atlanta.com
sip:alice@atlanta.com:5061
sip:alice:secretword@atlanta.com;transport=tcp
sip:+1-212-555-1212:1234@gateway.com;user=phone
sip:alice@192.0.2.4
sip:alice@[2001:db8::1]
sip:alice@[2001:db8::1]:5070
sip:atlanta.com;method=REGISTER?to=alice%40atlanta.com
sip:alice;day=tuesday@atlanta.com
```

e.g.

```
outx.destination_addr = "sip:alice@atlanta.com";
outx.destination_addr = "<sip:alice@atlanta.com;transport=tcp>";
```

SIPS URIs:

```
sips:alice@wonderland.com
sips:alice@wonderland.com:5062
```

```
sips:alice@192.0.2.4
```

e.g.

```
outx.destination_addr = "sips:alice@wonderland.com";
```

TEL URIs:

```
tel:+1-201-555-0123
tel:7042;phone-context=example.com
tel:863-1234;phone-context=+1-914-555
```

e.g.

```
outx.destination_addr = "tel:7042;phone-context=example.com";
```

NOTE

<>s are required to encapsulate the URI along with any URI parameters. Specific To header parameters do not go inside the <>.

```
<user@1.2.3.4;uri-parm=inside>;header-parm=outside
```

NOTE

The SIP service relaxes the URI-style restriction for certain destination_addr settings; when a purely numeric string is supplied e.g. 1234, the service converts this to sip:1234@proxy, where proxy is the address of the pre-configured SIP proxy. This is supplied as an optimisation for gateway applications. If no proxy is configured the SIP service will use the default SIP server address sip.mcast.net (224.0.1.75 IPv4).

NOTE

Currently SIP URIs and Tel URIs are supported by the SIP service. (See section Appendix E for additional information on Tel URI usage.)

The usage of this field is **mandatory**.

originating_addr

This is a pointer to an ASCII null terminated string. This provides for *originating_addr* to be specified on a per call basis.

The URI addressing style should be used. A URI contains the name of the scheme being used (<scheme>) followed by a colon and then a string (the <scheme-specific-part>) whose interpretation depends on the scheme. This string will be used to construct the SIP or TEL From: header.

If the address supplied does not conform exactly to the URI format, for example the <scheme>: section missing, the SIP service will try to construct a valid URI.

The specification of the *originating_addr* is optional, a NULL pointer may be passed, in this case the service will use "sip:<host address>" for this field.

For example:

```
outx.originating_addr = "sip:bob@biloxi.com";
outx.originating_addr = 0;
```

TEL URIs:

```
tel:+1-201-555-0123
tel:+1-212-555-3141;ext=456
```

e.g.

```
outx.originating_addr = "tel:+1-212-555-3141;ext=456";
```

contact_address

Used to build a non-default contact header, this is useful if the application is running on a multi-homed machine and wishes a particular IP address be used in the contact. For a chassis containing only one NIC card, this field may be left blank. It will be a null terminated ASCII string in URI address format.

For example:

```
outx.contact_address = "sip:user@192.168.0.9";
outx.contact_address = 0;
```

destination_display_name

Used to create the optional display name which when combined with the URI forms the To header. This must be supplied in null terminated ASCII text format.

For example:

```
outx.destination_display_name = "Alice";
outx.destination_display_name = 0;
```

originating_display_name

Used to create the optional display name which when combined with the URI forms the From header. This must be supplied in null terminated ASCII text format.

For example:

```
outx.originating_display_name = "Bob";
outx.originating_display_name = 0;
```

app_context_token

This is a user defined value that will be associated with the handle.

queue_id

The unique event queue identity as returned by `acu_allocate_event_queue()` when creating a queue.

request_notification_mask

This field permits an application to specify, for a given call, which SIP messages (requests) it wishes to be notified on the receipt of. On receipt of a relevant SIP message, a call that has specified this setting, experiences an `EV_DETAILS` event and a subsequent `sip_details()` will collect the entire message. By default, no additional notification is given for inbound SIP messages, other than that inherent with the API's generic call control event raising model. Below is an enumeration listing the SIP requests in which it is possible to receive notification on the receipt thereof.

```
typedef enum acu_sip_message_notification_masks
{
   ACU_SIP_INITIAL_INVITE_NOTIFICATION        = 0x00000001, (see note 1)
   ACU_SIP_REINVITE_NOTIFICATION              = 0x00000002, (see note 1)
   ACU_SIP_TRANSFER_INVITE_NOTIFICATION       = 0x00000004, (see note 1)
   ACU_SIP_INFO_NOTIFICATION                  = 0x00000008,
   ACU_SIP_NOTIFY_NOTIFICATION                = 0x00000010,
   ACU_SIP_REGISTER_NOTIFICATION              = 0x00000020, (see note 2)
   ACU_SIP_SUBSCRIBE_NOTIFICATION             = 0x00000040, (see note 2)
   ACU_SIP_OPTIONS_NOTIFICATION               = 0x00000080,
   ACU_SIP_BYE_NOTIFICATION                   = 0x00000100,
   ACU_SIP_MESSAGE_NOTIFICATION               = 0x00000200,
   ACU_SIP_UPDATE_NOTIFICATION                = 0x00000400,
```

```
   ACU_SIP_PRACK_NOTIFICATION                = 0x00000800, (see note 1)
   ACU_SIP_REFER_NOTIFICATION                = 0x00001000, (see note 1)
   ACU_SIP_INITIAL_ACK_NOTIFICATION          = 0x00002000, (see note 1)
   ACU_SIP_REINVITE_ACK_NOTIFICATION         = 0x00004000, (see note 1)
   ACU_SIP_TRANSFER_ACK_NOTIFICATION         = 0x00008000, (see note 1)
} ACU_SIP_MESSAGE_NOTIFICATION_MASKS;
```

In order to receive notification for a particular message, an application needs to set this field to the value of one of the bits in the above enumeration. Notification of more than one message class may be accomplished by bitwise `OR`-ing the required bits.

NOTE

1. These message types are not valid for use with `enable_response_mask`.
2. These message types are only valid for use with the out of dialog API.

For example:

```
outx.request_notification_mask = ACU_SIP_INFO_NOTIFICATION;
```

response_notification_mask

This field is the same as the above except that it effects whether or not a call receives notification of particular SIP response being received. Rules for assigning this field are the same as for *`request_notification_mask`*.

enable_midcall_response_mask;

This field permits an application to specify, for a given call, which SIP messages it wishes to formulate its own responses. See note above for which masks are valid for use with this field. If set for a given message type, the SIP service will only respond after the application calls `sip_send_response()`. The exception to this is `ACU_SIP_BYE_NOTIFICATION` which is a special case as it affects the termination of the call. If a `BYE` message is received and the `enable_midcall_response_mask` has the `ACU_SIP_BYE_NOTIFICATION` bit set then `EV_REMOTE_DISCONNECT` will be raised. The correct way to respond to this is by using `sip_disconnect()`. An `EV_IDLE` will be raised when the response is sent. If no response is sent the call will remain in this state, however the remote end will time out.

request_uri

This optional field allows the user to specify a Request-URI that is distinct from the To header.

custom_headers

The application may supply additional SIP headers to be added to the outgoing message. Note if multiple headers are being appended then `\r\n` should be used to delimit each header.

For example:

```
outx.custom_headers = "Subject: The meeting";
outx.custom_headers = "Subject: 10acb7899\r\nServer: VoIP server";
```

This field must be null terminated.

message_bodies

Message bodies to be added to the outgoing message can be specified here. See section 4 for further details of the setup of this structure `ACU_RAW_MESSAGE_BODY`.

For example:

```
ACU_RAW_MESSAGE_BODY arm;
memset(&arm, 0, sizeof(ACU_RAW_MESSAGE_BODY));
```

```
// setup the message body structure - see sip_send_request() for further
details

outx.message_bodies = &arm;
```

media_offer_answer

This field is a pointer to an ACU_MEDIA_OFFER_ANSWER specifying the settings relevant to starting a media session with the caller. If the field is set to zero then an INVITE with no SDP is sent out, this may be useful for certain modes of third party call control. Please refer to the documentation on the ACU_MEDIA_OFFER_ANSWER structure for more information on this structure.

For example:

```
ACU_MEDIA_OFFER_ANSWER mo;
memset(&mo, 0, sizeof(ACU_MEDIA_OFFER_ANSWER));

// setup the offer answer structure - see sip_media_propose for further
details

outx.media_offer_answer = &mo;
```

call_options

This is a field of bits, each of which specify a particular call option. The following enumeration lists the available call options and the bits used to set those options:

```
typedef enum acu_sip_call_options
{
  ACU_RAISE_MEDIA_EVENT_FOR_HOLD        =0x00000001,
  ACU_RAISE_HOLD_EVENT_FOR_C_EQUAL_ZERO =0x00000002,
  ACU_USE_C_EQUAL_ZERO_FOR_HOLD         =0x00000004,
  ACU_DISABLE_EARLY_MEDIA               =0x00000008,
  ACU_RAISE_HOLD_EVENT_FOR_INACTIVE     =0x00000010,
  ACU_USE_INACTIVE_FOR_HOLD             =0x00000020,
  ACU_DISABLE_TRANSFER_ON_REFER         =0x00000040,
  ACU_USE_MEDIA_DESCRIPTION_FOR_SESSION =0x00000080,
  ACU_DO_NOT_USE_PROXY_IF_SET           =0x00000100,
  ACU_USE_ASYNC_MODE                    =0x00000200,
  ACU_NO_HOLD_EVENTS                    =0x00000400,
  ACU_RAISE_FORKED_EVENT                =0x00000800
 } ACU_SIP_CALL_OPTIONS;
```

If a given bit is not set then the default behaviour is adopted for this option.

Below is a description of the options currently available in the SIP service:

- ACU_RAISE_MEDIA_EVENT_FOR_HOLD
  This option controls the SIP service's response to a hold or reconnect request made by the 'far end'. By default an extended hold or reconnect event is raised as appropriate and no media description information is presented to the application. This may be suitable for some applications, however others may desire to inspect the SDP body relating this request. By setting this option the application will receive an EV_MEDIA_PROPOSE event on receipt of such a request and the media will be available to the sip_details() function call.

- ACU_RAISE_HOLD_EVENT_FOR_C_EQUAL_ZERO
  This options controls the SIP service's response to the receipt of a re-INVITE with new SDP, whose connection address is 0.0.0.0. Previously such a string signified a hold request, however this means of request is now deprecated and has subtly different semantics in third party call control scenarios. By default, receipt of such a re-INVITE raises an EV_MEDIA_PROPOSE event. However, as some handsets still implement hold in this way, setting this option causes the re-INVITE to generate an extended hold request event.

- `ACU_USE_C_EQUAL_ZERO_FOR_HOLD`
  This option affects whether or not the SIP service uses a deprecated mechanism, `c=0.0.0.0`, to implement call hold. By default hold is implemented by using the `a=sendonly` SDP attribute. When this flag is set for a call's options, the SIP service uses `c=0.0.0.0` to implement any `call_hold()` requests made for this call.
- `ACU_DISABLE_EARLY_MEDIA`
  This flag is not supported.
- `ACU_RAISE_HOLD_EVENT_FOR_INACTIVE`
  This option controls the SIP service's response to the receipt of a re-`INVITE` with SDP containing a=inactive. The application should set this option if it is required that an EV_HOLD should be raised in such circumstances.
- `ACU_USE_INACTIVE_FOR_HOLD`
  By default hold is implemented by using the `a=sendonly` SDP attribute. When this option is set the SIP service uses `a=inactive` to implement `call_hold()` requests made for this call.
- `ACU_DISABLE_TRANSFER_ON_REFER`
  This feature allows the application to micromanage a call transfer. By default when a `REFER` is received the SIP service will raise an `EV_EXT_TRANSFER_INFORMATION` event and handle the transfer when `call_feature_send()` is called. With this option turned on the SIP service will (if the application has enabled `ACU_SIP_REFER_NOTIFICATION`) provide the raw `REFER` message to the application. The application must handle the transfer as they see fit using alternative api calls. See RFC 3515 for a description of the `REFER` method.
- `ACU_USE_MEDIA_DESCRIPTION_FOR_SESSION`
  By default it is not possible to for the application to send or receive session level attributes as `ACU_MISCELLANEOUS_MEDIA_ATTRIBUTES`. By setting this option an extra media description is inserted at the beginning of the list of received media descriptions with a 'special' type of `ACU_SESSION`. This media description will contain a session level connection address (if present), miscellaneous attributes (if present) and the `next` pointer will point to the first 'real' media description.
- `ACU_DO_NOT_USE_PROXY_IF_SET`
  This option affects calls when `ipt_set_sip_proxy()` has been used to configure all outbound calls to be routed to a proxy. When this flag is not enabled (which is the default) the initial `INVITE` will be sent to the configured outbound proxy. When this flag is enabled, the initial `INVITE` for an outbound call will be sent to the address provided in the destination address instead of the configured outbound proxy (other factors that affect this is the `request_uri` or if a `Route` header is supplied in `custom_headers`). This allows the application to override the global proxy setting for individual calls.
- `ACU_USE_ASYNC_MODE`
  This option affects outgoing calls. In some cases issues with DNS may lead to significant delays while processing the `sip_openout()` call. This can result in the application being blocked for long periods. When this option is set, the `sip_openout()` call will immediately return successfully, allowing the application to continue. The SIP service will resolve addresses and raise either `EV_WAIT_FOR_OUTGOING` if successful in sending an `INVITE`. If an error occurs an `EV_IDLE` will be raised instead.

- `ACU_NO_HOLD_EVENTS`
  This option prevents the SIP service from raising HOLD events in response to a hold or reconnect request made by the local end. By default an extended hold or reconnect event is raised as appropriate and no media description information is presented to the application. A HOLD state is determined solely by the inclusion of the attribute 'a=sendonly' within the SDP body offer. Based on IETF RFC 6337 the conditions for a HOLD state may be more complicated and it is left up to the application running the SIP service to recognize these conditions based on the contents of both the SDP offer/answer bodies. By setting this option the application will receive an `EV_MEDIA_PROPOSE` event on receipt of such an INVITE request or `EV_MEDIA` upon receiving a response having first issued an INVITE request. The media, in both cases, will be available when `sip_details()` function is called. The application can then determine the condition of the HOLD state if any.
- `ACU_RAISE_FORKED_EVENT`
  This option would allow SIP service to raise `EV_FORKED` event should it detect a forked response during a successful call connection. Typically raised after `EV_CALL_CONNECTED` event.

For example:

```
outx.call_options = ACU_RAISE_HOLD_EVENT_FOR_C_EQUAL_ZERO;
```

transport_type

This field has been reserved for future use.

callid

This optional field allows the user to specify a Call-ID instead of relying on sipserv to auto-generate one. Note it MUST be selected by the UAC as a globally unique identifier over space and time.

Return values

handle

If successful, this will contain a unique (non zero) call identifier, which is used in all successive call related operations on the driver.

On successful completion, a value of zero is returned otherwise a negative value will be returned indicating the type of error.

If the `ACU_USE_ASYNC_MODE` option is used, zero will always be returned. Subsequently receiving the event `EV_WAIT_FOR_OUTGOING` indicates success, whereas receiving the event `EV_IDLE` indicates an unspecified error occurred.

Example usage

```
SIP_OUT_PARMS outx;
INIT_ACU_STRUCT(&outx);

outx.net = sipPort;

// addressing
outx.destination_addr="sip:john@gw.com";
outx.originating_addr="sip:bob@voip-company.com";

outx.destination_display_name="john";
outx.originating_display_name="bob";

ACU_MEDIA_OFFER_ANSWER mo1;
memset(&mo1, 0, sizeof(ACU_MEDIA_OFFER_ANSWER));
```

```
// setup the offer answer - then assign to outx

outx.media_offer_answer=&mo1;

ACU_RAW_MESSAGE_BODY armb1;
memset(&armb1, 0, sizeof(ACU_RAW_MESSAGE_BODY));

// setup any custom message body - if required - then assign to outx

outx.message_bodies=&armb1;

// setup any custom header - if required
outx.custom_headers="Server: New-Company\r\nSubject: ab13-8909";

ACU_ERR rc = sip_openout(&outx);
```

3.3 sip_openin() - open for incoming call

This function allows an application to initiate a wait for an incoming call. The function registers the incoming call requirement with the SIP service. If the service is satisfied with the calling parameters, it will return a unique call identifier, the call handle, for that call. The *net* field supplied to this function must be allocated using sip_open_port().

This extended openin function permits an application to specify a collection of certain SIP message types received by this call for later presentation to the application.

Synopsis

```
ACU_ERR sip_openin(SIP_IN_PARMS* in_parms);

   typedef struct sip_in_parms
   {
     ACU_ULONG             size;
     ACU_CALL_HANDLE       handle;                          /* OUT */
     ACU_PORT_ID           net;                             /* IN */
     ACU_INT               cnf;                             /* IN */
     ACU_EVENT_QUEUE       queue_id;                        /* IN */
     ACU_ACT               app_context_token;               /* IN */
     ACU_UINT              request_notification_mask;       /* IN */
     ACU_UINT              response_notification_mask;      /* IN */
     ACU_UINT              enable_midcall_response_mask;    /* IN */
     ACU_UINT              call_options;                    /* IN */
   } SIP_IN_PARMS;
```

Input parameters

The sip_openin() function takes a pointer, in_parms, to a structure SIP_IN_PARMS. The structure must be initialised before invoking the function.

net

Specifies the *port_id* on the referring to the protocol stack on which the call is to be made, as returned from sip_open_port(). This is a **mandatory** field.

cnf

Reserved for future use.

queue_id

This field must be set to a valid queue. The unique event queue identity as returned by acu_allocate_event_queue() when creating a queue.

app_context_token

The *app_context_token* field is a user-defined token value to be associated with the handle.

request_notification_mask

This field permits an application to specify, for a given call, which SIP messages (requests) it wishes to be notified on the receipt of. On receipt of a relevant SIP message, a call that has specified this setting, experiences an EV_DETAILS event and a subsequent sip_details() will collect the entire message. By default, no additional notification is given for inbound SIP messages, other than that inherent with the API's generic call control event raising model.

Below is an enumeration listing the SIP requests in which it is possible to receive notification on the receipt thereof.

```
typedef enum acu_sip_message_notification_masks
{
   ACU_SIP_INITIAL_INVITE_NOTIFICATION        = 0x00000001, (see note 1)
   ACU_SIP_REINVITE_NOTIFICATION              = 0x00000002, (see note 1)
```

```
  ACU_SIP_TRANSFER_INVITE_NOTIFICATION       = 0x00000004, (see note 1)
  ACU_SIP_INFO_NOTIFICATION                  = 0x00000008,
  ACU_SIP_NOTIFY_NOTIFICATION                = 0x00000010,
  ACU_SIP_REGISTER_NOTIFICATION              = 0x00000020, (see note 2)
  ACU_SIP_SUBSCRIBE_NOTIFICATION             = 0x00000040, (see note 2)
  ACU_SIP_OPTIONS_NOTIFICATION               = 0x00000080,
  ACU_SIP_BYE_NOTIFICATION                   = 0x00000100,
  ACU_SIP_MESSAGE_NOTIFICATION               = 0x00000200,
  ACU_SIP_UPDATE_NOTIFICATION                = 0x00000400,
  ACU_SIP_PRACK_NOTIFICATION                 = 0x00000800, (see note 1)
  ACU_SIP_REFER_NOTIFICATION                 = 0x00001000, (see note 1)
  ACU_SIP_INITIAL_ACK_NOTIFICATION           = 0x00002000, (see note 1)
  ACU_SIP_REINVITE_ACK_NOTIFICATION          = 0x00004000, (see note 1)
  ACU_SIP_TRANSFER_ACK_NOTIFICATION          = 0x00008000, (see note 1)
} ACU_SIP_MESSAGE_NOTIFICATION_MASKS;
```

In order to receive notification for a particular message an application needs to set this field to the value of one of the bits in the above enumeration. Notification of more than one message class may be accomplished by bitwise `OR`-ing the required bits.

NOTE

1.These message types are not valid for use with `enable_response_mask`.
2. These message types are only valid for use with the out of dialog API.

response_notification_mask

This field is the same as the above except that it affects whether or not a call receives notification of a particular SIP response being received. Rules for assigning this field are the same as for *`request_notification_mask`*.

enable_midcall_response_mask;

This field permits an application to specify, for a given call, which SIP messages it wishes to formulate its own responses. See note above for which masks are valid for use with this field. If set for a given message type, the SIP service will only respond after the application calls `sip_send_response()`.The exception to this is `ACU_SIP_BYE_NOTIFICATION` which is a special case as it affects the termination of the call. If a `BYE` message is received and the `enable_midcall_response_mask` has the `ACU_SIP_BYE_NOTIFICATION` bit set then `EV_REMOTE_DISCONNECT` will be raised. The correct way to respond to this is by using `sip_disconnect()`. An `EV_IDLE` will be raised when the response is sent. If no response is sent the call will remain in this state, however the remote end will time out.

call_options

This is a field of bits each of which specify a particular call option. The following enumeration lists the available call options and the bits used to set those options:

```
typedef enum acu_sip_call_options
{
  ACU_RAISE_MEDIA_EVENT_FOR_HOLD        =0x00000001,
  ACU_RAISE_HOLD_EVENT_FOR_C_EQUAL_ZERO =0x00000002,
  ACU_USE_C_EQUAL_ZERO_FOR_HOLD         =0x00000004,
  ACU_DISABLE_EARLY_MEDIA               =0x00000008,
  ACU_RAISE_HOLD_EVENT_FOR_INACTIVE     =0x00000010,
  ACU_USE_INACTIVE_FOR_HOLD             =0x00000020,
  ACU_DISABLE_TRANSFER_ON_REFER         =0x00000040,
  ACU_DO_NOT_USE_PROXY_IF_SET           =0x00000100,
  ACU_NO_HOLD_EVENTS                    =0x00000400
} ACU_SIP_CALL_OPTIONS;
```

If a given bit is not set then the default behaviour is adopted for this option. Below is a description of the options currently available in the SIP service:

- ACU_RAISE_MEDIA_EVENT_FOR_HOLD
  This option controls the SIP service's response to a hold or reconnect request made by the 'far end'. By default an extended hold or reconnect event is raised as appropriate and no media description information is presented to the application. This may be suitable for some applications, however others may desire to inspect the SDP body relating this request. By setting this option the application will receive an EV_MEDIA_PROPOSE event on receipt of such a request and the media will be available to the sip_details() function call.
- ACU_RAISE_HOLD_EVENT_FOR_C_EQUAL_ZERO
  This options controls the SIP service's response to the receipt of a re-INVITE with new SDP, whose connection address is 0.0.0.0. Previously such a string signified a hold request, however this means of request is now deprecated and has subtly different semantics in third party call control scenarios. By default, receipt of such a re-INVITE raises an EV_MEDIA_PROPOSE event. However, as some handsets still implement hold in this way, setting this option causes the re-INVITE to generate an extended hold request event.
- ACU_USE_C_EQUAL_ZERO_FOR_HOLD
  This option affects whether or not the SIP service uses a deprecated mechanism, c=0.0.0.0, to implement call hold. By default hold is implemented by using the a=sendonly SDP attribute. When this flag is set for a call's options, the SIP service uses c=0.0.0.0 to implement any call_hold() requests made for this call.
- ACU_DISABLE_EARLY_MEDIA
  This flag is not supported.
- ACU_RAISE_HOLD_EVENT_FOR_INACTIVE
  This option controls the SIP service's response to the receipt of a re-INVITE with SDP containing a=inactive. The application should set this option if it is required that an EV_HOLD should be raised in such circumstances.
- ACU_USE_INACTIVE_FOR_HOLD
  By default hold is implemented by using the a=sendonly SDP attribute. When this option is set the SIP service uses a=inactive to implement call_hold() requests made for this call.
- ACU_DISABLE_TRANSFER_ON_REFER
  This feature allows the application to micromanage a call transfer. By default when a REFER is received the SIP service will raise an EV_EXT_TRANSFER_INFORMATION event and handle the transfer when call_feature_send() is called. With this option turned on the SIP service will (if the application has enabled ACU_SIP_REFER_NOTIFICATION) provide the raw REFER message to the application. The application must handle the transfer as they see fit using alternative api calls. See RFC 3515 for a description of the REFER method.
- ACU_USE_MEDIA_DESCRIPTION_FOR_SESSION
  By default it is not possible to for the application to send or receive session level attributes as ACU_MISCELLANEOUS_MEDIA_ATTRIBUTES. By setting this option an extra media description is inserted at the beginning of the list of received media descriptions with a type of ACU_SESSION. This media descriptions will contain a session level connection address (if present), miscellaneous attributes (if present) and the next pointer will point to the first 'real' media description.
- ACU_DO_NOT_USE_PROXY_IF_SET
  This flag has no effect for an incoming call.

- `ACU_NO_HOLD_EVENTS`
  This option prevents the SIP service from raising HOLD events in response to a hold or reconnect request made by the remote end. By default an extended hold or reconnect event is raised as appropriate and no media description information is presented to the application. A HOLD state is determined solely by the inclusion of the attribute 'a=recvonly' within the SDP body answer. Based on IETF RFC 6337 the conditions for a HOLD state may be more complicated and it is left up to the application running the SIP service to recognize these conditions based on the contents of both the SDP offer/answer bodies. By setting this option the application will receive an `EV_MEDIA_PROPOSE` event on receipt of such an INVITE request or `EV_MEDIA` event upon receiving a response having first issued an INVITE request. The media, in both cases, will be available when `sip_details()` function is called. The application can then determine the condition of the HOLD state if any.

For example:

```
inx.call_options = ACU_RAISE_HOLD_EVENT_FOR_C_EQUAL_ZERO;
```

Return values

handle

If successful, the value in the `handle` field will contain a unique call identifier. This value must be used for all subsequent operations relating to this call. The call handle supplied by the driver will always be non-zero. On successful completion, a value of zero is returned; otherwise, a negative value will be returned indicating the type of error.

Example usage

```
SIP_IN_PARMS inx;
INIT_ACU_STRUCT(&inx);

inx.net = sipPort;
inx.request_notification_mask = ACU_SIP_INFO_NOTIFICATION;

ACU_ERR rc = sip_openin(&inx);
```

3.4 sip_accept() - accept incoming call

This function may be used to accept the call after an incoming call has been indicated. In SIP this is achieved by replying to a received INVITE with a 200OK response.

In the generic API, the implementation of call_accept() for SIP builds the SDP body of the 200OK. In the extended API, the application is responsible for configuring the SDP.

This function should be used once the application has processed the remote (inviting) party's SDP, if any, which has been collected by calling sip_details() subsequent to the EV_INCOMING_CALL_DET. However, the invocation of sip_accept() may be delayed until after sip_incoming_ringing() or sip_progress() has been called.

With the exception of handle, *media_offer_answer* is the only mandatory field in sip_accept(). The way in which this field is configured depends on the SDP received from the remote party. The payloads specified should include a subset of those offered by the remote party. This ensures that a media session can occur between the two ends of the call. If the remote party sent no SDP in the INVITE then the accepting call is at liberty to offer an initial set of payloads.

Synopsis

```
ACU_ERR sip_accept(SIP_ACCEPT_PARMS* accept_parms);

  typedef struct sip_accept_parms
  {
    ACU_ULONG                    size;
    ACU_CALL_HANDLE              handle;                /* IN */
    char*                        contact_address;       /* IN */
    char*                        custom_headers; // reserved for future
    ACU_RAW_MESSAGE_BODY*        message_bodies; // reserved for future
    ACU_MEDIA_OFFER_ANSWER       media_offer_answer;    /* IN */
  } SIP_ACCEPT_PARMS;
```

Input parameters

The sip_accept() function takes a pointer, *accept_parms*, to a structure SIP_ACCEPT_PARMS. The structure must be initialised before invoking the function.

handle

Identifies the call to be accepted. This field is **mandatory**.

contact_address

Used to build a non-default contact header, this is useful if the application is running on a multi-homed machine and wishes a particular IP address be used in the contact. For chassis containing only one NIC card this field may be left blank. It will be in null terminated URI address format. Refer to the sip_openout() section for more details on the URI format.

custom_headers

This field has been reserved for future use.

message_bodies

This field has been reserved for future use.

media_offer_answer

This field is an instance of an ACU_MEDIA_OFFER_ANSWER structure specifying the local media settings relevant to this party. These settings may specify either a media 'answer' or an 'offer' depending on the presence or absence of SDP in the received INVITE respectively. Please refer to section 4.6 for further information on this structure. This field is **mandatory**.

Return values

On successful completion, a value of zero is returned; otherwise, a negative value will be returned indicating the type of error.

Example usage

```
SIP_ACCEPT_PARMS accx;
INIT_ACU_STRUCT(&accx);

accx.handle = hndl;

ACU_MEDIA_OFFER_ANSWER mo1;
memset(&mo1, 0, sizeof(ACU_MEDIA_OFFER_ANSWER));

// setup the offer answer - then assign to accx

accx.media_offer_answer=&mo1;

ACU_ERR rc = sip_accept(&accx);
```

3.5 sip_details() - get call details

This function is used to gather the details of the current call, either incoming or outgoing, connected through the SIP service. This routine must be called upon the receipt of the following events:

- EV_INCOMING_CALL_DET
  An incoming call has received a suitable INVITE message. The address fields will hold addressing information relevant to the INVITE received, the *destination_addr* and the *originating_addr* will hold URIs from the To: header and From: headers respectively. If the INVITE received carried an SDP body then the *media_offer_answer* will be a valid pointer to a structure holding this SDP.
- EV_DETAILS
  It is possible to request notification on the receipt of particular types of SIP message in the invocation of sip_openout() and sip_openin(). When a message of the previously specified type arrives, EV_DETAILS is raised. Calling *sip_details* at this point presents the entire message to the application in the structure pointed to by *sip_message*. Please refer to Appendix C for more details on the collection of raw SIP messages.
- EV_MEDIA
  When the media session between two parties has changed insofar as local and remote SDP have been negotiated, either initially or in a mid-call transaction, this event is raised. Calling sip_details() at this point presents a pointer to an ACU_MEDIA_SESSION structure. This structure contains three elements, sent SDP, received SDP and a flag indicating which of these was the answer in the offer-answer exchange – the transaction in which the SDP negotiation occurred. It is worth noting for a simple SIP call, that is one with a single audio stream, the first payload specified in the answer SDP is the payload to be used, initially, for both parties to send/receive media with.
- EV_MEDIA_PROPOSE
  This event is raised when a new media offer, that is a SIP message with SDP contained therein, has been received. (An exception to this statement being that of the classic inbound call setup, in which EV_INCOMING_CALL_DET will flag the presence of this offer.) Calling sip_details() at this point presents a pointer to an ACU_MEDIA_OFFER_ANSWER structure. This structure holds information relating to the SDP proposed by the other party.
- EV_EXTENDED/EXT_TRANSFER_INFORMATION
  The 'being transferred' and the 'transferred to' party in a transfer scenario receive this extended event on receipt of the first message relating to the transfer. For the 'being transferred' party this is a REFER message, whereas for the 'transferred to' party this is an INVITE (with Replaces). On receipt of this extended event the application must call sip_details(), the flag *transfer_refer_received* will be set in the first case and *media_offer_answer* will be populated in the second case. Please refer to sip_feature_send() for details on how to further proceed in these occurrences.
- EV_EXTENDED/EXT_DIVERSION
  This extended event will be raised on receipt of any 3xx response to an initial INVITE. On receipt of this extended event the application must call sip_details() where *redirect_info* will be populated.

NOTE

Once **sip_details** has been called and the information collected and processed, it is essential that **sip_free_details()** is called using the same **SIP_DETAIL_PARMS*** to free any memory dynamically allocated by **sip_details**. Failure to do so will result in a memory leak.

Synopsis

```
ACU_ERR sip_details(SIP_DETAIL_PARMS* detail_parms);

typedef struct sip_detail_parms
{
  ACU_ULONG                  size;
  ACU_CALL_HANDLE            handle;                                  /* IN */
  ACU_INT                    valid;                                   /* OUT */
  ACU_INT                    calltype;                                /* OUT */
  char*                      destination_addr;                        /* OUT */
  char*                      originating_addr;                        /* OUT */
  char*                      connected_addr;                          /* OUT */
  char*                      destination_display_name;                /* OUT */
  char*                      originating_display_name;                /* OUT */
  ACU_ACT                    app_context_token;                       /* OUT*/
  char                       reliable_provisional_response_supported; /* OUT */
  ACU_SIP_MESSAGE*           sip_message;                             /* OUT*/
  ACU_MEDIA_OFFER_ANSWER*    media_offer_answer;                      /* OUT */
  ACU_MEDIA_SESSION*         media_session;                           /* OUT */
  char                       transfer_refer_received;                 /* OUT */
  char                       transport_type;                          /* OUT */
  ACU_REDIRECT_INFO*         redirect_info;                           /* OUT */
  ACU_SUBSCRIPTION_INFO*     subscription_info;                       /* OUT */
} SIP_DETAIL_PARMS;
```

Input parameters

The `sip_details()` function takes a pointer, *details_parms*, to a structure `SIP_DETAIL_PARMS`. Before invocation, the structure must be cleared using `INIT_ACU_STRUCT` and the handle field assigned with the call under inspection.

handle

The *handle* field is used to identify the call that is to be examined. This field is **mandatory**.

Return values

valid

Is a Boolean value, which indicates whether the details returned are valid or not.

0	details invalid – indicates that there is no valid information in the structure
1	details valid – indicates that there is some valid information in the structure

With SIP the details are valid if there is addressing information for the call, that is, an initial `INVITE` has been sent/received.

calltype

Will indicate the direction of the call in progress and will have the values:

`OUTGOING:`	for outgoing call
`INCOMING:`	for incoming call

originating_addr and ***destination_addr***

Will contain the URIs extracted from the `From`: and `To:` headers of the call setup `INVITE` received/sent.

connected_addr

Will contain the URI of the remote `Contact:` header used in this call.

destination_display_name
The destination display name in ASCII text format, typically the destination parties name.

originating_display_name
The originating display name, in ASCII text format, typically the originating parties name.

app_context_token
The *`app_context_token`* field contains the value that was associated with the handle when the call was opened using `sip_openin()` or `sip_openout()`.

reliable_provisional_response_supported
For an incoming call, if the `INVITE` message from the caller suggests that the caller supports reliable provisional response, (presence of '`Supported: 100rel`'), this flag will be set.

sip_message
This field is a pointer to an `ACU_SIP_MESSAGE` structure. This will be populated by the `sip_details()` function if the call has queued a received SIP message as indicated by the `EV_DETAILS` event. Please refer to section 4.8 for further information on this field.

media_offer_answer
This field is a pointer to an `ACU_MEDIA_OFFER_ANSWER` structure. This field holds a media offer received from a remote party. This occurrence is flagged to the application by `EV_INCOMING_CALL_DET, EV_MEDIA_PROPOSE` or `EV_EXT_TRANSFER_INFORMATION`. Please refer to section 4.6 for further information on this field.

media_session
This field points to an `ACU_MEDIA_SESSION` structure. This is populated by `sip_details()` subsequent to receipt of the `EV_MEDIA` event. An application should use this information to affect how to control media streams relevant to the call. Please refer to section 4.7 for further information on this field.

transfer_refer_received
This flag is used in order that a party in a call transfer scenario can get an indication of which role in the transfer it is playing and how to proceed next. This flag will be set if the party is the 'being transferred' party and in which case a `REFER` was received, otherwise this implies that the party is the 'transferred to' entity: in which case SDP may or may not be present in the `media_offer_answer`.

transport_type
This field has been reserved for future use.

redirect_info
This field points to an `ACU_REDIRECT_INFO` structure. This is populated by `sip_details()` subsequent to receipt of the `EV_EXTENDED_DIVERSION` event. An application should use this information to supply the input parameters for `sip_recall()`. Please refer to section 4.10 for further information on this field.

subscription_info
This field points to an `ACU_SUBSCRIPTION_INFO` structure. This is populated by `sip_details()` when subscription information is available. For more information on

when subscription events can occur and when to use `sip_details()` see Appendix A

Return values

On successful completion, a value of zero is returned; otherwise, a negative value will be returned indicating the type of error.

3.6 sip_free_details() - return memory allocated by sip_details()

This function is used to free any memory, which may have been dynamically allocated in a call to `sip_details()`. It is essential that this routine is called on any `SIP_DETAIL_PARMS*` received by `sip_details()` once the information has been processed.

NOTE

As the memory being freed is associated with the call handle, **`sip_free_details()`** MUST be called before **`call_release`**.

NOTE

Any memory referred to by the **`SIP_DETAIL_PARMS*`** will be invalidated by this call.

Synopsis

```
ACU_ERR sip_free_details(SIP_DETAIL_PARMS* detail_parms);          /* IN */
```

Input parameters

The `sip_free_details()` function takes a pointer, *`detail_parms`*, to a structure `SIP_DETAIL_PARMS`. The structure has previously been populated by a successful call to `sip_details()`.

Return values

On successful completion, a value of zero is returned; otherwise, a negative value will be returned indicating the type of error.

Example usage

```
SIP_DETAIL_PARMS detx;
INIT_ACU_STRUCT(&detx)

detx.handle = hndl;

ACU_ERR rc = sip_details(&detx);

if(ERR_NO_ERROR==rc)
{
     // process the details structure

     sip_free_details(&detx);
}
```

3.7 sip_incoming_ringing() – send incoming ringing

This function may be used to send a 180 Ringing message to the remote party causing the caller to hear the ring tone. This function may be used after an incoming call has been detected but before the call has been accepted.

Synopsis

```
ACU_ERR sip_incoming_ringing(SIP_INCOMING_RINGING_PARMS* ringing_parms);

   typedef struct sip_incoming_ringing_parms
   {
     ACU_ULONG                    size;
     ACU_CALL_HANDLE              handle;                            /* IN */
     ACU_INT                      send_early_media;                  /* IN */
     ACU_INT                      use_183_response_for_early_media;  /* IN */
     ACU_INT                      send_reliable_provisional_response; /* IN */
     char*                        contact_address;                   /* IN */
     ACU_MEDIA_OFFER_ANSWER*      media_offer_answer;                /* IN */
   } SIP_INCOMING_RINGING_PARMS;
```

Input parameters

The `sip_incoming_ringing()` function takes a pointer, *`ringing_parms`*, to a structure `SIP_INCOMING_RINGING_PARMS`.

The use of `sip_incoming_ringing()` with no parameters set within the `SIP_INCOMING_RINGING_PARMS` structure, other than the handle, will result in a 180 message with no media offer being sent to the caller. However, this function may be used to initiate early media in the call if the *`send_early_media`* flag is set. In this case, the *`media_offer_answer`* field must be suitably configured with the required SDP.

handle

The *`handle`* field identifies the call that will send the incoming ringing message. This field is **mandatory**.

send_early_media

This flag has no effect. SDP is sent if provided in `media_offer_answer`.

use_183_response_for_early_media

Setting this will make `sip_incoming_ringing()` send a 183 message instead of a 180. This to all intents and purposes will make the call look like `call_progress()`. It is provided to support possible interoperability issues.

send_reliable_provisional_response

By default `18*` messages are sent unreliably, that is, the sender does not retransmit the message in absence of an acknowledge from the far end. Setting this flag to 1, forces them to be sent reliably and the sender resends until a `PRACK` message is received from the remote party. `ERR_PARM` is returned if the caller does not support the protocol extension (100rel) required for reliable provisional responses.

NOTE

If a reliable provisional response has been sent the application should not call sip_accept until it has received an EV_WAIT_FOR_ACCEPT as this will not be raised until the PRACK has been received.

contact_address

Used to build a non-default contact header, this being useful if the application is running on a multi-homed machine and wishes a particular IP address be used in the contact. For chassis containing only one NIC card this field may be left blank, (URI address format).

media_offer_answer

This field is a pointer to an `ACU_MEDIA_OFFER_ANSWER` specifying the settings relevant to starting an early media session with the caller. Please refer to section 4.6 for further information on this structure.

Return values

On successful completion, a value of zero is returned; otherwise, a negative value will be returned indicating the type of error.

NOTE

use of the function will result in the **`EV_WAIT_FOR_ACCEPT`** state, whereupon **`sip_accept()`** will connect the call. The application can therefore control the number of ring cadences by delaying the **`sip_accept()`** function.

Example usage

```
SIP_INCOMING_RINGING_PARMS ringx;
INIT_ACU_STRUCT(&ringx);

ringx.handle = hndl;
ringx.send_early_media = 1;

ACU_MEDIA_OFFER_ANSWER mo1;
memset(&mo1, 0, sizeof(ACU_MEDIA_OFFER_ANSWER));

// setup the offer answer - then assign to ringx

ringx.media_offer_answer = &mo1;

ACU_ERR rc = sip_incoming_ringing(&ringx);
```

3.8 sip_progress() - send progress information

This function may be used to send call progress information to the remote party. A 183 Session progress message with an SDP body is sent to the other party.

NOTE

For the SIP protocol use of this routine enables early media announcements.

Synopsis

```
ACU_ERR sip_progress(SIP_PROGRESS_PARMS* progress_parms);

  typedef struct sip_progress_parms
  {
    ACU_ULONG                  size;
    ACU_CALL_HANDLE            handle;                                /* IN */
    ACU_INT                    send_reliable_provisional_response;    /* IN */
    char*                      contact_address;                       /* IN */
    ACU_MEDIA_OFFER_ANSWER*    media_offer_answer;                    /* IN */
  } SIP_PROGRESS_PARMS;
```

Input parameters

The `sip_progress()` function takes a pointer, *progress_parms*, to a structure `SIP_PROGRESS_PARMS`. This function should be invoked in the following way:

handle

The *handle* field identifies the call that will send the progress message. This field is **mandatory**.

send_reliable_provisional_response

By default `18*` messages are sent unreliably, that is, the sender does not retransmit the message in the absence of an acknowledge from the far end. Setting this flag to 1, forces them to be sent reliably and the sender resends until a PRACK message is received. `ERR_PARM` is returned if the caller does not support the protocol extension required for reliable provisional responses.

NOTE

If a reliable provisional response has been sent the application should not call sip_accept until it has received an EV_WAIT_FOR_ACCEPT as this will not be raised until the PRACK has been received.

contact_address

Used to build a non-default contact header, this is useful if the application is running on a multi-homed machine and wishes a particular IP address be used in the contact. For chassis containing only one NIC card, this field may be left blank, (URI address format).

media_offer_answer

This field is a pointer to an `ACU_MEDIA_OFFER_ANSWER` specifying the settings relevant to starting an early media session with the caller. Please refer to section 4.6 for further information on this structure.

Return values

On successful completion, a value of zero is returned; otherwise, a negative value will be returned indicating the type of error.

Example usage

```
SIP_PROGRESS_PARMS progx;
```

```
INIT_ACU_STRUCT(&progx);

progx.handle = hndl;
progx.send_reliable_provisional_response = 1;

ACU_MEDIA_OFFER_ANSWER mo1;
memset(&mo1, 0, sizeof(ACU_MEDIA_OFFER_ANSWER));

// setup the offer answer - then assign to progx

progx.media_offer_answer = &mo1;

ACU_ERR rc = sip_progress(&progx);
```

3.9 sip_send_invite_response() - Send response to initial INVITE

This function may be used to send any response to an initial `INVITE` and may be used in place of the following API calls: sip_incoming_ringing, sip_progress, sip_accept, sip_disconnect and call_disconnect (if the call is in a valid state). It gives the application considerably more control over the content of the message allowing the inclusion of any message body and any additional SIP headers.

Synopsis

```
ACU_ERR sip_send_invite_response(SIP_SEND_INVITE_RESPONSE_PARMS* response_parms);

   typedef struct tSIP_SEND_INVITE_RESPONSE_PARMS
   {
        ACU_ULONG                size;                    /* IN */
        ACU_CALL_HANDLE          handle;                  /* IN */
        ACU_INT                  sip_code;                /* IN */
        ACU_INT                  send_reliably;           /* IN */
        ACU_STRING_LIST*         contact_list;            /* IN */
        char*                     custom_headers;         /* IN */
        ACU_MEDIA_OFFER_ANSWER* media_offer_answer;       /* IN */
        ACU_RAW_MESSAGE_BODY*    message_bodies;          /* IN */
   } SIP_SEND_INVITE_RESPONSE_PARMS;
```

Input parameters

The `sip_send_invite_response()` function takes a pointer, *`response_parms`*, to a structure `SIP_SEND_INVITE_RESPONSE`. This function should be invoked in the following way:

handle

The *`handle`* field identifies the call that will send the response. This field is **mandatory**.

sip_code

This field indicates the response code that will be used in the status line of the message. As this represents standardised response codes `ERR_PARM` will be returned for any value outside of the range 100 < ***sip_code*** < 700. This field is **mandatory**.

send_reliably

By default `1**` messages are sent unreliably, that is, the sender does not retransmit the message in the absence of an acknowledge from the far end. Setting this flag to 1, forces them to be sent reliably and the sender resends until a PRACK message is received. `ERR_PARM` is returned if the caller does not support the protocol extension required for reliable provisional responses. This field is ignored for ***sip_code*** > 199.

NOTE

If a reliable provisional response has been sent the application should not call sip_accept until it has received an EV_WAIT_FOR_ACCEPT as this will not be raised until the PRACK has been received.

contact_list

For 1xx and 2xx responses, this field is used to build a non-default contact header. This is useful if the application is running on a multi-homed machine and wishes a particular IP address be used in the contact. For chassis containing only one NIC card, this field may be left blank, (URI address format). For these responses, only a single contact is valid and `ERR_PARM` will be returned if there are more than one.

For 3xx responses, which redirect the incoming call to an alternative target, any

number of contact addresses may be supplied for the remote user to try.
This field is ignored for ***sip_code*** > 399.

custom_headers
The application may supply additional SIP headers to be added to the outgoing message. Note if multiple headers are being appended then `\r\n` should be used to delimit each header. For example:

```
"Subject: The meeting"
"Subject: 10acb7899\r\nServer: VoIP server"
```

This string must be null terminated.

media_offer_answer
This field is a pointer to an `ACU_MEDIA_OFFER_ANSWER` specifying the settings relevant to starting a media session with the caller. Please refer to section 4.6 for further information on this structure. This field is mandatory for 2xx responses where no reliable. This field is ignored for response codes greater than 299.

message_bodies
Message bodies to be added to the outgoing response can be specified here. See the section 4.1 for further details in the setup of this structure.

Return values

On successful completion, a value of zero is returned; otherwise, a negative value will be returned indicating the type of error.

Example usage

```
SIP_SEND_INVITE_RESPONSE_PARMS responsex;
INIT_ACU_STRUCT(&responsex);

responsex.handle = hndl;
responsex.sip_code = 182;
responsex.send_reliably = 1;
responsex.custom_headers = "Server: Aculab"

ACU_MEDIA_OFFER_ANSWER mo1;
memset(&mo1, 0, sizeof(ACU_MEDIA_OFFER_ANSWER));

// setup the offer answer - then assign to responsex

responsex.media_offer_answer = &mo1;

ACU_ERR rc = sip_send_invite_response(&responsex);
```

3.10 sip_feature_send() - sending feature information

Currently this function is used to acknowledge or reject a supplementary service request received from the remote party and indicated to the local party by the receipt of the following extended events:

- **EV_EXT_HOLD_REQUEST** – The local party has received a re-INVITE message with the following criteria: the a=sendonly attribute and an increment in the SDP session version number (the third field in the SDP o= line).
- **EV_EXT_RECONNECT_REQUEST** – The local party has received an INVITE message with the following criteria: an increment in the SDP session version number (the third field in the SDP o= line) and an implicit or otherwise indication of the sendrecv attribute. In addition the call must have previously received a hold request.
- **EV_EXT_TRANSFER_INFORMATION** - The local party has either received a (transfer) REFER message or an INVITE (with Replaces). Refer to sip_details() to find out how either of these cases is determined.

Synopsis

```
ACU_ERR sip_feature_send(SIP_FEATURE_DETAIL_PARMS* feature_parms);

  typedef struct sip_feature_detail_parms
  {
    ACU_ULONG                 size;
    ACU_CALL_HANDLE           handle;                    /* IN */
    ACU_ULONG                 feature_type;              /* IN */
    union
    {
      struct
      {
        ACU_INT                   command;               /* IN */
      } hold;
      struct
      {
        ACU_INT                   failure_code;          /* IN */
        ACU_MEDIA_OFFER_ANSWER*   media_offer_answer;    /* IN */
      } transfer;
    } feature;
  } SIP_FEATURE_DETAIL_PARMS;
```

Input parameters

The sip_feature_send() function takes a pointer, *feature_parms*, to a structure SIP_FEATURE_DETAIL_PARMS. The structure must be initialised in the following way before invoking the function.

handle

The *handle* field is used to identify the call that will send feature information. This field is **mandatory**.

feature_type

The *feature_type* field should be used to indicate the type of feature. The values this field can take are:

FEATURE_HOLD_RECONNECT

FEATURE_TRANSFER

This field is **mandatory**.

command

This field is relevant for `feature_type == FEATURE_HOLD_RECONNECT` and must contain one of the following values to transmit an appropriate response to the hold or reconnect request.

```
HOLD_ACKNOWLEDGE_CMD
HOLD_REJECT_CMD
RECONNECT_ACKNOWLEDGE_CMD
RECONNECT_REJECT_CMD
```

The SIP protocol stack will reply with either 200OK for acknowledgement of the request or '403 Forbidden' for rejection.

failure_code

This field is relevant for `feature_type==FEATURE_TRANSFER`. If the application wishes to grant this transfer, it should leave this code as zero. Otherwise to decline the transfer it set this code to be valid SIP error code e.g. `403` – forbidden.

media_offer_answer

This field is relevant for `feature_type==FEATURE_TRANSFER`. The application should populate this with a pointer to an `ACU_MEDIA_OFFER_ANSWER` where the settings for the media session between the transferred and the transferred-to are specified.

This field is **mandatory** for the 'transferred to' party and strongly recommended for the 'being transferred' party.

Return values

On successful completion, a value of zero is returned otherwise a negative value will be returned indicating the type of error.

Example usage: Acknowledging/Rejecting a hold request

```
SIP_FEATURE_DETAIL_PARMS fdx;
INIT_ACU_STRUCT(&fdx);

fdx.handle = hndl;
fdx.feature_type = FEATURE_HOLD_RECONNECT;

if(accept_hold)
{
   fdx.feature.hold.command = HOLD_ACKNOWLEDGE_CMD;
}
else
{
   fdx.feature.hold.command = HOLD_REJECT_CMD;
}

ACU_ERR rc = sip_feature_send(&fdx);
```

Example usage: Accepting/Rejecting a transfer (either 'being transferred' or 'transferred party')

```
SIP_FEATURE_DETAIL_PARMS fdx;
INIT_ACU_STRUCT(&fdx);

ACU_MEDIA_OFFER_ANSWER* mo = 0;

fdx.handle = hndl;
fdx.feature_type = FEATURE_TRANSFER;

if(accept_transfer)
{
   mo = malloc(sizeof(ACU_MEDIA_OFFER_ANSWER));

   // configure the media offer answer structure

     fdx.feature.transfer.media_offer_answer = mo;
```

```
}
else
{
     fdx.feature.transfer.failure_code = 486;
}

ACU_ERR rc = sip_feature_send(&fdx);

free(mo);
```

3.11 sip_media_propose() – send a media proposal

Send a new media proposal (a new offer in an offer-answer exchange), to the remote party in a SIP call. This proposal usually takes the form of an SDP body in a re-INVITE, although in certain situations the proposal may be in a `200OK` message. The intended outcome of the proposal is a re-negotiation of a call's media settings. Various call control features in SIP can be implemented by re-negotiating the media session:

- Redirect of media streams to alternative addresses/ports.
- Holding the media stream.
- Adding a new media stream.
- Deleting a media stream.
- Changing a payload setting in an existing stream.

NOTE

The use case to which the first bullet relates is particularly useful in the implementation of third party call control; the controller sends the RTP settings of party A to party B, and vice-versa.

If the proposal is successful, that is, the remote party replied with a suitable 200OK (or ACK), `EV_MEDIA` is raised. If the proposal is declined `EV_MEDIA_REJECT_PROPOSAL` is raised, and it is possible to make a new, altered, proposal.

Refer to RFC 3725 and Aculab's SIP Programming Guide to find more information on this functionality.

Synopsis

```
ACU_ERR sip_media_propose(SIP_MEDIA_PROPOSE_PARMS*
                                                        media_propose_parms);

  typedef struct tSIP_MEDIA_PROPOSE_PARMS
  {
    ACU_ULONG                      size;
    ACU_CALL_HANDLE                handle;                      /* IN */
    ACU_MEDIA_OFFER_ANSWER         media_offer_answer;          /* IN */
    char*                          custom_headers;              /* IN */
  } SIP_MEDIA_PROPOSE_PARMS;
```

Input parameters

The `sip_media_propose()` function takes a pointer *`media_propose_parms`* to a `SIP_MEDIA_PROPOSE_PARMS` structure. The structure must be initialised in the following way before invoking the function.

handle

The *`handle`* field identifies the call that is to send the proposal. This field is **mandatory**.

media_offer_answer

The *`media_offer_answer`* field specifies the media settings applying to this proposal.

This field is **mandatory**. See section 4.6 for further details.

Return values

On successful completion, a value of zero is returned. Otherwise, a negative value will be returned indicating the type of error.

Example Usage

```
SIP_MEDIA_PROPOSE_PARMS propose_parms;
INIT_ACU_STRUCT(&propose_parms);

propose_parms.handle = hndl;
propose_parms.media_offer_answer.connection_address.address =
"10.202.123.2";

ACU_MEDIA_DESCRIPTION media_desc;
memset(&media_desc, 0, sizeof(media_desc));

// setup the ACU_MEDIA_DESCRIPTION

propose_parms.media_offer_answer.media_descriptions = &media_desc;

ACU_ERR rc = sip_media_propose(&propose_parms);
```

custom_headers

The application may supply additional SIP headers to be added to the outgoing message. Note if multiple headers are being appended then `\r\n` should be used to delimit each header.

For example:

```
outx.custom_headers = "P-Asserted-Identity: \"Cullen Jennings\"";

outx.custom_headers = "Subject: 10acb7899\r\nServer: VoIP server";
```

This field must be null terminated.

3.12 sip_media_accept() – accept a media proposal

Accept a media proposal received from the remote party in a SIP call. This involves sending an SDP `answer` body, typically in a `200OK`, but sometimes in an `ACK`. This function must be called in response to an `EV_MEDIA_PROPOSE` event occurring for this call.

This function may be used in combination with `sip_media_propose()` to implement third party call control.

Synopsis

```
ACU_ERR sip_media_accept(SIP_MEDIA_ACCEPT_PARMS* media_accept_parms);
```

```
typedef struct tSIP_MEDIA_ACCEPT_PARMS
{
  ACU_ULONG                    size;
  ACU_CALL_HANDLE              handle;                    /* IN */
  ACU_MEDIA_OFFER_ANSWER       media_offer_answer;        /* IN */
} SIP_MEDIA_ACCEPT_PARMS;
```

Input parameters

The `sip_media_accept()` function takes a pointer *`media_accept_parms`* to a `SIP_MEDIA_ACCEPT_PARMS` structure. The structure must be initialised in the following way before invoking the function.

handle

The *`handle`* field identifies the call that is to accept the proposal. This field is **mandatory**.

media_offer_answer

The *`media_offer_answer`* field specifies the media settings applying to this proposal accept, see section 4.6 for further details. This field is **mandatory**.

Return values

On successful completion, a value of zero is returned. Otherwise, a negative value will be returned indicating the type of error.

Example Usage

Refer to the example usage in the section describing `sip_media_propose()`.

3.13 sip_media_request_proposal() – request a media proposal

Request the remote party to send a new media offer. This is implemented by sending an `INVITE` with no SDP to the party. A positive response from the remote end would be a `200OK` with SDP resulting in an `EV_MEDIA_PROPOSE` being raised. Otherwise, `EV_MEDIA_REJECT_REQUEST_PROPOSAL` is raised if the request was declined.

Synopsis

```
ACU_ERR sip_media_request_proposal
            (SIP_MEDIA_REQUEST_PROPOSAL_PARMS* media_request_proposal_parms);
```

```
typedef struct tSIP_MEDIA_REQUEST_PROPOSAL_PARMS
{
  ACU_ULONG             size;
  ACU_CALL_HANDLE       handle;                /* IN */
  char*                 custom_headers;        /* IN */
} SIP_MEDIA_REQUEST_PROPOSAL_PARMS;
```

Input parameters

The `sip_media_request_proposal()` function takes a pointer *`media_request_proposal_parms`* to a `SIP_MEDIA_REQUEST_PROPOSAL_PARMS` structure. The structure must be initialised in the following way before invoking the function.

handle

The *`handle`* field identifies the call that is to request the proposal. This field is **mandatory**.

custom_headers

The application may supply additional SIP headers to be added to the outgoing message. Note if multiple headers are being appended then `\r\n` should be used to delimit each header.

For example:

```
outx.custom_headers = "P-Asserted-Identity: \"Cullen Jennings\"";
```

```
outx.custom_headers = "Subject: 10acb7899\r\nServer: VoIP server";
```

This field must be null terminated.

Return values

On successful completion, a value of zero is returned. Otherwise, a negative value will be returned indicating the type of error.

Example Usage

```
SIP_MEDIA_REQUEST_PROPOSAL_PARMS req_propx;
INIT_ACU_STRUCT(&req_propx);

req_propx.handle = hndl;

ACU_ERR rc = sip_media_request_proposal(&req_propx);
```

3.14 sip_media_reject_proposal() – reject a media proposal

Reject a media proposal received from the remote party. This may be called in response to an `EV_MEDIA_PROPOSE` or `EV_MEDIA_REQUEST_PROPOSAL` event. If the proposal received was an `INVITE` with SDP, then this reject will be implemented, as `4xx` class of response and the original media session will persist. However, if the proposal, which is being rejected, is in the form of a 200`OK` then, due to SIP protocol rules, the rejection will result in the clearing of the existing call.

Synopsis

```
ACU_ERR sip_media_reject_proposal(SIP_MEDIA_REJECT_PROPOSAL_PARMS*
                                                    media_reject_proposal_parms);
```

```
typedef struct tSIP_MEDIA_REJECT_PROPOSAL_PARMS
{
  ACU_ULONG             size;
  ACU_CALL_HANDLE       handle;                 /* IN */
  union
  {
    ACU_INT               sip_code;             /* IN */
  } protocol_specific;
} SIP_MEDIA_REJECT_PROPOSAL_PARMS;
```

Input parameters

The `sip_media_reject_proposal()` function takes a pointer *`media_reject_proposal_parms`* to a `SIP_MEDIA_REJECT_PROPOSAL_PARMS` structure. The structure must be initialised in the following way before invoking the function.

handle

The *`handle`* field identifies the call that is to reject the proposal. This field is **mandatory**.

sip_code

The optional *`sip_code`* field enables the application to specify the SIP response code set to the proposer. By default '488 Not acceptable here' will be sent. Note that this field has no meaning when the proposal being rejected was in a `200OK`; the call will be cleared by a subsequent `BYE/200OK` transaction.

Return values

On successful completion, a value of zero is returned. Otherwise, a negative value will be returned indicating the type of error.

Example Usage

```
SIP_MEDIA_REJECT_PROPOSAL_PARMS reject_parms;
INIT_ACU_STRUCT(&reject_parms);

reject_parms.handle = hndl;
reject_parms.protocol_specific.sip_code = 403; // "Forbidden"

ACU_ERR err = sip_media_reject_proposal(&reject_parms);
```

3.15 sip_send_request() - send a mid call SIP request

Used to send a mid call SIP request message to the remote party. This routine enables the application to specify the message type, additional SIP headers and message bodies to compose a mid call message to be sent using the signalling path previously setup in a call. Note: under certain circumstances in the case of the `UPDATE` method the `sip_send_request()` can be sent during call set-up before the `INVITE` has received a final response in a pending SIP session (RFC 3311).

Synopsis

```
ACU_ERR sip_send_request(SIP_SEND_REQUEST_PARMS* send_request_parms);
```

```
typedef struct tSIP_SEND_REQUEST_PARMS
{
  ACU_ULONG                 size;
  ACU_CALL_HANDLE           handle;                /* IN */
  unsigned char             message_type;          /* mandatory */
  char*                     custom_headers;        /* IN */
  ACU_RAW_MESSAGE_BODY*     message_bodies;        /* IN */
  ACU_MEDIA_OFFER_ANSWER*   media_offer_answer;    /* IN */

} SIP_SEND_REQUEST_PARMS;
```

Input parameters

The `sip_send_request()` function takes a pointer *`send_request_parms`* to a `SIP_SEND_REQUEST_PARMS` structure. The structure must be initialised in the following way before invoking the function.

handle

The *`handle`* field identifies the call that is to send the SIP message. This field is **mandatory**.

message_type

The SIP message to be sent should be configured. The following enumeration has been supplied to assist the application in setting this field.

```
typedef enum acu_sip_message_type
{
     ACU_SIP_MESSAGE_NULL        =0,
     ACU_SIP_MESSAGE_INFO        =1,
     ACU_SIP_MESSAGE_NOTIFY      =2,
     ACU_SIP_MESSAGE_REGISTER    =3,
     ACU_SIP_MESSAGE_SUBSCRIBE   =4,
     ACU_SIP_MESSAGE_OPTIONS     =5,
     ACU_SIP_MESSAGE_UPDATE      =6,
     ACU_SIP_MESSAGE_MESSAGE     =7
     ACU_SIP_MESSAGE_REFER       =8
 } ACU_SIP_MESSAGE_TYPE;
```

NOTE

`ACU_SIP_MESSAGE_REGISTER` and `ACU_SIP_MESSAGE_SUBSCRIBE` are not applicable to this API call. The same `enum` is used for the out of dialog API.

custom_headers

The application may supply additional SIP headers to be added to the outgoing message. Note if multiple headers are being appended then `\r\n` should be used to delimit each header. For example:

```
"Subject: The meeting"
"Subject: 10acb7899\r\nServer: VoIP server"
```

This string must be null terminated.

message_bodies

Message bodies to be added to the outgoing message can be specified here. See the section 4.1 for further details in the setup of this structure.

media_offer_answer

This field is a pointer to an `ACU_MEDIA_OFFER_ANSWER` specifying the settings relevant to starting a media session with the caller. This field should only be used with `UPDATE` requests. Please refer to the documentation on the `ACU_MEDIA_OFFER_ANSWER` structure for more information on this structure.

Return values

On successful completion, a value of zero is returned. Otherwise, a negative value will be returned indicating the type of error.

Example Usage

```
SIP_SEND_REQUEST_PARMS sip_message_parms;
INIT_ACU_STRUCT(sip_message_parms);

sip_message_parms.handle = hndl;
sip_message_parms.message_type = ACU_SIP_MESSAGE_INFO;

sip_message_parms.custom_headers = "Subject: 1024-abcd";

ACU_RAW_MESSAGE_BODY message_body;
memset(&message_body, 0, sizeof(message_body));

unsigned char BIN_BODY[] = {
0x81, 0x82, 0x1c, 0x05, 0xe4, 0x87, 0xe7, 0x86,
0xc8, 0x00, 0x01, 0x00, 0x00, 0x00, 0x0a, 0x00,
0x02, 0x07, 0x05, 0x04, 0x00, 0x21, 0x43, 0x65,
0x0a, 0x04, 0x84, 0x00, 0x32, 0x04 , 0x00};

message_body.body_type = "application/isup";
message_body.body = (unsigned char*)BIN_BODY;
message_body.body_length = sizeof(BIN_BODY);

sip_message_parms.message_bodies = &message_body;

ACU_ERR err = sip_send_request(&sip_message_parms);
```

The above snippet will send an INFO message with an attached ISUP message body.

3.16 sip_send_response() - send a mid call SIP response

Used to send a mid call SIP response message to the remote party. This routine enables the application to specify additional SIP headers and message bodies to compose a mid call message to be sent in response to a previously received mid call SIP request. Note: under certain circumstances in the case of the `UPDATE` method the `sip_send_response()` can be called during call set-up before the `INVITE` has received a final response in a pending SIP session (RFC 3311).

Synopsis

```
ACU_ERR sip_send_response(SIP_SEND_RESPONSE_PARMS* send_response_parms);

   typedef struct tSIP_SEND_RESPONSE_PARMS
   {
        ACU_ULONG               size;
        ACU_CALL_HANDLE         handle;                 /* mandatory */
        ACU_UINT                message_handle;         /* IN */
        ACU_INT                 sip_code;               /* IN */
        char*                   custom_headers;         /* IN */
        ACU_RAW_MESSAGE_BODY*   message_bodies;         /* IN */
        ACU_MEDIA_OFFER_ANSWER* media_offer_answer;     /* IN */
   } SIP_SEND_RESPONSE_PARMS;
```

Input parameters

The `sip_send_response()` function takes a pointer *`send_response_parms`* to a `SIP_SEND_REQUEST_PARMS` structure. The structure must be initialised in the following way before invoking the function.

handle

The `handle` field identifies the call that is to send the SIP message. This field is **mandatory**.

message_handle

The `message_handle` field identifies the request that this response corresponds to. This may be retrieved by a call to `sip_details()` after the initial request has been received.

sip_code

The `sip_code` field allows the application to specify which SIP response code is to be used for the message.

custom_headers

The application may supply additional SIP headers to be added to the outgoing message. Note if multiple headers are being appended then `\r\n` should be used to delimit each header. For example:

```
"Subject: The meeting"
"Subject: 10acb7899\r\nServer: VoIP server"
```

This string must be null terminated.

message_bodies

Message bodies to be added to the outgoing message can be specified here. See the section 4.1 for further details in the setup of this structure.

media_offer_answer

This field is a pointer to an `ACU_MEDIA_OFFER_ANSWER` specifying the settings relevant to starting a media session with the caller. This field should only be used in response to `UPDATE` requests that contained an SDP offer. Please refer to the documentation on

the ACU_MEDIA_OFFER_ANSWER structure for more information on this structure.

Return values

On successful completion, a value of zero is returned. Otherwise, a negative value will be returned indicating the type of error.

Example Usage

```
SIP_SEND_RESPONSE_PARMS sip_message_parms;
INIT_ACU_STRUCT(sip_message_parms);

sip_message_parms.handle = hndl;
sip_message_parms.message_handle = message_hndl;
sip_message_parms.sip_code = 200;

sip_message_parms.custom_headers = “Subject: 1024-abcd”;

ACU_RAW_MESSAGE_BODY message_body;
memset(&message_body, 0, sizeof(message_body));

unsigned char BIN_BODY[] = {
0x81, 0x82, 0x1c, 0x05, 0xe4, 0x87, 0xe7, 0x86,
0xc8, 0x00, 0x01, 0x00, 0x00, 0x00, 0x0a, 0x00,
0x02, 0x07, 0x05, 0x04, 0x00, 0x21, 0x43, 0x65,
0x0a, 0x04, 0x84, 0x00, 0x32, 0x04 , 0x00};

message_body.body_type = "application/isup";
message_body.body = (unsigned char*)BIN_BODY;
message_body.body_length = sizeof(BIN_BODY);

sip_message_parms.message_bodies = &message_body;

ACU_ERR err = sip_send_response(&sip_message_parms);
```

The above snippet will send an 200 OK response to a previous request with an attached ISUP message body.

3.17 sip_set_reason_phrase() – modify a SIP response reason phrase

In the SIP protocol a human readable string is sent along with a numeric code in a response message. The string's content is not mandated by the protocol, as it is for informational use only. The SIP stack encapsulated below the Aculab API uses default settings for this string, so for example, the string `Ringing` is sent with the 180 response. A different string value may be associated with this numeric code by using this routine. This function call may be used with an application that is based on either the extended or generic API.

NOTE

To provide an optimal implementation, a very small memory leak will occur if this routine is called repeatedly for the same response code. The design assumes that the reason phrase associated with a particular response code will only need to be set once by an application.

Synopsis

```
ACU_ERR sip_set_reason_phrase(SIP_SET_REASON_PHRASE_PARMS*
                                                    set_reason_phrase_parms);
```

```
typedef struct tSIP_SET_REASON_PHRASE_PARMS
{
  ACU_ULONG        size;
  ACU_INT          sip_code;              /* IN */
  const char*      reason_phrase;         /* IN */
} SIP_SET_REASON_PHRASE_PARMS;
```

Input parameters

The `sip_set_reason_phrase()` function takes a pointer *`set_reason_phrase_parms`* to a `SIP_SET_REASON_PHRASE_PARMS` structure. The structure must be initialised in the following way before invoking the function.

sip_code

The *`sip_code`* field identifies the SIP response to which this reason phrase relates.

reason_phrase

A `NULL`-terminated ASCII string, specifying the standard reason phrase to transmit with the SIP response of the above code.

Return values

On successful completion, a value of zero is returned. Otherwise, a negative value will be returned indicating the type of error.

Example Usage

```
SIP_SET_REASON_PHRASE sip_reason_parms;
INIT_ACU_STRUCT(&sip_reason_parms);

sip_reason_parms.sip_code = 180;
sip_reason_parms.reason_phrase = "Alerting";
```

3.18 sip_add_answer_challenge_credentials() – provide authentication credentials

The function is used by the application to pass authentication credentials to the protocol stack. The stack will use the appropriate credentials when challenged by an Authenticating proxy or User Agent. SIP Digest Authentication works in the following way:

- The calling party sends an un-authenticated request to a proxy
- The remote party 'challenges' with an appropriate response (401 from a user agent or registrar, 407 from a proxy) containing the realm and cryptographically opaque nonce
- The stack looks up a username and password pair matching the realm. If there are multiple users associated with that realm the stack will attempt to match the user with the username part of the relevant URI. If no such match exists the password associated with the first user in the realm is used.
- Using the username, password and nonce, the calling party generates a cryptographically opaque response and embeds this in a new request to be resent to the proxy.

`sip_add_answer_challenge_credentials()` should be called by the application prior to making calls which are likely to be challenged. This function call may be used with an application that is based on either the extended or generic API.

NOTE

A 'realm' represents an administrative zone which an entity, for example, a server controls.

Synopsis

```
ACU_ERR sip_add_answer_challenge_credentials
(SIP_ANSWER_CHALLENGE_CREDENTIALS_PARMS* answer_challenge_credentials_parms);

typedef struct tSIP_ANSWER_CHALLENGE_CREDENTIALS_PARMS
{
  ACU_ULONG      size;              /* IN */
  char*          realm;             /* IN */
  char*          user;              /* IN */
  char*          password;          /* IN */
  ACU_INT        retries;           /* IN */
} SIP_ANSWER_CHALLENGE_CREDENTIALS_PARMS;
```

Input parameters

The `sip_add_answer_challenge_credentials()` function takes a pointer `answer_challenge_credentials_parms` to a `SIP_ANSWER_CHALLENGE_CREDENTIALS_PARMS` structure. The structure must be initialised in the following way before invoking the function.

realm

A NULL terminated string containing the name of the administrative zone to which these credentials apply. This field is mandatory.

user

A NULL terminated string containing the name of a user authorised to access the above realm. This field is mandatory.

password

A NULL terminated string containing the password of the above user. This field is

mandatory.

retries
Reserved for future use.

Return values

On successful completion a value of zero is returned; Otherwise, a negative value will be returned indicating the type of error.

Example usage

```
SIP_ANSWER_CHALLENGE_CREDENTIALS_PARMS cred;
INIT_ACU_STRUCT(&cred);

cred.realm = "proxies.com";
cred.user = "john";
cred.password = "password";

ACU_ERR err=sip_add_answer_challenge_credentials(&cred);
```

3.19 sip_remove_answer_challenge_credentials() – remove authentication credentials

This function may be used by an application to remove credentials previously added by `sip_add_answer_challenge_credentials()`. The credentials may then be added again using `sip_add_answer_challenge_credentials()`, for example, in the case of a password change. This function call may be used with an application that is based on either the extended or generic API.

Synopsis

```
ACU_ERR sip_remove_answer_challenge_credentials
(SIP_ANSWER_CHALLENGE_CREDENTIALS_PARMS* answer_challenge_credentials_parms);

typedef struct tSIP_ANSWER_CHALLENGE_CREDENTIALS_PARMS
{
  ACU_ULONG      size;
  char*          realm;         /* IN */
  char*          user;          /* IN */
  char*          password;      /* UNUSED */
  ACU_INT        retries;       /* UNUSED */
} ACU_PACK_DIRECTIVE SIP_ANSWER_CHALLENGE_CREDENTIALS_PARMS;
```

Input parameters

The `sip_remove_answer_challenge_credentials()` function takes a pointer `answer_challenge_credentials_parms` to a `SIP_ANSWER_CHALLENGE_CREDENTIALS_PARMS` structure. The structure must be initialised in the following way before invoking the function.

realm

A NULL terminated string containing the name of the administrative zone to which these credentials apply. This field is mandatory.

user

This field is optional. If this field is not supplied all users in the specified realm will be removed. If it is, only that user will be removed.

password

This field is not required.

retries

This field is not required.

Return values

On successful completion a value of zero is returned. Otherwise a negative value will be returned indicating the type of error.

Example usage

To remove all the users from one realm:

```
SIP_ANSWER_CHALLENGE_CREDENTIALS_PARMS cred;
INIT_ACU_STRUCT(&cred);

cred.realm = "proxies.com";

ACU_ERR err=sip_remove_answer_challenge_credentials(&cred);
```

To remove one user from a realm:

```
SIP_ANSWER_CHALLENGE_CREDENTIALS_PARMS cred;
INIT_ACU_STRUCT(&cred);

cred.realm = "proxies.com";
cred.user = "john";

ACU_ERR err=sip_remove_answer_challenge_credentials(&cred);
```

3.20 sip_disconnect() – send a 3xx response, CANCEL or BYE or response to BYE

This function is used to disconnect a call, specifically an incoming call with a 3xx response and to specify alternative contact details, or to provide custom headers in a `CANCEL` or `BYE` or response to `BYE` message..

Synopsis

```
ACU_ERR sip_disconnect(SIP_DISCONNECT_PARMS* sip_disconnect_parms);

typedef struct tSIP_DISCONNECT_PARMS
{
  ACU_ULONG           size;
  ACU_CALL_HANDLE     handle;                    /* IN */
  ACU_INT             sip_code;                  /* IN */
  ACU_STRING_LIST*    redirect_contact_list;     /* IN */
  ACU_INT             generic_cause;             /* IN */
  char*               custom_headers;            /* IN */
  ACU_RAW_MESSAGE_BODY*   message_bodies;        /* IN */
} SIP_DISCONNECT_PARMS;
```

Input parameters

The `sip_disconnect()` function takes a pointer `sip_disconnect_parms` to a `SIP_DISCONNECT_PARMS` structure. The structure must be initialised in the following way before invoking the function.

handle

The *handle* field identifies the call that is to send the SIP message. This field is **mandatory**.

sip_code

The *sip_code* field specifies which 3xx response to use. This field is **mandatory**.

redirect_contact_list

A linked list of NULL terminated strings containing at least one alternative contact address. This field is **mandatory**.

generic_cause

For backwards compatability with `call_disconnect()`. This field is **optional**.

custom_headers

A CRLF delimited list of additional SIP headers to be included in the request or response. This field is **optional**.

message_bodies

Message bodies to be added to the outgoing message can be specified here. See the section 4.1 for further details in the setup of this structure. This field is **optional**.

Return values

On successful completion a value of zero is returned. Otherwise a negative value will be returned indicating the type of error.

Example usage

```
SIP_DISCONNECT_PARMS disconnect_parms;
INIT_ACU_STRUCT(&disconnect_parms);

disconnect_parms.handle = hndl;
disconnect_parms.sip_code = 302;
disconnect_parms.redirect_contact_list =
                      (ACU_STRING_LIST*)malloc(sizeof(ACU_STRING_LIST));
```

```
disconnect_parms. redirect_contact_info->string =
                          (char*)malloc(strlen(ANOTHER_ADDRESS_STR) + 1);
strpcy(disconnect_parms.redirect_contact_info->string, ANOTHER_ADDRESS_STR);
disconnect_parms.redirect_contact_info->next = NULL;
disconnect_parms.custom_headers = "Date: Mon, 10 Jul 2000 03:55:07 GMT\r\n";

ACU_ERR err=sip_disconnect(&disconnect_parms);

free(disconnect_parms.redirect_contact_info->string);
free(disconnect_parms.redirect_contact_info);
```

A 302 response will now be sent to the UAC containing a single alternative contact address as specified by `ANOTHER_ADDRESS_STR`. An `EV_IDLE` will then be raised to the application.

3.21 sip_recall() – call an alternative address

This function is used to try calling alternative addresses as supplied in a 3xx response. The application will be notified of such responses with an `EV_EXT_DIVERSION`.

```
ACU_ERR sip_recall(SIP_RECALL_PARMS* recall_parms);
```

```
typedef struct tSIP_RECALL_PARMS
{
  ACU_ULONG           size;
  ACU_CALL_HANDLE     handle;                 /* IN */
  char*               destination_addr;       /* IN */
} SIP_RECALL_PARMS;
```

Input parameters

The sip_recall() function takes a pointer `recall_parms` to a `SIP_RECALL_PARMS` structure. The structure must be initialised in the following way before invoking the function.

handle

The *handle* field identifies the call that is to send the SIP message. This field is **mandatory**.

destination_addr

The destination address of the alternative contact to be called. This field is **mandatory**.

Return values

On successful completion a value of zero is returned. Otherwise a negative value will be returned indicating the type of error.

Example usage

```
SIP_RECALL_PARMS recall_parms;
INIT_ACU_STRUCT(&recall_parms);

recall_parms.handle = hndl;
recall_parms.destination_addr = (char*)malloc(strlen(ALTERNATE_ADDRESS) + 1);
strncpy(recall_parms.destination_addr, ALTERNATE_ADDRESS,
strlen(ALTERNATE_ADDRESS) + 1);

ACU_ERR err=sip_recall(&recall_parms);

free(recall_parms.destination_addr);
```

An identical INVITE to that sent in the original `sip_openout()` (apart from the Cseq and To headers which will have been updated) will now be sent to the alternative address supplied. The call will then proceed in the normal manner.

3.22 sip_set_tls_private_key_password() – pass a password for a TLS private key

A TLS enabled SIP, or rather 'SIPS', application may wish to either act as a 'server' in the TLS handshake and/or possess its certificate for authentication purposes. The TLS implementation looks for a private key this certificate. The private key *may* be password protected; in this case it is necessary that the password be supplied to the TLS implementation at the time of initialisation. This routine enables the application to pass the password down to the TLS implementation. A brief discussion of the usage of TLS in the Aculab SIP service is given in Appendix D:.

```
ACU_ERR
sip_set_tls_private_key_password(SIP_SET_TLS_PRIVATE_KEY_PASSWD_PARMS*
sip_set_tls_private_key_password_parms);

typedef struct tSIP_SET_TLS_PRIVATE_KEY_PASSWD_PARMS
{
  ACU_ULONG      size;
  char*          password; /* IN */
  char*          reserved; /* reserved */
  ACU_UINT options;
} ACU_PACK_DIRECTIVE SIP_SET_TLS_PRIVATE_KEY_PASSWD_PARMS;
```

Input parameters

The `set_tls_private_key_password()` function takes a pointer to a `SIP_SET_TLS_PRIVATE_KEY_PASSWD PARMS` structure. The structure must be initialised in the following way before invoking the function.

password
A NULL terminated string containing the required password.

reserved
Reserved for future use.

options
Reserved for future use.

Return values

On successful completion a value of zero is returned. ERR_PARM is returned if a password of zero length is passed by the application.

Example usage

```
SIP_SET_TLS_PRIVATE_KEY_PASSWD_PARMS parms;
INIT_ACU_STRUCT(&parms);

// read password from a file or data-base
char buffer[512];

parms.password = buffer;
sip_set_tls_private_key_password(&parms);
```

3.23 sip_load_tls_configuration() – load a new set of TLS certificates

The SIP service must be configured to listen for TLS via the configuration file sipserv.cfg, however, it may be necessary to change the current set of certificates at runtime.

The application calls this routine to load a new set of TLS certificates into the stack. Ongoing calls using existing TLS connections will not be affected until the connection expires. At that point, any further messaging will use the new certificates.

Synopsis:

```
ACU_ERR sip_load_tls_configuration(SIP_TLS_CONFIG_PARMS* sip_tls_config)

typedef struct tSIP_TLS_CONFIG_PARMS
{
   ACU_ULONG size;
   char* trusted_certificates_file; /* IN */
   char* server_certificates_file;  /* IN */
   char* dh512_file;                /* IN */
   char* dh1024_file;               /* IN */
   char* password;                  /* IN */
   ACU_INT reserved;
}
SIP_TLS_CONFIG_PARMS;
```

Input Parameters:

trusted_certificates_file

Mandatory. The value should be an absolute path to a file containing the certificates, which this application trusts. Typically, these certificates will be the public certificates of the 'Certification Authority' (CA) for the application.

server_certificates_file

Optional. This value should be an absolute path to a file containing the 'server certificate', which this application wishes to use. 'Server certificate' is a slight misnomer in that the contents of the file may be used by the client entity in a transaction; it is named as such since it is mandatory in a server application.

dh512_file

Optional. The value should be an absolute path to a 512-bit Diffie-Hellmann file. Please refer to Appendix D for details.

dh1024_file

Optional. The value should be an absolute path to a 1024-bit Diffie-Hellmann file. Please refer to Appendix D for details.

password

Optional. If the private key contained within the server certificate is password protected, the password must be supplied here as a NULL terminated string.

reserved

Reserved for future use.

Return values

On successful completion, a value of zero is returned; otherwise a negative value will be returned indicating the type of error.

Example usage

```
SIP_TLS_CONFIG_PARMS parms;
INIT_ACU_STRUCT(&parms);

parms.trusted_certificates_file = "\my_certificates_path\new_trusted.pem";
parms.server_certificates_file = "\my_certificates_path\new_client.pem";
parms.password = my_password_buffer;

sip_load_tls_configuration(&parms);
```

The above example will load certificates files `new_trusted.pem` and `new_client.pem` into the SIP service for use with new TLS connections. Ongoing calls on existing TLS connections will not be affected until those connections expire.

3.24 sip_set_message_notification() - declare an interest in out of dialog messages

The application calls this routine to configure the SIP service in order that it may be notified on receipt of particular out of dialog SIP messages. The application may specify whether or not to be notified on receipt of requests and/or on receipt of responses. After a request is sent the service may be instructed to either automatically send a 200 response or to delegate response sending to the application. The application is notified of the receipt of message or a timeout by a port event.

Synopsis:

```
ACU_ERR sip_set_message_notification(SIP_MESSAGE_NOTIFICATION_PARMS*)

typedef struct  tSIP_MESSAGE_NOTIFICATION_PARMS
{
  ACU_LONG    size;
  ACU_PORT_ID port_id;                        /* IN */
  ACU_UINT    request_notification_mask;      /* IN */
  ACU_UINT    response_notification_mask;     /* IN */
  ACU_UINT    enable_response_mask;           /* IN */
} SIP_MESSAGE_NOTIFICATION_PARMS;
```

Input parameters:

port_id

Mandatory. Specifies which Aculab call control port the notification should be sent to.

request_notification_mask

Holds a bitmask of the types of request in which the application would like to receive notification. The available options are presented in the enum `ACU_SIP_MESSAGE_NOTIFICATION_MASKS`.

response_notification_mask

Holds a bitmask of the types of response in which the application would like to receive notification. The available options are presented in the enum `ACU_SIP_MESSAGE_NOTIFICATION_MASKS`.

enable_response_mask

Holds a bitmask of the types of request in which the application would like to send a response to, as opposed to the service responding automatically. The available options are presented in the enum `ACU_SIP_MESSAGE_NOTIFICATION_MASKS`.

Return values

On successful completion, a value of zero is returned; otherwise a negative value will be returned indicating the type of error.

Example usage

```
SIP_MESSAGE_NOTIFICATION_PARMS parms;
INIT_ACU_STRUCT(&parms);

parms.port_id = g_sip_port;
parms.request_notification_mask =
   ACU_SIP_REGISTER_NOTIFICATION | ACU_SIP_OPTIONS NOTIFICATION;
parms.enable_response_mask =
ACU_SIP_REGISTER_NOTIFICATION | ACU_SIP_OPTIONS NOTIFICATION;

sip_set_message_notification(&parms);
```

The above example results in the application being notified by a port event on the specified port on receipt of a REGISTER or OPTIONS request. Additionally the application wishes to make the responses to such requests itself.

3.25 sip_send_out_of_dialog_request() - send an out of dialog request

The application sends an out of dialog request using this function. Such a request has no associated telephony call and therefore is not identified by an Aculab call handle. The application is required to supply a message type field and the relevant SIP 'To' and 'From' headers. After a successful call to this routine, the `transaction_id` field is populated with a value, which can be used to identify the subsequent response or timeout relevant to this transmission.

For some out of dialog messages it may be necessary to configure the `request_uri` for correct routing. Other message type will require `custom_headers` and `message_bodies` to be used.

Synopsis:

```
ACU_ERR sip_send_out_of_dialog_request(SIP_SEND_OUT_OF_DIALOG_REQUEST_PARMS*
parms)

typedef struct  tSIP_SEND_OUT_OF_DIALOG_REQUEST_PARMS
{
   ACU_ULONG                size;              /*IN*/
   unsigned char            message_type;      /*IN*/
   char*                    request_uri;       /*IN*/
   char*                    to;                /*IN*/
   char*                    from;              /*IN*/
   char*                    local_address;     /*IN*/
   char*                    custom_headers;    /*IN*/
   ACU_RAW_MESSAGE_BODY*    message_bodies;    /*IN*/
   ACU_POINTER              transaction_id;    /*OUT*/
} SIP_SEND_OUT_OF_DIALOG_REQUEST_PARMS;
```

Input parameters:

The function takes a pointer to a `SIP_SEND_OUT_OF_DIALOG_REQUEST_PARMS` structure. The structure must be set up in the following way.

message_type

Mandatory. Specifies the SIP message type, e.g. REGISTER, PUBLISH. Refer to the enum `ACU_SIP_MESSAGE_TYPE` for the available choices.

request_uri

Optional. Specifies the Request URI to be used for this message. The URI in the *`to`* parameter is used if this is not set. If set, then this must be a fully qualified sip or sips URI. Note that for certain SIP requests, e.g. REGISTER, this will be different to the *`to`* parameter, and should therefore be specified.

to

Mandatory. Valid right hand side of a SIP To: header. e.g. `sip:fred@aculab` or `<sip:fred@aculab>;to-param=value`

Valid right hand side of a TEL To: header.

e.g. `tel:+1-201-555-0123` or <tel:7042;phone-context=example.com>;to-param=value

from

Mandatory. Valid right hand side of a SIP From: header. e.g. `sip:fred@aculab` or `<sip:fred@aculab>;tag=value`

Valid right hand side of a TEL From: header.
e.g. `tel:+1-201-555-0123` or `<tel:+1-212-555-3141;ext=456>;tag=value`

local_address

Reserved for future use. i.e. STUN.

custom_headers
An optional CRLF delimited list of additional SIP headers that will be included in the request. If a TEL URI scheme is used in the formation of the To header then a Route header must be added using custom headers or by the setting of a proxy. See Appendix E.

message_bodies
An optional list of message bodies that will be included in the request.

Return values

transaction_id
If successful, the `transaction_id` field is populated by the SIP process with a number identifying this message. This id is no longer valid once the final response is received or the SIP transaction times out.

On successful completion, a value of zero is returned; otherwise, a negative value will be returned indicating the type of error.

If the config option `AsyncOOD` is set then zero will always be returned. Following this an attempt will be made to resolve the destination URI before sending the request. Receiving the event `ACU_SIP_EV_REQUEST_FAILED` indicates that an error occurred.

Example usage

```
SIP_SEND_OUT_OF_DIALOG_REQUEST_PARMS parms;
INIT_ACU_STRUCT(&parms);

parms.message_type = ACU_SIP_MESSAGE_NOTIFY;

parms.to = “sip:alice@example.com”;
parms.from = “sip:alice@example.com”;

parms.custom_headers = “Date: Mon, 10 Jul 2000 03:55:07 GMT\r\n”
             “Contact: <sip:alice@vmail.example.com>\r\n”
                       “Event: message-summary\r\n”
                       “Subscription-State: active\r\n”
                    “Content-Type: application/simple-message-summary\r\n”;

char* gIsupRequestBody = "IsupRequestBody";
char* gQsigRequestBody = "QsigRequestBody";

ACU_RAW_MESSAGE_BODY isupBody;
ACU_RAW_MESSAGE_BODY qsigBody;

memset(&isupBody, 0, sizeof(ACU_RAW_MESSAGE_BODY));
isupBody.body_type = "application/isup; version=nxv3";
isupBody.body = gIsupRequestBody;
isupBody.body_length = strlen(gIsupRequestBody);

memset(&qsigBody, 0, sizeof(ACU_RAW_MESSAGE_BODY));
qsigBody.body_type = "application/qsig";
qsigBody.body = gQsigRequestBody;
qsigBody.body_length = strlen(gQsigRequestBody);

parms.message_bodies = &isupBody;
parms.message_bodies->next = &qsigBody;

ACU_POINTER trans_id = 0;

if(ERR_NO_ERROR == sip_send_out_of_dialog_request(&parms))
{
   // cache the transaction identifier for subsequent responses
   trans_id = parms.transaction_id;
}
```

The above example sends an out of dialog NOTIFY message.

3.26 sip_send_out_of_dialog_response() - send an out of dialog response

The application sends a response to an out of dialog request using this function. The response must specify the relevant `transaction_id` received with the request in order that the stack replies correctly.

Synopsis:

```
ACU_ERR sip_send_out_of_dialog_response(SIP_SEND_OUT_OF_DIALOG_RESPONSE_PARMS* parms)

typedef struct tSIP_SEND_OUT_OF_DIALOG_RESPONSE_PARMS
{
  ACU_ULONG                 size;
  ACU_INT                   sip_code;          /*IN*/
  ACU_POINTER               transaction_id;    /*IN*/
  ACU_LONG                  port_id;           /*IN*/
  char*                     custom_headers;    /*IN*/
  ACU_RAW_MESSAGE_BODY*     message_bodies;    /*IN*/
} SIP_SEND_OUT_OF_DIALOG_RESPONSE_PARMS;
```

Input parameters:

The function takes a pointer to a `SIP_SEND_OUT_OF_DIALOG_RESPONSE_PARMS` structure. The structure must be set up in the following way.

sip_code
Mandatory. Valid SIP response code, such as 200 for 200OK and so on.

transaction_id
Mandatory. Identifies which request this function call is a response to.

port_id
Reserved for future use.

custom_headers
Optional. A CRLF delimited list of additional SIP headers to be included in the request. Contact header must be supplied by Registrar implementers.

message_bodies
Optional. A list of message bodies to be included in the request.

Return values

On successful completion, a value of zero is returned; otherwise a negative value will be returned indicating the type of error.

Example usage

```
SIP_SEND_OUT_OF_DIALOG_RESPONSE_PARMS parms;
INIT_ACU_STRUCT(&parms);

parms.transaction_id = trans_id;
parms.sip_code = 200;
parms.custom_headers = "Contact: <sip:fred@192.168.3.4>\r\n";
sip_send_out_of_dialog_response(&parms);
```

3.27 sip_read_request() - collect an out of dialog request

This function collects an out of dialog request. The presence of such a request is signalled by a port event of type `ACU_SIP_EV_REQUEST`. If required the application should cache the `transaction_id` reported by this function for use in a subsequent response.

Synopsis:

```
ACU_ERR sip_read_request(SIP_READ_MESSAGE_PARMS* parms)

typedef struct tSIP_READ_MESSAGE_PARMS
{
  ACU_LONG                size;
  ACU_SIP_MESSAGE         message;           /*OUT*/
  ACU_PORT_ID             port_id;           /*IN*/
  ACU_POINTER             transaction_id;    /*OUT*/
  ACU_UINT                cause;             /*OUT*/
} SIP_READ_MESSAGE_PARMS;
```

Input parameters:

This function takes a pointer to a `SIP_READ_MESSAGE_PARMS` structure. The structure should be initialised in the following way.

port_id
Mandatory. Specifies the *`port_id`* in which the notification occurred.

Return values

message
Populated with the received request.

transaction_id
Populated with the `transaction_id` of the request that is to be cached for use in the response.

cause
Not used.

On successful completion, a value of zero is returned; otherwise a negative value will be returned indicating the type of error.

Example usage

```
SIP_READ_MESSAGE PARMS parms;
INIT_ACU_STRUCT(&parms);

parms.port_id = g_sip_port;

ACU_POINTER trans_id = 0;

if(ERR_NO_ERROR == sip_read_request(&parms))
{
   trans_id = parms.transaction_id;

   // parse details from message and respond
   // if desired
}
```

3.28 sip_read_response() - collect an out of dialog response

This function collects an out of dialog response. The presence of such a response is signalled by a port event of type `ACU_SIP_EV_RESPONSE`. The `transaction_id` reported by this function enables the application to match the response to the earlier request.

Synopsis:

```
ACU_ERR sip_read_response(SIP_READ_MESSAGE_PARMS* parms)

typedef struct tSIP_READ_MESSAGE_PARMS
{
  ACU_LONG                size;
  ACU_SIP_MESSAGE         message;          /*OUT*/
  ACU_PORT_ID             port_id;          /*IN*/
  ACU_POINTER             transaction_id;   /*OUT*/
  ACU_UINT                cause;            /*OUT*/
} SIP_READ_MESSAGE_PARMS;
```

Input parameters:

This function takes a pointer to a `SIP_READ_MESSAGE_PARMS` structure. The structure should be initialised in the following way.

port_id
Mandatory. Specifies the `port_id` in which the notification occurred.

Return values

message
Populated with the response received.

transaction_id
Populated to enable identification of the previous request.

cause
Not used.

On successful completion, a value of zero is returned; otherwise, a negative value will be returned indicating the type of error.

Example usage

```
SIP_READ_MESSAGE PARMS parms;
INIT_ACU_STRUCT(&parms);

parms.port_id = g_sip_port;

ACU_POINTER trans_id = 0;

if(ERR_NO_ERROR == sip_read_response(&parms))
{
   trans_id = parms.transaction_id;

   // lookup up original request and parse response
}
```

3.29 sip_read_out_of_dialog_failure () – collect out of dialog failure notification

This function collects out of dialog timeout information, after a request was sent but the response did not arrive in a timely fashion. This occurrence is signalled by a port event of type `ACU_SIP_EV_REQUEST_FAILED`. The *`transaction_id`* reported by this function enables the application to match this to the earlier request. This function replaces `sip_read_timeout()`. The event `ACU_SIP_EV_REQUEST_TIMEOUT` is deprecated.

Synopsis

```
ACU_ERR sip_read_out_of_dialog_failure(SIP_READ_MESSAGE_PARMS* parms)

typedef struct tSIP_READ_MESSAGE_PARMS
{
  ACU_LONG                size;
  ACU_SIP_MESSAGE         message;          /*NOT USED*/
  ACU_PORT_ID             port_id;          /*IN*/
  ACU_POINTER             transaction_id;   /*OUT*/
  ACU_UINT                cause;            /*OUT*/
} SIP_READ_MESSAGE_PARMS;
```

Input parameters

This function takes a pointer to a `SIP_READ_MESSAGE_PARMS` structure. The structure should be initialised before invoking the function.

port_id

Mandatory. Specifies the *`port_id`* in which the notification occurred.

Return values

message

Not used.

transaction_id

Populated with the `transaction_id` of the response to enable identification of the previous request.

cause

This is populated with the reason for the failure of the out of dialog request. These values are defined in the following enum:

```
typedef enum acu_sip_ood_causes
{
   ACU_OOD_NO_ERROR                  =    0,
   ACU_OOD_REQUEST_TIMEOUT           =    1,
   ACU_OOD_TCP_CONNECT_FAILED        =    2,
   ACU_OOD_SSL_ERROR                 =    3,
   ACU_OOD_SSL_PEER_CERT_NOT_TRUSTED=    4,
   ACU_OOD_SSL_PEER_CERT_INVALID     =    5,
   ACU_OOD_UNRESOLVABLE_NAME         =    6,
   ACU_OOD_REQUEST_FAILED            =    7
} ACU_SIP_OOD_CAUSES;
```

On successful completion, a value of zero is returned; otherwise, a negative value will be returned indicating the type of error.

Example usage

```
SIP_READ_MESSAGE PARMS parms;
INIT_ACU_STRUCT(&parms);

parms.port_id = g_sip_port;
```

```
ACU_POINTER trans_id = 0;

if(ERR_NO_ERROR == sip_read_out_of_dialog_failure(&parms))
{
   trans_id = parms.transaction_id;

   // lookup up original request and parse response
}
```

3.30 sip_free_message()- free memory associated with out of dialog notification

The routines `sip_read_request()` and `sip_read_response()` dynamically allocate memory from the heap. Once the information provided by these functions has been processed, it is necessary to return the resource back to the free store. This role is achieved by `sip_free_message()`.

Synopsis

```
ACU_ERR sip_free_message(SIP_READ_MESSAGE_PARMS* parms)

typedef struct tSIP_READ_MESSAGE_PARMS
{
  ACU_LONG                size;
  ACU_SIP_MESSAGE         message;           /*NOT USED*/
  ACU_PORT_ID             port_id;           /*IN*/
  ACU_POINTER             transaction_id;    /*OUT*/
} SIP_READ_MESSAGE_PARMS;
```

Input parameters

This function takes a pointer to a `SIP_READ_MESSAGE_PARMS` structure, which has previously been populated by a call to either `sip_read_request()` or `sip_read_response()`.

Return values

On successful completion, a value of zero is returned; otherwise, a negative value will be returned indicating the type of error.

Example usage

```
SIP_READ_MESSAGE PARMS parms;
INIT_ACU_STRUCT(&parms);

parms.port_id = g_sip_port;

ACU_POINTER trans_id = 0;

if(ERR_NO_ERROR == sip_read_request(&parms))
{
   // process details resulting from the sip_read_request call

   sip_free_message(&parms);
}
```

3.31 sip_sub_subscriber() – SUBSCRIBE to an event package

This routine populates the contents of a SUBSCRIBE request and sends it to a notifier.

Synopsis

```
ACU_ERR sip_sub_subscriber(SIP_SUB_SUBSCRIBER_PARMS* parms)

typedef struct tSIP_SUB_SUBSCRIBER_PARMS
{
   ACU_ULONG               size;                   /*IN*/
   ACU_CALL_HANDLE         handle;                 /*OUT*/
   ACU_PORT_ID             net;                    /*IN*/
   ACU_EVENT_QUEUE         queue_id;               /*IN*/
   ACU_ACT                 app_context_token;      /*IN*/
   ACU_INT                 response_notification;  /*IN*/
   ACU_INT                 enable_response;        /*IN*/
   ACU_INT                 expires;                /*IN*/
   ACU_INT                 allow_multiple_dialogs  /*IN*/
   char*                   event_package_token     /*IN*/
   char*                   event_id_param;         /*IN*/
   char*                   request_uri;            /*IN*/
   char*                   to_header;              /*IN*/
   char*                   from_header;            /*IN*/
   char*                   contact_address;        /*IN*/
   char*                   custom_headers;         /*IN*/
   ACU_STRING_LIST*        body_types;             /*IN*/
   ACU_RAW_MESSAGE_BODY*   message_bodies;         /*IN*/
} SIP_SUB_SUBSCRIBER_PARMS;
```

Input parameters

This function takes a pointer to a SIP_SUB_SUBSCRIBER_PARMS structure. The structure should be initialised before invoking the function.

net

Specifies the ACU_PORT_ID referring to the protocol stack on which the subscription is to be created, as returned from sip_open_port(). This is a mandatory field.

queue_id

The unique event queue identity as returned by acu_allocate_event_queue() when creating a queue.

app_context_token

This is a user-defined token value that will be associated with the supplied handle.

response_notification

This field permits an application to specify if it wishes to be notified of responses to the SUBSCRIBE requests sent by this subscriber. On receipt of a SUBSCRIBE response, an application that has set this value to 1 will receive an EV_DETAILS event and a subsequent call to sip_details() will collect the entire message. There is no equivalent for request notifications as NOTIFY requests are already presented in their entirety to the application.

enable_response

If this field is set to 1, NOTIFY requests received by this subscriber will not be automatically responded to by the SIP service. It is then the application's responsibility to send the response itself using sip_send_response().

expires

This value, given in seconds, is used to set the duration of the subscription. This is supplied in the request and the notifier is free to reduce this value according to its local policy. The actual duration will be contained in the response to the SUBSCRIBE. The SIP service will re-submit the SUBSCRIBE after half this duration has expired. If this field is set to 0, the resulting SUBSCRIBE will be sent with a 0 expiration time causing

an immediate fetch of state but no persistent subscription.

allow_multiple_dialogs
Some event packages allow multiple dialogs to be established as a result of the initial `SUBSCRIBE` request forking in a proxy. Set this field to 1 if multiple dialogs are allowed. If this field is set to 0, all received NOTIFY requests that do not match the initial dialog will be responded to with `481 Subscription does not exist`.

event_package_token
This field contains the token defining the event package for which a subscription is being requested (usually IANA registered). This field corresponds directly to the contents of the `Event` header contained in the `SUBSCRIBE` message. This is a mandatory field.

event_id_param
Reserved for future use.

request_uri
This optional field specifies the Request-URI to be contained within the `SUBSCRIBE` request. If not present, the URI part of the *`to_header`* field will be used.

to_header
This field specifies the destination address of the notifier including, if necessary, any display name. This is a mandatory field.

from_header
This field specifies the originating address of the subscriber including, if necessary, any display name. If not present, the SIP service will use the local host address in the form `"sip:<host address>"` for this field.

contact_address
Used to build a non-default contact header, this is useful if the application is running on a multi-homed machine and wishes a particular IP address be used in the `Contact` header. For a chassis containing only one NIC card, this field may be left blank. It should be supplied as a null terminated ASCII string in URI address format.

custom_headers
The application may supply additional SIP headers to be added to the outgoing `SUBSCRIBE` request. Note if multiple headers are being appended then `\r\n` should be used to delimit each header.

For example:

```
subscriber.custom_headers = "Subject: The meeting";
subscriber.custom_headers = "Subject: 10acb7899\r\nServer: VoIP server";
```

This field must be null terminated.

body_types
This field corresponds directly to additional `Accept` headers to be placed into the outgoing `SUBSCRIBE` request.

For example:

```
subscriber.body_types->string = "application/simple-message-summary";
subscriber.body_types->next = 0;
```

This will result in an "`Accept: application/simple-message-summary`" header being placed into the `SUBSCRIBE`.

message_bodies
Message bodies to be added to the outgoing message can be specified here. See section 4 for further details of the setup of this structure `ACU_RAW_MESSAGE_BODY`.

For example:

```
ACU_RAW_MESSAGE_BODY arm;
memset(&arm, 0, sizeof(ACU_RAW_MESSAGE_BODY));

// setup the message body structure - see sip_send_request() for further
details

subscriber.message_bodies = &arm;
```

Return values

handle

If successful, this will contain a unique (non zero) call identifier, which is used in all successive call related operations on the driver.

On successful completion, a value of zero is returned otherwise a negative value will be returned indicating the type of error.

If the config option `AsyncSubscribe` is set then zero will always be returned. Following this an attempt will be made to resolve the destination URI before sending the request. Receiving the event `EV_SIP_SUBSCRIPTION_CANCELLED` indicates that an error occurred.

Example usage

```
SIP_SUB_SUBSCRIBER_PARMS subscriber;
INIT_ACU_STRUCT(&subscriber);

subscriber.net = sipPort;

// addressing
subscriber.to_header="\”John\” <sip:john@gw.com>";
subscriber.from_header="\”Bob\” <sip:bob@voip-company.com>";

// Event package identifier must be present
subscriber.event_package_token=”DummyEventPackage”;

ACU_ERR rc = sip_sub_subscriber(&subscriber);
```

3.32 sip_sub_notifier() – wait for SUBSCRIBE requests to a specific event package

This routine allows the application to wait for an incoming SUBSCRIBE request for a specific event package.

Synopsis

```
ACU_ERR sip_sub_notifier(SIP_SUB_NOTIFIER_PARMS* parms);


typedef struct sip_sub_notifier_parms
{
   ACU_ULONG          size;                    /*IN*/
   ACU_CALL_HANDLE    handle;                  /*OUT*/
   ACU_PORT_ID        net;                     /*IN*/
   ACU_EVENT_QUEUE    queue_id;                /*IN*/
   ACU_ACT            app_context_token;       /*IN*/
   ACU_INT            request_notification;    /*IN*/
   ACU_INT            response_notification;   /*IN*/
   ACU_INT            enable_response;         /*IN*/
   char*              event_package_token      /*IN*/
} SIP_SUB_NOTIFIER_PARMS;
```

Input parameters

This function takes a pointer to a SIP_SUB_NOTIFIER_PARMS structure. The structure should be initialised before invoking the function.

net

Specifies the ACU_PORT_ID referring to the protocol stack on which the subscription is to be created, as returned from sip_open_port(). This is a mandatory field.

queue_id

The unique event queue identity as returned by acu_allocate_event_queue() when creating a queue.

app_context_token

This is a user-defined token value which will be associated with the handle.

request_notification

By default, notifiers are only presented with the raw SUBSCRIBE request that creates the initial subscription. If this field is set to 1 the application will be presented with all subsequent SUBSCRIBE requests for this subscription.

response_notification

This field permits an application to specify if it wishes to be notified of responses to the NOTIFY requests sent by this notifier. On receipt of a NOTIFY response, an application that has set this value to 1 will receive an EV_DETAILS event and a subsequent call to sip_details() will collect the entire message.

enable_response

If this field is set to 1, SUBSCRIBE requests received by this subscriber subsequent to the initial SUBSCRIBE will not be automatically responded to by the SIP service. It is then the application's responsibility to send the response itself using sip_send_response().

event_package_token
This field contains the token defining the event package for which this notifier is prepared to accept `SUBSCRIBE` requests (usually IANA registered). This is a mandatory field.

Return values

handle
If successful, this will contain a unique (non zero) call identifier, which is used in all successive call related operations on the driver.

On successful completion, a value of zero is returned otherwise a negative value will be returned indicating the type of error.

Example usage

```
SIP_SUB_NOTIFIER_PARMS notifier;
INIT_ACU_STRUCT(&notifier);

notifier.net = sipPort;

// Event package identifier must be present
subscriber.event_package_token=”DummyEventPackage”;

ACU_ERR rc = sip_sub_notifier(&notifier);
```

3.33 sip_sub_accept() – accept or acknowledge a SUBSCRIBE request

This routine allows an application to accept or acknowledge an incoming SUBSCRIBE request.

Synopsis

```
ACU_ERR sip_sub_accept(SIP_SUB_ACCEPT_PARMS* parms);


typedef struct sip_sub_accept_parms
{
   ACU_ULONG                size;              /*IN*/
   ACU_CALL_HANDLE          handle;            /*IN*/
   ACU_INT                  expires;           /*IN*/
   char*                    contact_address;   /*IN*/
   char*                    custom_headers;    /*IN*/
   char*                    event_id_param;    /*IN*/
   ACU_RAW_MESSAGE_BODY*    message_bodies;    /*IN*/
   char                     acknowledge;       /*IN*/
} SIP_SUB_ACCEPT_PARMS;
```

Input parameters

This function takes a pointer to a SIP_SUB_ACCEPT_PARMS structure. The structure should be initialised before invoking the function.

handle

The *handle* field identifies the subscription which is to be accepted. This field is mandatory.

expires

This field specifies the duration in seconds for which the subscription is to be active. ERR_PARM will be returned if this is greater than the expires value contained in the initial SUBSCRIBE request which can be ascertained from sip_details() after receiving an EV_SIP_SUBSCRIPTION_REQUEST.

contact_address

Used to build a non-default contact header, this is useful if the application is running on a multi-homed machine and wishes a particular IP address be used in the contact. For a chassis containing only one NIC card, this field may be left blank. It must be a null terminated ASCII string in URI address format.

custom_headers

The application may supply additional SIP headers to be added to the outgoing SUBSCRIBE request. Note if multiple headers are being appended then \r\n should be used to delimit each header.

For example:

```
accept.custom_headers = "Subject: The meeting";
accept.custom_headers = "Subject: 10acb7899\r\nServer: VoIP server";
```

This field must be null terminated.

event_id_param

Reserved for future use.

message_bodies

Message bodies to be added to the outgoing message can be specified here. See section 4.1 for further details of the setup of this structure `ACU_RAW_MESSAGE_BODY`.

For example:

```
ACU_RAW_MESSAGE_BODY arm;
memset(&arm, 0, sizeof(ACU_RAW_MESSAGE_BODY));

// setup the message body structure - see sip_send_request() for further
details

accept.message_bodies = &arm;
```

acknowledge

This field specifies whether the subscription is to be immediately accepted (with a `200 OK` response) or acknowledged (with a 202 response). If set, a `202 Accepted` response will be sent.

Return values

On successful completion, a value of zero is returned otherwise a negative value will be returned indicating the type of error.

Example usage

```
SIP_SUB_ACCEPT_PARMS accept;
INIT_ACU_STRUCT(&accept);

accept.handle = handle;

ACU_ERR rc = sip_sub_accept&accept);
```

3.34 sip_sub_notify() – NOTIFY a subscriber of a state change in an event package

This routine allows an application to notify a subscriber of a state change in the event package being implemented. There are certain times when this must be called in order to conform to RFC 3265. These are the following:

- Immediately after the initial `SUBSCRIBE` has been accepted or acknowledged. The application will be notified of this requirement through an `EV_SIP_SUBSCRIBED` event.
- Immediately after all subsequent `SUBSCRIBE` requests have been received. The application will be notified of this requirement through an `EV_SIP_SUBSCRIPTION_REFRESH` event.
- Immediately after a final `SUBSCRIBE` that contains an `Expires` header whose value is 0 is received. A `SUBSCRIBE` of this type cancels the subscription. The application will be notified of this requirement through an `EV_SIP_SUBCRIPTION_CANCELLED` event.

Synopsis

```
ACU_ERR sip_sub_notify(SIP_SUB_NOTIFY_PARMS* parms);


typedef struct sip_sub_notify_parms
{
   ACU_ULONG               size;              /*IN*/
   ACU_CALL_HANDLE         handle;            /*IN*/

   char*                   custom_headers;    /*IN*/

   ACU_RAW_MESSAGE_BODY*   message_bodies;    /*IN*/

   char                    pending;           /*IN*/

   char*                   event_id_param;    /*IN*/

} SIP_SUB_NOTIFY_PARMS;
```

Input parameters

This function takes a pointer to a `SIP_SUB_NOTIFY_PARMS` structure. The structure should be initialised before invoking the function.

handle

The *handle* field identifies the subscription that is to send the `NOTIFY` message. This field is mandatory.

custom_headers

The application may supply additional SIP headers to be added to the outgoing `NOTIFY` request. Note if multiple headers are being appended then `\r\n` should be used to delimit each header.

For example:

```
notify.custom_headers = "Subject: The meeting";
notify.custom_headers = "Subject: 10acb7899\r\nServer: VoIP server";
```

This field must be null terminated.

message_bodies

Message bodies to be added to the outgoing message can be specified here. See

section 4.1 for further details of the setup of this structure ACU_RAW_MESSAGE_BODY.

For example:

```
ACU_RAW_MESSAGE_BODY arm;
memset(&arm, 0, sizeof(ACU_RAW_MESSAGE_BODY));

// setup the message body structure - see sip_send_request() for further
details

notify.message_bodies = &arm;
```

pending
This field may be used to transition a subscription from “pending” to “active” if and only if the previous call to sip_sub_accept() had the acknowledge field set to 1. ERR_PARM will be returned if pending is set when the subscription is already active.

event_id_param
Reserved for future use.

Return values

On successful completion, a value of zero is returned otherwise a negative value will be returned indicating the type of error.

Example usage

```
SIP_SUB_NOTIFY_PARMS notify;
INIT_ACU_STRUCT(&notify);

notify.handle = handle;

// This is not a mandatory field but it is highly likely to be required by
// the specific event package being implemented.
ACU_RAW_MESSAGE_BODY arm;
memset(&arm, 0, sizeof(ACU_RAW_MESSAGE_BODY));

// setup the message body structure - see sip_send_request() for
// further details

notify.message_bodies = &arm;

ACU_ERR rc = sip_sub_notify(&notify);
```

3.35 sip_sub_cancel() – cancel an existing subscription

This routine allows an application to cancel an existing subscription. This results in different behaviour depending on whether the handle parameter references a notifier or a subscriber:

- For a notifier, a `NOTIFY` request will be sent containing a `Subscription-State` header whose value is `"terminated"`. An `EV_SIP_SUBSCRIPTION_CANCELLED` will be raised on receipt of the response.
- For a subscriber, one or more `SUBSCRIBE` requests will be sent containing an `Expires` header whose value is 0. `EV_SIP_SUBSCRIPTION_CANCELLED` will not be raised until the notifier at the far end sends a final `NOTIFY` to terminate the subscription.

Synopsis

```
ACU_ERR sip_sub_cancel(SIP_SUB_CANCEL_PARMS* parms);


typedef struct sip_sub_cancel_parms
{
   ACU_ULONG               size;              /*IN*/
   ACU_CALL_HANDLE         handle;            /*IN*/
   ACU_INT                 dialog_id;         /*IN*/
   ACU_INT                 response_code;     /*IN*/
   char*                   reason;            /*IN*/
   char*                   custom_headers;    /*IN*/
   char*                   event_id_param;    /*IN*/
   ACU_RAW_MESSAGE_BODY*   message_bodies;    /*IN*/
}  SIP_SUB_CANCEL_PARMS;
```

Input parameters

This function takes a pointer to a `SIP_SUB_CANCEL_PARMS` structure. The structure should be initialised before invoking the function.

handle

The *`handle`* field identifies the subscription that is to be cancelled. This field is mandatory.

dialog_id

`SUBSCRIBE` requests may result in multiple notifiers creating a dialog within the subscription if the request is forked (e.g. by a proxy). This field identifies which dialog is to be cancelled. If it is set to 0, all dialogs associated with this subscription will be cancelled. See `ACU_SUBSCRIPTION_INFO` for more details on the usage of this field.

This field has no meaning for notifiers as they are only ever associated with a single dialog.

response_code

This field only applies to notifiers who wish to reject an initial `SUBSCRIBE` request at `EV_SIP_SUBSCRIPTION_REQUEST`. For example, the application may wish to authenticate the request by sending a `401` response with the appropriate ***`custom_headers`***. By default, `487 Transaction Cancelled` will be used.

reason

`NOTIFY` requests always contain a `Subscription-State` header. When cancelling a

subscription, its value will be `terminated` and, by default, this header will contain a `reason=noresource` parameter. Set this field if it is required to change the default reason from `noresource`.

This field has no meaning for subscribers.

custom_headers
The application may supply additional SIP headers to be added to the outgoing `SUBSCRIBE` request. Note if multiple headers are being appended then `\r\n` should be used to delimit each header.

For example:

```
cancel.custom_headers = "Subject: The meeting";
cancel.custom_headers = "Subject: 10acb7899\r\nServer: VoIP server";
```

This field must be null terminated.

event_id_param
Reserved for future use.

message_bodies
Message bodies to be added to the outgoing message can be specified here. See section 4.1 for further details of the setup of this structure `ACU_RAW_MESSAGE_BODY`.

For example:

```
ACU_RAW_MESSAGE_BODY arm;
memset(&arm, 0, sizeof(ACU_RAW_MESSAGE_BODY));

// setup the message body structure - see sip_send_request() for further
details

cancel.message_bodies = &arm;
```

Return values

On successful completion, a value of zero is returned otherwise a negative value will be returned indicating the type of error.

Example usage

```
SIP_SUB_CANCEL_PARMS cancel;
INIT_ACU_STRUCT(&cancel);

cancel.handle = handle;

// This is not a mandatory field but, for notifiers, it is highly likely
// to be required by the specific event package being implemented to
// indicate the final state of the subscription.
ACU_RAW_MESSAGE_BODY arm;
memset(&arm, 0, sizeof(ACU_RAW_MESSAGE_BODY));

// setup the message body structure - see sip_send_request() for
// further details

cancel.message_bodies = &arm;

ACU_ERR rc = sip_sub_cancel(&cancel);
```

3.36 sip_sub_release() – release the internal resources associated with a subscription

This routine is similar to `call_release()` in that it is used to release the internal resources associated with a subscription. However, its usage is distinctly different as subscribers may have multiple dialogs associated with them. As such, this function may not be called until all dialogs have been cancelled. See `ACU_SUBSCRIPTION_INFO` for more details on when to call this function.

Synopsis

```
ACU_ERR sip_sub_release(SIP_SUB_RELEASE_PARMS* parms);


typedef struct sip_sub_release_parms
{
   ACU_ULONG          size;
   ACU_CALL_HANDLE    handle;
} SIP_SUB_RELEASE_PARMS;
```

Input parameters

This function takes a pointer to a `SIP_SUB_RELEASE_PARMS` structure. The structure should be initialised before invoking the function.

handle

The *`handle`* field identifies the subscription that is to be released. This field is mandatory.

Return values

On successful completion, a value of zero is returned otherwise a negative value will be returned indicating the type of error.

Example usage

```
SIP_SUB_RELEASE_PARMS release;
INIT_ACU_STRUCT(&release);

release.handle = handle;


ACU_ERR rc = sip_sub_release(&release);
```

3.37 sip_sub_fetch() – request an immediate fetch of subscription state

This routine allows a subscriber to request an immediate fetch of subscription state on one or all of its associated dialogs. It corresponds to a `SUBSCRIBE` request being sent with an `Expires` header that contains the remaining subscription duration.

Synopsis

```
ACU_ERR sip_sub_fetch(SIP_SUB_FETCH_PARMS* sip_sub_fetch_parms);


typedef struct sip_sub_fetch_parms
{
   ACU_ULONG               size;              /*IN*/
   ACU_CALL_HANDLE         handle;            /*IN*/
   ACU_INT                 dialog_id;         /*IN*/
   char*                   custom_headers;    /*IN*/
   char*                   event_id_param;    /*IN*/
   ACU_RAW_MESSAGE_BODY*   message_bodies;    /*IN*/
} SIP_SUB_FETCH_PARMS;
```

Input parameters

This function takes a pointer to a `SIP_SUB_FETCH_PARMS` structure. The structure should be initialised before invoking the function.

handle

The *`handle`* field identifies the subscriber that is to request the fetch of state. This field is mandatory.

dialog_id

This field specifies which dialog is requesting the fetch of state. A `SUBSCRIBE` will be sent on all associated dialogs if this field is left blank.

custom_headers

The application may supply additional SIP headers to be added to the outgoing `SUBSCRIBE` request. Note if multiple headers are being appended then `\r\n` should be used to delimit each header.

For example:

```
fetch.custom_headers = "Subject: The meeting";
fetch.custom_headers = "Subject: 10acb7899\r\nServer: VoIP server";
```

This field must be null terminated.

event_id_param

Reserved for future use.

message_bodies

Message bodies to be added to the outgoing message can be specified here. See section 4.1 for further details of the setup of this structure `ACU_RAW_MESSAGE_BODY`.

For example:

```
ACU_RAW_MESSAGE_BODY arm;
memset(&arm, 0, sizeof(ACU_RAW_MESSAGE_BODY));

// setup the message body structure - see sip_send_request() for further
details
```

```
fetch.message_bodies = &arm;
```

Return values

On successful completion, a value of zero is returned otherwise a negative value will be returned indicating the type of error.

Example usage

```
SIP_SUB_FETCH_PARMS fetch;
INIT_ACU_STRUCT(&fetch);

fetch.handle = handle;
fetch.dialog_id = 3;

ACU_ERR rc = sip_sub_fetch(&fetch);
```

3.38 sip_set_global_tos() – Change the ToS (Type of Service)

This routine allows an application to alter the DSCP and ECN values.

This routine is only applicable to Linux. For Windows please refer to Appendix F.

Synopsis

```
ACU_ERR sip_set_global_tos(SIP_SET_GLOBAL_TOS_PARMS* parms);


typedef struct tSIP_SET_GLOBAL_TOS_PARMS
{
   ACU_ULONG              size;              /*IN*/
   ACU_UCHAR              tos_value;         /*IN*/
} SIP_SET_GLOBAL_TOS_PARMS;
```

Input parameters

This function takes a pointer to a `SIP_SET_GLOBAL_TOS_PARMS` structure. The structure should be initialised before invoking the function.

tos_value

The *`tos_value`* field is an 8-bit value used to allow the network equipment to prioritise certain traffic at times of high load. The 6 MSB correspond to the DSCP (Differentiated Services Codepoint) value which determines the flow path of datagram packets. The 2 LSB correspond to the ECN (Explicit Congestion Notification) value which is usually assigned by the routers. When choosing a DSCP value the ECN value is usually set to “0”

An example for SIP signalling messages is a DSCP code **AF31**. This requires the ToS field to be set to 0x68 (hex) or 104 (decimal).

Return values

On successful completion, a value of zero is returned otherwise a negative value will be returned indicating the type of error.

Example usage

```
SIP_SET_GLOBAL_TOS_PARMS global_tos;
INIT_ACU_STRUCT(&global_tos);

global_tos.tos_value = 0x68;

ACU_ERR rc = sip_set_global_tos(&global_tos);
```

4 SIP specific structures

The following are helper structures that are used to assist composition of the top-level structures passed to the extended SIP API functions.

4.1 ACU_RAW_MESSAGE_BODY

This structure permits an application to specify a message body to be appended to an outbound message. The usage is primarily targeted at the relay of `isup` and `qsig` (see RFC 3204 for further information) data, though in theory any custom message could be sent. The application, however, is advised not to use this structure to send an SDP body, as this API has specialised structures (`ACU_MEDIA_OFFER_ANSWER`) for the transit of SDP.

NOTE

An instance of this structure may be used as an element in a linked-list, enabling the application to specify multiple messages in a single function call.

```
typedef struct acu_raw_message_body
{
  char*                 body_type;
  unsigned char*        body;
  ACU_INT               body_length;
  char*                 additional_body_headers;

  struct acu_raw_message_body* next;
} ACU_RAW_MESSAGE_BODY;
```

body_type

This is a `NULL`-terminated ASCII string specifying the message body type. This should be in the format expected by a SIP '`Content-Type`' header; '`Content-Type`' header parameters may also be supplied in this string.

For example:

```
arm.body_type = "application/isup";
arm.body_type = "application/QSIG; version=iso";
```

This string will be incorporated into the SIP '`Content-Type`' header, in the main SIP message if the resultant message is single-part or in the message body if the resultant message is multi-part.

body

This is a pointer to the message body to be sent. This is not `NULL`-terminated.

body_length

The length of the message pointed to by **body**.

additional_body_headers

A `NULL`-terminated string, specifying any message body additional headers. Currently only 'Content-Disposition' is supported.

For example:

```
arm.additional_body_headers  = "Content-Disposition: signal;
handling=optional";
```

(Beware of line-break in the above.)

next

If more than one message body is being specified here then this should point to the

next message. Otherwise set this to 0 to terminate the ‘list’.

For example:

```
// setup the first message body
ACU_RAW_MESSAGE_BODY message_body;
memset(&message_body, 0, sizeof(message_body));
message_body.body_type = "application/isup";

unsigned char BIN_BODY[] = {
0x81, 0x82, 0x1c, 0x05, 0xe4, 0x87, 0xe7, 0x86,
0xc8, 0x00, 0x01, 0x00, 0x00, 0x00, 0x0a, 0x00,
0x02, 0x07, 0x05, 0x04, 0x00, 0x21, 0x43, 0x65,
0x0a, 0x04, 0x84, 0x00, 0x32, 0x04 , 0x00};

message_body.body = (unsigned char*)BIN_BODY;
message_body.body_length = sizeof(BIN_BODY);

// setup a second message body

ACU_RAW_MESSAGE_BODY message_body2;
memset(&message_body2, 0, sizeof(message_body2));

message_body2.body_type = "custom-defined/message-type";

// setup other fields necessarily

// hook up the message bodies into a simple list
message_body.next = &message_body2;
```

4.2 ACU_IP_ADDRESS

This structure is used to offer flexibility regarding the setup of IP addresses.

```
typedef struct acu_ip_address
{
  unsigned char    address_type;
  char*            address;
} ACU_IP_ADDRESS;
```

address_type

This field is used to influence any IP version dependent treatment of this address by the stack. Most of the time this is transparent and an application may leave this field zeroed.

`address_type` may be chosen from the following enumeration:

```
typedef enum acu_ip_type
{
  ACU_IP_DEFAULT=0,
  ACU_IPv4=1,
  ACU_IPv6=2,
} ACU_IP_TYPE;
```

address

This is a `NULL`-terminated string containing the IP address either in dotted quad, `Ipv6` or fully qualified domain name format.

For example:

```
ACU_IP_ADDRESS ia1;
memset(&ia1, 0, sizeof(ACU_IP_ADDRESS));

ia1.address = "10.202.161.89";
```

And an IPv6 example:

```
ACU_IP_ADDRESS ia2;
memset(&ia2, 0, sizeof(ACU_IP_ADDRESS));

ia2.address_type = ACU_IPv6;
ia2.address = "2001:0DB8::1428:57ab";
```

And a FQDN example:

```
ACU_IP_ADDRESS ia3;
memset(&ia3, 0, sizeof(ACU_IP_ADDRESS));

ia3.address = "voipcard.domain.com";
```

4.3 ACU_PAYLOAD

This structure is used in order to specify a payload for the transfer of information in a media session. Examples of payload would include g.729 as an audio payload or T.38 as a fax/image payload. It is also possible to specify dynamic/bespoke payload definitions within this structure. Since SDP is the mechanism used by SIP endpoints to negotiate media settings, this structure is loosely based on the representation of a payload within SDP.

`ACU_PAYLOAD` structures represent different media types depending on which element of the payload union is used. These structures are always elements of the structure `ACU_MEDIA_DESCRIPTION`, which has a *`media_type`* element. It is important to ensure that the payload type described within a payload structure corresponds to the *`media_type`* of the containing `ACU_MEDIA_DESCRIPTION`. For example, for a linked list of *`audio_video`* payloads the containing `ACU_MEDIA_DESCRIPTION` structure should have *`media_type`* set as `ACU_AUDIO` or `ACU_VIDEO`.

```
typedef struct acu_payload
{
  union
  {
    struct
    {
      char*                                   rtp_payload_name;
      ACU_INT                                 rtp_payload_number;
      ACU_INT                                 packet_length;
      ACU_INT                                 clock_rate;
      char*                                   payload_specific_options;
    } audio_video;
    struct
    {
      char*                                   image_payload_name;
    } image;
    struct
    {
      ACU_INT                                 dummy;
    } control;
    struct
    {
      ACU_INT                                 dummy;
    } application;
  } payload;
  struct acu_payload*                       next;
} ACU_PAYLOAD;
```

NOTE

Additions to the ACU_PAYLOAD structure may be made in the future.

payload

The majority of this structure is within a union *`payload`* of structures, each internal structure describing a specific payload type. Here is a description of the currently implemented payload types available within this API.

The diversity of information configurable by SDP presents difficulties in modelling payload types within fixed size “C” style structures. The only payload type for which it is advantageous to employ a simple “C” style structure representing a subset of the payload parameters is the audio/video type. To configure non-audio/types the application writer is offered a more flexible scheme, that of “*miscellaneous_attributes*”. These permit configuration of various, unlimited, media description associated “a=” SDP parameters; T.38 fax and MRCPv2 payloads are

represented by the Aculab API accordingly. Refer to the ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE and ACU_MEDIA_DESCRIPTION for further details.

audio_video

Populate this element of the union in order to configure an audio/video payload e.g. g729 or telephone-event. A payload of this type would be a member of an `m=audio` or `m=video` stream in SDP. The definition contains the following elements:

rtp_payload_name

A `NULL`-terminated string, holding the name for the payload being described. This may be a string as referred to by RFC1890 e.g. `PCMU`, a well-known dynamic payload name e.g. 'telephone-event' or bespoke dynamic payload name e.g. 'VoIP-codec-123'. If this field is left as `NULL`, then `PCMU` will be assumed.

The API header file contains commonly used RTP payload names for reference:

For example:

```
#define ACU_PCMU "PCMU"
#define ACU_PCMA "PCMA"
#define ACU_G723 "G723"
#define ACU_G729 "G729"

#define ACU_TELEPHONE_EVENT "telephone-event"

// the RTP payload used for DTMF
#define ACU_DTMF_RTP_PAYLOAD ACU_TELEPHONE_EVENT
```

rtp_payload_number

A number, used in the RTP header to describe the payload type. This number may be chosen from RFC1890 or may be determined as required by a bespoke application.

The API header file contains commonly used RTP payload numbers for reference:

```
#define ACU_PCMU_PAYLOAD_NUMBER 0
#define ACU_PCMA_PAYLOAD_NUMBER 8
#define ACU_G723_PAYLOAD_NUMBER 4
#define ACU_G729_PAYLOAD_NUMBER 18
```

(Note that the 'telephone-event' payload type is dynamically assigned and hence a number is not suggested in Aculab's API.)

packet_length

The field has been included for future development when it will be possible to specify (within SDP) the packetisation of a payload in a payload specific way. Currently not supported by SDP.

clock_rate

This field specifies the clock rate used in the codec. If left as 0 the `sipserv` assumes the rate to be 8000Hz and writes this value into the SDP.

The API supplies a suitable constant for this value should applications wish to be explicit regarding this field.

```
enum ePAYLOAD_DEFINITIONS
{
  ACU_EIGHT_K = 8000,
};
```

payload_specific_options

Certain audio/video codecs have options that are specific to that codec. SDP uses the `fmtp` attribute to describe this. This field is a NULL-terminated string, which can be used to specify that codec's specific options.

For example:

"0-15,66,70" for telephone-event, this advertises the supported DTMF events. See RFC 2833 for more details on telephone-event.

"annexb=no" for g729, this advertises a lack of support for Annex B.

image

Populate this element of the union in order to configure an image or FAX payload e.g. T38 over UDP. A payload of this type would be a member of an m=image stream in SDP. It contains the following the elements:

image_payload_name

This is a NULL-terminated string which specifies the name of the image payload.

For example:

```
#define ACU_T38 "t38"
```

control

This element of the union has been provided for future development. It is envisaged that the payload types of m=control streams will be described here, e.g. MRCP, 'whiteboard' etc.

application

This element of the union has been provided for future development.

The ACU_PAYLOAD structure may be an element in a list. This is enabled by one element being a pointer to another ACU_PAYLOAD, as follows:

next

This is a pointer to the next element in a list of ACU_PAYLOAD structures, or 0 if this structure is the last/only element in the list.

For example, configuring a g729 and g711 codec and linking the definitions together

```
ACU_PAYLOAD payload1;
memset(&payload1, 0, sizeof(ACU_PAYLOAD));
ACU_PAYLOAD payload2;
memset(&payload2, 0, sizeof(ACU_PAYLOAD));

payload1.payload.audio_video.rtp_payload_name="g729";
payload1.payload.audio_video.rtp_payload_number=18;
payload1.payload.audio_video.clock_rate=8000;
payload1.payload.audio_video.payload_specific_options="annexb=no";
payload1.next = &payload2; // link payloads

payload2.payload.audio_video.rtp_payload_name="PMCU";
payload2.payload.audio_video.clock_rate=8000;
```

For example, configuring a t38 over UDP payload

```
ACU_PAYLOAD payloadi;
memset(&payloadi, 0, sizeof(ACU_PAYLOAD));

payloadi.payload.image.image_payload_name="t38";

// T.38 attributes configured using ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE
// see below
```

4.4 ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE

All of the commonly used media attributes and properties of SDP are represented by the other structures described by this document. However, additions are frequently made to SDP to cater for new requirements, for example, those facilitating NAT traversal, MRCPv2, and T.38 configurations, specifications of which may be achieved using this function. Session level attributes may also be provided using this structure, when supplied in a media description with the type `ACU_SESSION`.

A list of attribute strings is included in the API to store any additional miscellaneous SDP media elements in a generic fashion. Media attributes, which are not conveyed by a specific element in this API, may be given here.

```
typedef struct acu_miscellaneous_media_attribute
{
  char* attribute;
  struct acu_miscellaneous_media_attribute* next;
} ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE;
```

attribute

This is a NULL-terminated string where the attribute is specified. Note that the `a=` specifier of the SDP line is not included, for example the silence suppression line `a=silenceSupp:on - - - -` would be represented as:

```
  ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE media_attrib;
  memset(&media_attrib, 0, sizeof(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE));

  media_attrib.attribute = "silenceSupp:on - - - -";
```

next

This is a pointer to the next `ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE`.

For example, set this endpoint as being `active` in an asymmetric exchange:

```
  ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE media_attrib;
  memset(&media_attrib, 0, sizeof(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE));

  media_attrib.attribute = "direction:active";
```

For example, set this endpoint as being `active` in an asymmetric exchange and indicating silence suppression (as specified in RFC 3108).

```
  ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE media_attrib, media_attrib2;
  memset(&media_attrib, 0, sizeof(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE));
  memset(&media_attrib2, 0, sizeof(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE));

  media_attrib.attribute = "direction:active";
  media_attrib.next = &media_attrib2;

  media_attrib2.attribute = "silenceSupp:on - - - -";
```

4.5 ACU_MEDIA_DESCRIPTION

This structure is used to represent a media description. This may be offered (proposed) by one party in a SIP session or may comprise an answer as provided by the alternate party. The structure attempts to encapsulate the information conveyed in an SDP body by the `m=` line and associated (optional) `a=` and `c=` lines. Facilitated by the `next` field it is quite possible for an instance of this structure to be an element in a list of these structures.

```
typedef struct acu_media_description
{
  ACU_IP_ADDRESS                        connection_address;
  ACU_USHORT                            port;
  unsigned char                         media_direction;
  unsigned char                         media_type;
  char*                                 transport;
  ACU_INT                               packet_length;
  ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE*    miscellaneous_attributes;
  ACU_PAYLOAD*                          payloads;

  struct acu_media_description*         next;
} ACU_MEDIA_DESCRIPTION;
```

connection_address

The ‘media description specific’ connection address is placed here. It is the address in which the endpoint wishes to have media sent. It is optional to have the field here, for most applications; it is sufficient to supply the connection address from within the `acu_media_offer_answer` structure instead. If unused, the field should be left `memset` to zero. If this field is used then a `c=` line is written to the media description in the SDP body. If used for IPv4 the `address_type` should be set to `ACU_IPv4`. For IPv6 the field should be set to `ACU_IPv6`. When creating an answer the ‘media description specific’ `address_type` should match the respective `address_type` of the offer otherwise the SDP will be considered invalid (i.e. don’t mix `c=` lines with IP4 and IP6).

port

This field is the port number in which the endpoint would like to receive the media at. It should be a valid port number unless the endpoint wishes to delete/decline a stream in which case it should be 0. If the application is using `c="0.0.0.0"` as the connection address then the port must be specified as any non-zero number.

media_direction

This field represents the direction/activity attribute, which may be present in an SDP definition. It appears in the SDP body in the form of `a=sendonly`, `a=inactive`. Please refer to RFC 2327 for further information on these attributes. The API header file holds an enumeration that represents these attributes:

```
typedef enum acu_media_direction
{
  ACU_SEND_RECV,
  ACU_SEND_ONLY,
  ACU_RECV_ONLY,
  ACU_INACTIVE
} ACU_MEDIA_DIRECTION;
```

NOTE

The default setting in both SDP and this API is **`sendrecv`**.

media_type

This field specifies the type of media session in which this SIP call is trying to setup. This choice is reflected in the SDP by the first string in the `m=` line, e.g. `m=audio`. The enumeration below is present within the API header and lists the range of choices for this field. The `ACU_SESSION` type is provided as a means to set session level attributes for the SDP body. An `ACU_SESSION` media description may contain a connection address as well as miscellaneous attributes. The `ACU_SESSION` media description must be provided as the first media description in the list. If the application wishes to retrieve session level information then the `ACU_USE_MEDIA_DESCRIPTION_FOR_SESSION` option must be provided in the call to `sip_openin()` or `sip_openout()`.

```
typedef enum acu_media_types
{
  ACU_AUDIO,
  ACU_VIDEO,
  ACU_IMAGE,
  ACU_CONTROL,
  ACU_TEXT,
  ACU_APPLICATION,
  ACU_UNKNOWN_MEDIA_TYPE,
  ACU_SESSION
} ACU_MEDIA_TYPES;
```

NOTE

Only audio, video, image, text, session and application types are currently supported.

transport

A NULL-terminated string, specifying the transport mechanism employed by the media description negotiated. When not supplied by the developer the SIP service will choose a sensible default, for example, for audio/video media types: `RTP/AVP`, for image (fax) media types: `udptl`.

For example, configuring transport for audio media types such as g711, g729

```
ACU_MEDIA_DESCRIPTION md;
memset(&md, 0, sizeof(ACU_MEDIA_DESCRIPTION));

md.transport = 0;

// SIP service defaults this to "RTP/AVP"
```

For example, configuring transport for an MRCPv2 media description

```
ACU_MEDIA_DESCRIPTION md;
memset(&md, 0, sizeof(ACU_MEDIA_DESCRIPTION));

md.transport = "TCP/MRCPv2";
```

packet_length

This field enables a non-default packetisation interval to be specified for all the payloads present in the media description, as a number in milliseconds. This maps to the `a=ptime:` attribute in the SDP body. In order that default packet lengths are used, this field should be zero.

miscellaneous_attributes

This field specifies a list of miscellaneous media attributes, which do not have an alternative logical definition within this API. Any media related additions to SDP or similar esoteric options, which cannot be easily categorised, maybe specified in a generic fashion. See the section `ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE` for further details.

payloads
A pointer to a list of payloads in this media description.

next
A pointer to the next media description in the list or 0 to indicate termination of the list.

For example, configuring an audio media description and a T38 media description and linking them together

```
ACU_MEDIA_DESCRIPTION md1;
memset(&md1, 0, sizeof(ACU_MEDIA_DESCRIPTION));
ACU_MEDIA_DESCRIPTION md2;
memset(&md2, 0, sizeof(ACU_MEDIA_DESCRIPTION));

// set up the first media description
md1.port = 4088;
md1.connection_address.address="10.202.10.17";

md1.media_direction=ACU_SEND_ONLY;
md1.media_type=ACU_AUDIO;
md1.packet_length=40;

ACU_PAYLOAD payload1;
memset(&payload1, 0, sizeof(ACU_PAYLOAD));
ACU_PAYLOAD payload2;
memset(&payload2, 0, sizeof(ACU_PAYLOAD));

payload1.payload.audio_video.rtp_payload_name="g729";
payload1.payload.audio_video.rtp_payload_number=18;
payload1.payload.audio_video.clock_rate=8000;
payload1.payload.audio_video.payload_specific_options="annexb=no";
payload1.next = &payload2;

payload2.payload.audio_video.rtp_payload_name="PMCU";
payload2.payload.audio_video.clock_rate=8000;

md1.payloads = &payload1;
md1.next = &md2; // link media descriptions

// set up the second media description
md2.port = 4092;
md2.media_type=ACU_IMAGE;

ACU_PAYLOAD payloadi;
memset(&payloadi, 0, sizeof(ACU_PAYLOAD));

payloadi.payload.image.image_payload_name="t38";

md2.payloads = &payloadi;

ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE mma1,mma2,mma3,mma4;

memset(&mma1, 0, sizeof(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE));
memset(&mma2, 0, sizeof(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE));
memset(&mma3, 0, sizeof(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE));
memset(&mma4, 0, sizeof(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE));

mma1.attribute="T38MaxBitRate:512”;

mma2.attribute="T38FaxRateManagement:localTCF”;

mma3.attribute="T38FaxUdpEC:t38UDPFEC”;

mma4.attribute="T38FaxMaxDatagram:1024”;


mma1.next=&mma2;
mma2.next=&mma3;
mma3.next=&mma4;

md2.miscellaneous_attributes=&mma1;
```

For example, configuring an audio media description and an MRCPv2 media description and linking them together

```
ACU_MEDIA_DESCRIPTION md1;
memset(&md1, 0, sizeof(ACU_MEDIA_DESCRIPTION));
ACU_MEDIA_DESCRIPTION md2;
memset(&md2, 0, sizeof(ACU_MEDIA_DESCRIPTION));

// set up the first media description
md1.port = 4088;
md1.connection_address.address="10.202.10.17";

md1.media_direction=ACU_SEND_ONLY;
md1.media_type=ACU_AUDIO;
md1.packet_length=40;

ACU_PAYLOAD payload1;
memset(&payload1, 0, sizeof(ACU_PAYLOAD));
ACU_PAYLOAD payload2;
memset(&payload2, 0, sizeof(ACU_PAYLOAD));

payload1.payload.audio_video.rtp_payload_name="g729";
payload1.payload.audio_video.rtp_payload_number=18;
payload1.payload.audio_video.clock_rate=8000;
payload1.payload.audio_video.payload_specific_options="annexb=no";
payload1.next = &payload2;

payload2.payload.audio_video.rtp_payload_name="PMCU";
payload2.payload.audio_video.clock_rate=8000;

md1.payloads = &payload1;
md1.next = &md2; // link media descriptions

// set up the second media description
md2.port = 4092;
md2.media_type=ACU_APPLICATION;

              md2.transport= "TCP/MRCPv2";

              ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE mma1,mma2,mma3,mma4;

              memset(&mma1, 0, sizeof(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE));
              memset(&mma2, 0, sizeof(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE));
              memset(&mma3, 0, sizeof(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE));
              memset(&mma4, 0, sizeof(ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE));

              mma1.attribute="setup:active";
              mma2.attribute="connection:new";
              mma3.attribute="resource:speechsynth";
              mma4.attribute="cmid:1";

              mma1.next=&mma2;
              mma2.next=&mma3;
              mma3.next=&mma4;

              md2.miscellaneous_attributes=&mma1;
```

4.6 ACU_MEDIA_OFFER_ANSWER

This structure will contain either an offer or answer received in a SIP message or an offer or answer to be sent out to a remote party. This `offer/answer` is conveyed by SIP as an SDP body and the role of the `ACU_MEDIA_OFFER_ANSWER` is to encapsulate information relevant to the composition of an SDP body. RFC 3264 contains guidelines useful in the configuration of SDP bodies for SIP. An application will need to populate structures of this type prior to calling various SIP call control functions, for example `sip_openout()`; it will also be presented these structures by `sip_details()`.

The information held in this structure is available in two conceptually different forms: abstracted and raw.

The abstracted form is presented by the *`media_descriptions`* field which represents the `m=` line (or potentially a list of these) and its associated `a=` lines and `c=` line and the `connection_address` field which specifies the session global `c= line`. A representation of the `v=, o=, s=` and `t=` lines mandatory to an SDP body is omitted from the abstracted form, it is viewed that these may be satisfactorily deduced by the SIP service without input from the application, as those fields have no effect on the media negotiation effected by the SDP body. If global session attributes are required then these can be supplied by supplying a media description with the type `ACU_SESSION` as the first entry in *`media_descriptions`*.

The raw form of SDP is available, in its entirety, in the field *`raw_sdp`* – if it is an offer, all lines must be included.

```
typedef struct acu_media_offer_answer
{
  ACU_IP_ADDRESS              connection_address;
  ACU_MEDIA_DESCRIPTION*      media_descriptions;
  char*                       raw_sdp;
} ACU_MEDIA_OFFER_ANSWER;
```

When the `ACU_MEDIA_OFFER_ANSWER` is used to convey information received in an inbound SIP message both the raw and abstracted forms are present in the structure. However when the structure is being used to configure an outbound SIP message then it is the application writer's decision as to whether or not to use the raw or abstracted form. In most situations it will be adequate to use the abstracted form; it is certainly easier to manipulate than the raw form. However, if the writer wishes to use an SDP attribute absent from the Aculab API's abstraction then it is possible to effect this by the writer supplying an entire well-formed SDP body as the argument to the *`raw_sdp`* field. The SIP service will ignore the fields comprising the abstracted form in circumstance of the *`raw_sdp`* field being not `NULL`.

connection_address

The session global connection address should be entered here. In the SDP this will appear in the `c=` line outside of the media descriptions. For IPv4 the `address_type` should be set to `ACU_IPv4`. For IPv6 the field should be set to `ACU_IPv6`. If the application wishes to present a 'black-hole' SDP address, for IPv4 they should set the IP address within this field to `0.0.0.0`. For IPv6 the field should be set to a domain within the .invalid top level DNS domain (not `::`), for example `sip.invalid`. When creating an answer the `address_type` should match the `address_type` of the offer otherwise the SDP will be considered invalid (i.e. don't mix `c=` lines with IP4 and IP6).

media_descriptions

A pointer to a `ACU_MEDIA_DESCRIPTION` structure. This pointer could be: the first element in a list of such structures, which would imply that the SDP body has multiple `m=` lines; a pointer of a single structure, which is the single `m=` line `case`; or `NULL` which

implies that the SDP has no associated media lines.

raw_sdp

A pointer to a `NULL`-terminated string assumed to contain an entire SDP body. If this is present then the SIP Service will ignore the other fields in the structure.

For example, application defines the SDP used for a `sip_openout()` using the abstracted mode of the structure:

```
ACU_MEDIA_OFFER_ANSWER mo1;
memset(&mo1, 0, sizeof(ACU_MEDIA_OFFER_ANSWER));

mo1.connection_address.address="10.202.10.16";

ACU_MEDIA_DESCRIPTION md1;
memset(&md1, 0, sizeof(ACU_MEDIA_DESCRIPTION));

// refer the above sections to configure the ACU_MEDIA_DESCRIPTION
// structure(s).

mo1.media_descriptions = &md1;
```

See Appendix B for further information on using raw SDP.

4.7 ACU_MEDIA_SESSION

An instance of ACU_MEDIA_SESSION will be queued for a call in the SIP service whenever an offer answer exchange of SDP bodies has been effected by the SIP call. An offer answer exchange is complete after one party has sent some initial/new SDP [1] and the other party has sent its SDP [2] – which is compatible with SDP [1]. The structure's presence is flagged by an EV_MEDIA event being raised to the application, calling sip_details() at this point will collect this structure for the application to inspect.

The purpose of this structure is to convey the SDP bodies, which have been used in a media negotiation – the sent and received SDP being presented. It is very important for an application to process this structure when flagged by the EV_MEDIA event as it signifies the opportunity in which an application should start, stop or re-configure media streams.

```
typedef struct acu_media_session
{
  ACU_MEDIA_OFFER_ANSWER     sent_media;
  ACU_MEDIA_OFFER_ANSWER     received_media;
  char                       sent_is_answer;
} ACU_MEDIA_SESSION;
```

sent_media

This field holds the SDP body, which was sent by this call in the most recent offer answer exchange. In a simple call, that is, one without any third party call control, the field can also be perceived as the local SDP settings, regarding IP address/port details, payload choices and so on. However, in the case of third party call control when this party may be acting as a controller between two other parties, this field is just the sent SDP. It is no longer to be considered local to this call, as this party merely controls the media streams, it does not participate in them.

NOTE

If an application, when populating an **ACU_MEDIA_OFFER_ANSWER** structure prior to transmission, leaves any fields unset, that is assumes use of the SIP service's default settings, then those fields will appear unset in the ***sent_media*** element of the **ACU_MEDIA_SESSION** structure. This is due to a considered optimisation in which these fields are not converted to their real values until the preparation of the raw SDP body.

received_media

This field holds the SDP body, which was received by this call in the most recent offer answer exchange. In a simple call, that is, one without any third party call control, the field can also be perceived as the remote SDP settings regarding IP address/port details, payload choices and so on. In a simple call setup, this field contains the other party's IP address/port details to which the application should be sending its media. However, in the case of third party call control when this party may be acting as a controller between two other parties, this field is just the `received` SDP. It is no longer to be considered remote *only* to this call as this party merely controls the media streams, it does not participate in them.

sent_is_answer

The above fields contain SDP bodies as sent and received during a call in an offer answer exchange. Though the received media field holds the IP details of the destination of egress media packets, in a simple call it is the answer SDP, which specifies the payloads to be used for the media session. Hence, this flag is required in order that the application may determine these payloads with ease.

4.8 ACU_SIP_MESSAGE

This structure is used to represent a SIP message received from the IP network by the application. The presence of a SIP message, queued by the SIP service and available for collection is flagged by the raising of `EV_DETAILS`. The message may be collected by `sip_details()`.

```
typedef struct acu_sip_message
{
  unsigned char*        message;
  ACU_INT               message_length;
  ACU_UINT              message_handle;
} ACU_SIP_MESSAGE;
```

message
This points to memory holding an inbound SIP message.

message_length
The size of the above message in bytes is held here.

message_handle
Reserved for future use

4.9 ACU_STRING_LIST

This structure provides a basic list of NULL terminated strings.

```
typedef struct _ACU_STRING_LIST
{
  char*                         string;
  struct _ACU_STRING_LIST*      next;
} ACU_STRING_LIST;
```

string
The NULL terminated string for the current element in the list.

next
A pointer to the next string in the list, or 0 to indicate termination of the list.

4.10 ACU_REDIRECT_INFO

This structure provides `sip_details` with the contact details and SIP response code contained in any 3xx response.

```
typedef struct
{
  int                 sip_response_code;
  ACU_STRING_LIST*    contact_list;
} ACU_REDIRECT_INFO;
```

sip_response_code
The SIP response code. This will always be a 3xx response.

contact_list
A linked list of NULL terminated strings containing the Contact headers contained in the 3xx response.

4.11 ACU_SUBSCRIPTION_INFO

This structure, in conjunction with ACU_SIP_MESSAGE and ACU_REDIRECT_INFO provides sip_details() with subscription related information.

```
typedef struct _ACU_SUBSCRIPTION_INFO
{
   ACU_INT            expires;
   ACU_INT            dialog_count;
   ACU_INT            dialog_id;
   ACU_INT            notify_required;
   ACU_STRING_LIST*   body_types;
   char*              event_id_param;
} ACU_SUBSCRIPTION_INFO;
```

expires

In general, this field represents the remaining seconds left on the subscription with the following exception. At EV_SIP_SUBSCRIPTION_REQUEST the ***expires*** parameter corresponds to the Expires header contained in the initial SUBSCRIBE request which is the subscriber requested duration of the subscription in seconds. An application wishing to lower this duration must specify an alternative ***expires*** value in a call to sip_sub_accept(). Note that ERR_PARM will be returned should any attempt be made to increase this value.

dialog_count

Notifiers will only ever be associated with a single SIP dialog. As such, this value will always be 1 until an EV_SIP_SUBSCRIPTION_CANCELLED event is raised at which point it will be set to 0.

Subscribers, however, may be associated with multiple dialogs if the initial SUBSCRIBE request forked in a proxy. This field will contain the current number of dialogs associated with the subscription. Because of this, EV_SIP_SUBSCRIPTION_CANCELLED may be raised many times on a single subscription, each time lowering this field by 1.

The application should call sip_sub_release() when this count reaches 0. ERR_COMMAND will be returned if this is attempted beforehand.

dialog_id

For notifiers, this field will always be set to 1 as there will only ever be a single dialog associated with the subscription.

For subscribers, each successive successful response to the initial SUBSCRIBE will create a new dialog and associate a ***dialog_id*** with it. This allows the application to know which dialog has been notified or cancelled. It also allows the application cancel specific dialogs within a subscription as some event packages do not allow multiple dialogs to be created.

notify_required
This field has no meaning for subscribers and will always be set to 0.

For notifiers, some events require the application to call `sip_sub_notify()` in order to conform to RFC 3265. This field will always be set after the events `EV_SIP_SUBSCRIPTION_PENDING` and `EV_SIP_SUBSCRIPTION_REFRESH` are raised. At `EV_SIP_SUBSCRIPTION_CANCELLED` this field will be set if the underlying cause was the receipt of a `SUBSCRIBE` request with an `expires` value of 0. At `EV_SIP_SUBSCRIBED` this field will be set if the subscription was not previously in a pending state.

body_types
This field only applies to notifiers and will only ever be populated at `EV_SIP_SUBSCRIPTION_REQUEST`. It denotes the formats of the message bodies that the subscriber is able to accept. These formats will be defined by the individual event package being implemented.

event_id_param
Reserved for future use.

5 Dual Redundant SIP Service (DRSS)

5.1 Description

The SIP service may be deployed on two separate servers that act to provide a resilient system that can reduce interruptions to applications if hardware or network issues affect one of the servers.

The resilient SIP service consists of a pair of separate servers configured to share a well-known address to the outside world for SIP traffic. One of these servers will process call control traffic, known as the active server and the other server that is known as the passive mirrors the state of calls on the active, using a separate connection. When a problem occurs on the active the passive performs a takeover (whereby it assumes the role of the active) and manages all existing connected calls, indicating to the application that a takeover has occurred.

Extra functionality provides applications with the means to query the existing state of the resilient system and also with the means to force a takeover, for maintenance purposes.

Extra configuration options need to be set for the applications and the individual sip servers to ensure that the system operates correctly.

A pre-existing application does not need to be modified to take advantage of the resiliency provided by these features. However, some new events and routines are provided to make management of the system easier.

5.2 Terms used during this section

Active server

This is one of the servers that in combination with a passive server will provide the resilient SIP service. The active server will handle incoming and outgoing calls while providing the passive server with data that would allow the passive server to takeover in the event of a problem. The active server will advertise itself as the owner of the floating IP address, meaning SIP traffic will be directed towards it.

Passive server

This server builds a mirror of the current state of the active server allowing it to takeover in the event of an error. The passive server will only handle SIP traffic during maintenance.

Floating IP address

The floating IP address acts as the public address for the resilient SIP service. The active server will configure one of its interfaces to use this as a virtual address or alias. The system uses ARP to advertise the current owner of the floating IP address to ensure that traffic reaches the correct destination.

Takeover

This is the process that results in the passive server becoming an active server when the system discovers a problem with the current active server. The passive takes on the floating IP address and advertises that it is now the owner of that IP address. It then becomes the active server. When the former active server is no longer in an error condition then that server will become a passive server, meaning that their roles have switched.

Maintenance

When it is necessary to modify the current active server in some way the active server can be shut down gracefully allowing calls to migrate to the passive server. This process is called maintenance and is invoked by the user.

5.3 Pre-requisites, restrictions and usage information

- See the current release notes for information on supported operating systems and variants.
- It is very strongly recommended that each server used to host the resilient sip service use at least two separate network interface cards, one for communication between the application and the SIP service and another for communication between the active and the passive server.
- DHCP is not supported. Each server must be configured with static IP addresses.
- It is strongly recommended that applications reside on separate servers to those used by the SIP servers. Each application must be configured to use the same DRSS servers in the same order.
- In the event of a failure resulting in a takeover calls that had been connected will be available after the takeover. In addition if an incoming call had successfully sent a 180 or 183 response then that call will be available. If an outgoing call had received a 180 or 183 then it will be available.
- Calls associated with application failover using call_get_failover_id() will only be recoverable after at most one takeover. If two or more takeovers occur it will not be possible to reopen the call.
- If media resources are located on the same servers used for the resilient SIP service then adequate arrangements need to be made to ensure that the media connections are maintained after a takeover. These arrangements are outside the scope of the functionality provided by the resilient SIP service. However API functionality is provided to provide a means of indicating that resources have moved, see sip_target_refresh().
- Use of DRSS functionality requires the installation of a licence on each server.
- In the event of a takeover the passive server that takes over will mirror the queue of sip_details() information for every connected call. It is possible that the failure that caused the takeover may result in the new active server having fewer items in the queue than the application was expecting. If this occurs the application may receive a SIP_DETAILS_PARMS structure which has not supplied any information in the sip_message, media_offer_answer or media_session fields, or has supplied a different field to that which the application might have been expecting. It is recommended that these fields are tested for NULL on a successful return from sip_details before attempting to access them.
- Using the Generic Call Control API (also referred to as the Media Handler Plugin or MHP) with DRSS is not supported.

5.4 How it works

Every application is configured to connect to the same pair of SIP servers. These SIP servers are in turn configured to accept connections from remote applications. The SIP servers are also configured to know about their corresponding peer SIP server and how to connect to each other. In addition each SIP server knows the well-known IP address, known as the 'Floating IP Address'. This Floating IP address must be the only IP address used to deliver normal SIP traffic to the server. The IP addresses of the individual SIP servers will be used during managed maintenance.

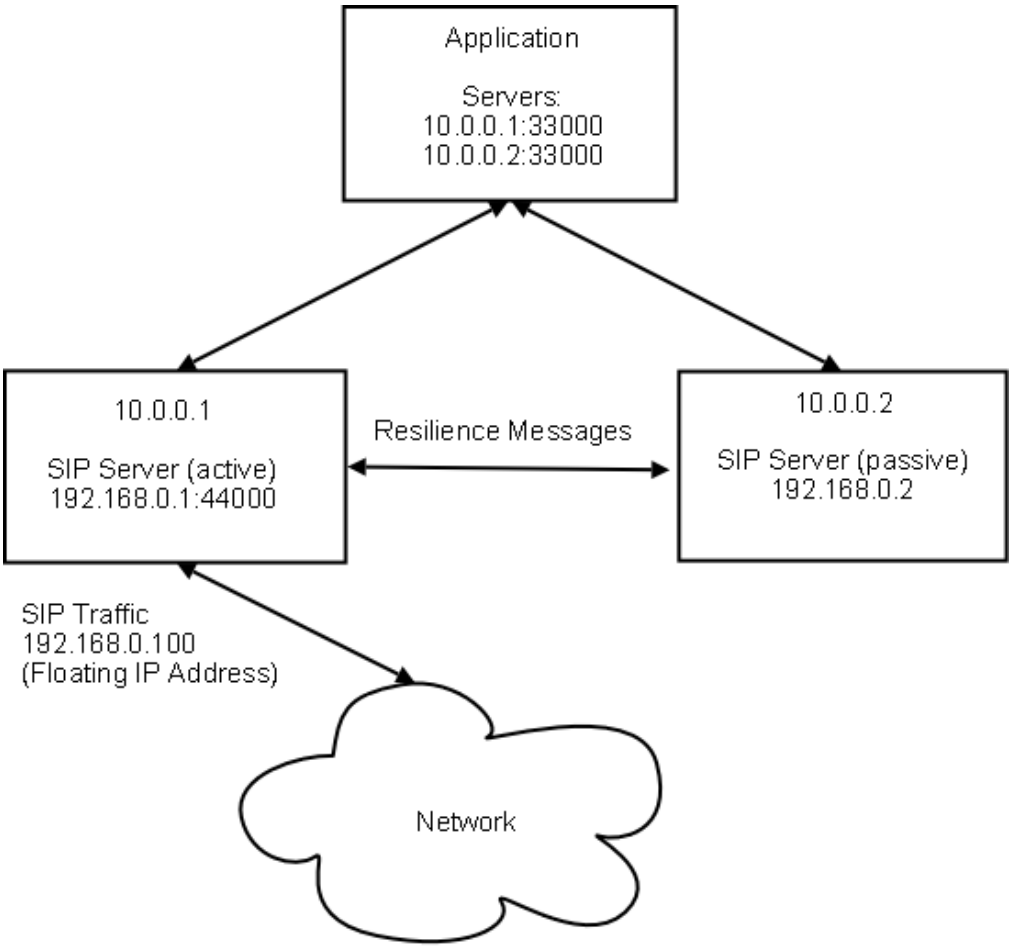
When the application starts the SIP servers will be designated either the active role or the passive role. The active server will configure the Floating IP address and advertise that it owns this address. The passive server will wait for messages from the active server that allow it to act as a replacement for the active server in the event of a failure.

During normal operation the active server will send messages to the passive server that allow the state of calls to be mirrored on the passive server. In the event of a failure the passive server will be instructed to assume the active role and it will configure the Floating IP address and resume call processing for existing calls and new traffic.

If a server loses its connection to the application (or all applications if multiple applications are present) then it will revert to an un-configured state.

It is also possible for an application to force a takeover to occur using maintenance commands allowing a server to be stopped gracefully.

5.4.1 Example


Application
Servers:
10.0.0.1:33000
10.0.0.2:33000
10.0.0.1
SIP Server (active)
192.168.0.1:44000
Resilience Messages
10.0.0.2
SIP Server (passive)
192.168.0.2
SIP Traffic
192.168.0.100
(Floating IP Address)
Network

5.5 Configuration

5.5.1 Application Configuration

Each application must be configured identically; otherwise the system will not work correctly.

The application must configure both SIP servers in `sipplugin.cfg`. This configuration is described in Distributed SIP Settings. Where multiple applications are used on different servers the order for the SIP servers should be the same for each `sipplugin.cfg`.

For the example above `sipplugin.cfg` should be as follows, assuming port 33300 is being used for IPC on each server :

```
RemoteIPCAddress = 10.0.0.1:33300
RemoteIPCAddress = 10.0.0.2:33300
```

5.5.2 SIP Server Configuration

The following options apply to the sipserv.cfg configuration file as described in Configuring the SIP service.

5.5.2.1 IPC Configuration

The SIP server must be configured to listen for connections from remote applications, as described in Distributed SIP Settings, and these settings must correspond with the settings that have been chosen for the application.

5.5.2.2 Fault Tolerance Configuration Options

The following options must be configured to ensure that communication between the SIP servers will work correctly. All parameters are mandatory unless stated otherwise.

`EnableFT = <0 or 1>`
This value must be set to 1 in order for the SIP service to run in resilient mode.

`FTPort = <n>`
This value must specify an IP port number. It should be greater than 0 and less than 65535, and should not be already in use by the operating system. It is also recommended that you avoid IP ports that could already be in use by other servers, for example, 0-1023, which are assigned by the Internet Assigned Numbers Authority (IANA), and registered ports 1024 to 49151.

The SIP service will use this port to listen for resilience messages from its peer.

`FTPeerPort = <n>`
This optional value is used to specify the port that the peer server is listening on. If it is not present, both servers must have `FTPort` configured to the same value.

`FTPeerInterface = <IP address>`
This value must be an IP address resident on the host in which the `SIP service` is deployed.

The indicates to the SIP server which NIC to use to communicate with the peer SIP server.

`FTPeer =  <IP address>`
This value must be the IP address that the peer SIP server uses for it's inter-service communication.

`FTPeerFloatingInterface = <IP address>`
This value must be the IP address that the peer SIP server uses as its floating

interface. This is required so that the active server may redirect calls to the correct address during a period of maintenance.

`FTFloatingInterface = <IP address>`
This value must be an IP address resident on the host in which the `SIP server` is deployed.

This indicates to the SIP server which NIC to use to configure the floating IP address.

`FTFloatingAddress = <IP address>`
This value must be an IP address that will be used as the well-known IP address to which SIP traffic will be directed.

`FTRole = <active|passive>`
If a pair of SIP servers are to be used in a resilient configuration one of them must be set to active and one to passive. This role does not determine their initial configuration that is designated when the application connects to the SIP server. It is used instead to act as a 'tie-breaker' in the event of a race condition.

Example Configuration for Server 10.0.0.1

Assuming 2 NICs 192.168.0.1, 10.0.0.1

```
FTPort = 44400
FTPeerInterface = 10.0.0.1
FTPeer = 10.0.0.2
FTFloatingInterface = 192.168.0.1
FTPeerFloatingInterface = 192.168.0.2
FTFloatingAddress = 192.168.0.100
FTRole = active
```

Example Configuration for Server 10.0.0.2

Assuming 2 NICs 192.168.0.2, 10.0.0.2

```
FTPort = 44400
FTPeerInterface = 10.0.0.2
FTPeer = 10.0.0.1
FTFloatingInterface = 192.168.0.2
FTPeerFloatingInterface = 192.168.0.1
FTFloatingAddress = 192.168.0.100
FTRole = passive
```

In this example, the heart beat exchange and the persistence of call state between the two servers takes place between 10.0.0.1:44400 and 10.0.0.2:44400. When active, a server will configure 192.168.0.100 (the floating IP address) as an additional address on the floating interface. SIP signalling will take place on 192.168.0.100 in normal operation and during periods of maintenance, calls will be redirected to 192.168.0.1 or 192.168.0.2 depending on which server is active when the maintenance starts.

The FTRole options will only be used in the event that more than one application connect to the servers at the same time and the period of negotiation fails to determine which server should take on the active role. This is only possible if the applications are configured to connect to the servers in a different order.

5.6 API For Resilient SIP Service

5.6.1 sip_rss_get_server_details() – get resilient server status

This function is used to obtain the current state of the SIP servers that make up the resilient SIP service. This routine may called at any time.

Synopsis

```
ACU_ERR sip_rss_get_server_details(SIP_RSS_SERVER_DETAILS_PARMS*
sip_server_details_parms);

   typedef struct sip_rss_server_detail_parms
   {
        ACU_ULONG                  size;                 /* IN */
        RSS_SERVER_DETAILS*        server_details;       /* OUT */
   } SIP_RSS_SERVER_DETAILS_PARMS;

   typedef struct rss_server_details
   {
        ACU_CHAR                   address[ACU_MAX_IP_ADDRESS];    /* OUT */
        ACU_USHORT                 port;                           /* OUT */
        ACU_UINT                   state;                          /* OUT */
        ACU_CHAR                   heartbeat_present;              /* OUT */
        ACU_CHAR                   server_alive;                   /* OUT */
        ACU_UINT                   calls;                          /* OUT */
        ACU_UINT                   error;                          /* OUT */
        struct rss_server_details* next;                           /* OUT */
   } RSS_SERVER_DETAILS;
```

Input parameters

The `sip_rss_get_server_details()` function takes a pointer, *`sip_server_details_parms`*, to a structure `SIP_RSS_SERVER_DETAIL_PARMS`. Before invocation, the structure must be cleared using `INIT_ACU_STRUCT`.

server_details

The *`server_details`* field is a pointer to a `RSS_SERVER_DETAILS` structure. This structure contains the following fields:

address

This field contains the IP address of a SIP server, which is used to differentiate between the servers forming the resilient SIP service.

port

This field contains the port number that the SIP server is listening on.

state

This field holds a value reflecting the current 'fault tolerant' state for this server. This value will be one of the following:

RSS_NONE	not capable of resilience
RSS_NOT_CONFIGURED	not configured
RSS_PASSIVE	working as a passive server
RSS_ACTIVE	working as an active server
RSS_FAILED	currently not working correctly due to error

heartbeat_present

A value of 1 indicates that this server is receiving valid 'heartbeat' messages from its peer. These messages act as a means of determining the health of the connection between the two servers. If there is a problem with these messages then a value of 0 will be returned in this field.

server_alive

A value of 1 indicates that this application is able to communicate correctly with this

server. A value of 0 indicates that there is a problem with communicating with this server.

calls
For an active server this field contains the number of calls currently managed by this server.

error
a value of 1 indicates that the system is not operating correctly, possibly due to configuration problems or connection issues. This field will be set for all servers if an error is detected.

next
This field contains a pointer to the next set of server details. A NULL value indicates that no more details are available.

Return values

On successful completion, a value of zero is returned; otherwise, a negative value will be returned indicating the type of error.

5.6.2 sip_rss_maintenance() – initiate a takeover from an application

This function may be used by an application when it is felt necessary to explicitly initiate a takeover for maintenance reasons. As a result of invoking sip_rss_maintenance() the active server will enter a maintenance mode and will redirect any new calls to the passive server. During this maintenance period the passive server will handle new incoming and outgoing calls while waiting for the system to perform a takeover. Depending on the options supplied in the call to sip_rss_maintenance() the system will wait until the calls in progress on the active server have completed before the takeover occurs. After the takeover the formerly active server will either switch to passive mode or quit depending on the options provided.

Events will be raised to all applications using the resilient SIP service when maintenance mode has begun, and when it has ended.

Synopsis

```
ACU_ERR ACU_EXPORT sip_rss_maintenance(SIP_RSS_MAINTENANCE_PARMS*
sip_maintenance_parms);
```

```
typedef struct sip_rss_maintenance_parms
{
     ACU_ULONG       size;           /* IN */
     ACU_INT         timeout;        /* IN */
     ACU_UINT        shutdown;       /* IN */
} SIP_RSS_MAINTENANCE_PARMS;
```

Input parameters

The `sip_rss_maintenance()` function takes a pointer *`sip_maintenance_parms`* to a structure `SIP_RSS_MAINTENANCE_PARMS`. This structure must be initialised in the following way before being used.

timeout

While there are calls in progress that have not been connected the active server will wait before forcing a takeover. Only connected calls will be migrated during a takeover. This *`timeout`* field contains the maximum positive number of seconds that the system should wait in maintenance mode before forcing a takeover. When all calls are in the connected state the takeover will happen. If the timeout has elapsed then any calls that have not reached the connected state will be dropped. The other possible values that this field can have are:

```
-1        Infinite timeout - takeover will only happen when all calls in
          progress have reached the connected state.
0         Immediate timeout - only calls that have already reached the
          connected state will be migrated.
```

shutdown

A non-zero value in the shutdown field will result in the SIP service halting after the takeover. A zero value indicates that after maintenance is over the formerly active server will become a passive server.

Return values

On successful completion, a value of zero is returned; otherwise, a negative value will be returned indicating the type of error.

5.6.3 sip_target_refresh() – update the route set for a call

A target refresh may be required after takeover for calls that were redirected towards the passive server during maintenance mode. These calls will be using the IP address of the passive server and not the floating IP address. The result will be to update the remote endpoint's route set in order that it uses the floating IP address for the remainder of the call. It may be achieved by calling either `sip_target_refresh()` that will simply update the route set or, if the media has also moved during the takeover process, `sip_media_propose()` may be used to send both the target refresh and the new media offer.

Synopsis

```
ACU_ERR sip_target_refresh(SIP_TARGET_REFRESH_PARMS* sip_target_refresh_parms);

   typedef struct tSIP_TARGET_REFRESH_PARMS
   {
        ACU_ULONG  size;           /* IN */
        ACU_CALL_HANDLE handle;    /* IN */
   } SIP_TARGET_REFRESH_PARMS;
```

Input parameters

The `sip_target_refresh()` function takes a pointer `sip_target_refresh_parms` to a `SIP_SEND_REQUEST_PARMS` structure. The structure must be initialised in the following way before invoking the function.

handle

The *`handle`* field identifies the call that is to send the target refresh. This field is **mandatory**.

Return values

On successful completion, a value of zero is returned; otherwise, a negative value will be returned indicating the type of error.

5.7 Events for Resilient SIP Service

A number of new events will be generated by the resilient SIP service.

5.7.1 Global Events

EV_SIP_RSS_PEER_CONNECTED
This event occurs whenever a SIP server establishes a connection to a peer server

EV_SIP_RSS_SERVER_LOST
This event occurs whenever a connection from an application to a resilient SIP server fails.

EV_SIP_RSS_MAINTENANCE_STARTED
This event occurs when a call to `sip_rss_maintenance()` has completed its initial phase.

EV_SIP_RSS_MAINTENANCE_COMPLETED
If maintenance has completed without needing to drop calls before they got to the connected state then this event will be raised.

EV_SIP_RSS_MAINTENANCE_TIMEOUT
If maintenance has completed due to a timeout then this event will be raised. Some calls were dropped.

EV_SIP_RSS_SERVER_CONFIG_ERROR
This event occurs if a problem is detected involving the status or role of the servers.

EV_SIP_RSS_SERVER_HALT
Some configuration errors may require that a server must shut down to ensure that some service is possible. This event is raised when a server halts due to this condition.

EV_SIP_RSS_SERVER_CONFIGURED
This event occurs when a server goes from a `RSS_NOT_CONFIGURED` state to a `RSS_ACTIVE` or `RSS_PASSIVE`

EV_SIP_RSS_TAKEOVER_STARTED
This event occurs when the servers start to reconfigure their respective roles. This will happen once when the first application connects and the roles are configured for the first time. It will also happen when the roles change due to either an error condition or a period of maintenance.

EV_SIP_RSS_TAKEOVER_COMPLETED
This event occurs when the servers have completed reconfiguring their respective roles. During the interval between receipt of an `EV_SIP_RSS_TAKEOVER_STARTED` and an `EV_SIP_RSS_TAKEOVER_COMPLETED`, the application may experience a pause in response from the servers. During this period, the servers are in an indeterminate state where the plugin cannot reliably decide to which server a given API call should be sent. It will therefore hold these API calls back until the takeover is complete.

EV_SIP_RSS_IPTAKEOVER_SUCCESSFUL
This event occurs when a passive server changes to an active server during a Takeover process and has successfully acquired the floating address. It would normally occur between the `EV_SIP_RSS_TAKEOVER_STARTED` and `EV_SIP_RSS_TAKEOVER_COMPLETED` events but in extreme cases may take place after the `EV_SIP_RSS_TAKEOVER_COMPLETED` event. It is only when both the `EV_SIP_RSS_TAKEOVER_COMPLETED` and `EV_SIP_RSS_IPTAKEOVER_SUCCESSFUL` events have been received that full service is restored to the new active server. The user may wish to set a timer triggered by the `EV_SIP_RSS_TAKEOVER_STARTED` event and when both `EV_SIP_RSS_TAKEOVER_COMPLETED` and `EV_SIP_RSS_IPTAKEOVER_SUCCESSFUL` events

are received within a specified interval duration will there be no need for the user/application to intervene by restarting the 'non-responsive' server (i.e, previous active server). Note that if `EV_SIP_RSS_TAKEOVER_STARTED` and `EV_SIP_RSS_TAKEOVER_COMPLETED` events are received it means that all active calls have been successfully transferred and the new active server is ready to takeover upon receipt of the floating address. If it takes too long to receive the `EV_SIP_RSS_IPTAKEOVER_SUCCESSFUL` event then upon restarting the previous active server it will assume a passive role and surrender the floating address. It is left up to the user to determine and appropriate interval duration.

EV_SIP_RSS_HEARTBEAT_LOST
This event occurs when a server loses contact with its peer.

EV_SIP_RSS_HEARTBEAT_ESTABLISHED
This event occurs when a server establishes contact with its peer.

5.7.2 Call Events

EV_SIP_RSS_CALL_MIGRATION
After a takeover when a call has migrated from one server to another, this event is generated. This allows the user to make adjustments for this call (e.g. if media settings need to be modified).

EV_SIP_RSS_CALL_LOST
If a takeover occurs then only those connected calls that migrated will still be available on the new server. To indicate if a call was 'lost' during a takeover (e.g. an unconnected call) this event will occur. The application need not act on this event as an EV_IDLE will immediately be raised. This allows existing applications to clear their resources in the usual fashion.

Appendix A: SIP specific events

A number of SIP specific events have been added:

EV_MEDIA

A SIP call has completed an offer answer exchange and it is necessary to start, stop or re-configure the underlying media session. The application must now call `sip_details()` in order that it may view the currently negotiated media settings.

EV_MEDIA_PROPOSE

A SIP call has received a new media offer, *outside of the generic call control model*, for example in a re-`INVITE` transaction. (This event may also be raised if an initial `200OK` is received for an outgoing call, which was initiated by an `INVITE` with no SDP). The application should call `sip_details()` and process the *`media_offer_answer`*, then `sip_media_accept()` with an appropriate answer. Alternatively the application may decline this proposal with `sip_media_reject_proposal()`. This event may also be received when an `UPDATE` request containing an SDP offer is received, and may be responded to using either `sip_media_accept()` or `sip_send_response()`.

If the application makes an outgoing call by sending an `INVITE` without SDP (when acting as a B2BUA for example) there is no way to guarantee that the SDP offer will arrive in a `200 OK`. If it arrives in a `180` or `183` then calling `sip_media_accept()` before the arrival of the `200 OK` will be ignored. The application should register for the `ACU_SIP_INITIAL_INVITE_NOTIFICATION` and inspect the raw message to determine if a `200 OK` has been received and at that point call `sip_media_accept()`.

EV_MEDIA_REJECT_PROPOSAL

This event occurs when a SIP call, after previously sending an offer using `sip_media_propose()` receives a rejection, a non-2xx final response, for that offer. The application need not take any particular action at this point, as the previously established media session will still be valid.

EV_MEDIA_REQUEST_PROPOSAL

A SIP call has received a re-`INVITE` with no SDP. The application may either provide a media proposal using `sip_media_propose()` or reject the `INVITE` using `sip_media_reject_proposal()`.

EV_MEDIA_REJECT_REQUEST_PROPOSAL

This event is raised on a SIP call when, subsequent to sending a re-`INVITE` with no SDP (e.g. with `sip_media_request_proposal()`), receives a non-2xx response.

EV_MEDIA_REJECT_COLLISION

This (configurable) event is raised on a SIP call when, subsequent to sending a re-`INVITE` with `sip_media_propose()` or `sip_media_request_proposal()`, the re-`INVITE` collides with a re-`INVITE` sent at the same time from the remote end. The re-`INVITE` from the remote end 'wins' and after the ***EV_MEDIA_REJECT_COLLISION*** there will be either an ***EV_MEDIA_PROPOSE*** or an ***EV_MEDIA_REQUEST_PROPOSAL***.

EV_SIP_MEMORY_LIMIT_EXCEEDED

This event is raised on the global notification event queue to indicate that a pre-configured maximum memory limit has been reached by the SIP service. See configuring the Overload Monitor for more details.

EV_SIP_MEMORY_LIMIT_WARNING

This event is raised on the global notification event queue to indicate that a pre-configured memory limit to indicate high levels of traffic has been exceeded by the SIP service. See configuring the Overload Monitor for more details.

EV_SIP_MEMORY_OK

This event is raised on the global notification event queue to indicate that the SIPservice's memory usage has fallen back to acceptable levels after a previous spike in traffic. See configuring the Overload Monitor for more details.

EV_SIP_WAIT_FOR_SUBSCRIBER

This event is raised on a SIP subscription after a new notifier has been created using `sip_sub_notifier()`. `sip_details()` should not be called at this point.

EV_SIP_WAIT_FOR_NOTIFIER

This event is raised on a SIP subscription after a new subscriber has been created using `sip_sub_subscriber()`. `sip_details()` should not be called at this point.

EV_SIP_SUBSCRIBED

`sip_details() should always be called here.`

This event is raised on a SIP subscription given the following conditions:

- The application has accepted an incoming `SUBSCRIBE` request with a call to `sip_sub_accept()`. `sip_sub_notify()` must be called in order to conform to RFC 3265.
- The application accepted an initial `SUBSCRIBE` with a `202 Accepted` and subsequently called `sip_sub_notify()` with the ***pending*** field set to 0.
- The SIP service has received either a `200 OK` response to an initial `SUBSCRIBE` request or a `NOTIFY` request whose `Subscription-State` is `active` for the same subscription but on a new dialog. As such, the application may receive multiple events of this type for a single subscription.

EV_SIP_SUBSCRIPTION_PENDING

`sip_details()` should always be called here.

This event is raised on a SIP subscription given the following conditions:

- The application has accepted an incoming `SUBSCRIBE` request with a call to `sip_sub_accept()` with the ***acknowledge*** field set to 1. `sip_sub_notify()` must be called in order to conform to RFC 3265.
- The SIP service has received either a `202 Accepted` response to an initial `SUBSCRIBE` request or a `NOTIFY` request whose `Subscription-State` is `pending` for the same subscription but on a new dialog. As such, the application may receive multiple events of this type for a single subscription.

EV_SIP_SUBSCRIPTION_CANCELLED

`sip_details() should always be called here.`

This event is raised on a SIP subscription given the following conditions:

- A subscriber has received a `NOTIFY` whose `Subscription-State` is `terminated`. `sip_sub_release()` should be called here if the ***dialog_count*** is 0.
- A subscriber has received a non-2xx response to the initial `SUBSCRIBE` request. `sip_sub_release()` should be called here if the ***dialog_count*** is 0.
- A subscriber has received one of the following responses to a `SUBSCRIBE` sent to refresh the subscription. The responses at the time of writing are 404, 405, 410, 416, 480, 481, 482, 483, 484, 485, 489, 501 and, 604.
- `sip_sub_release()` should be called here if the ***dialog_count*** is 0.
- A notifier has received a `SUBSCRIBE` with an `expires` value of 0. `sip_sub_notify()` must be called here in order to conform to RFC 3265.
- A notifier has received a non-2xx response to a `NOTIFY` request.

Notifiers should always call `sip_sub_release()` on receipt of this event.

EV_SIP_SUBSCRIPTION_REQUEST

This event is raised on a SIP subscription when a notifier receives an initial `SUBSCRIBE` request. The application should examine `sip_details()` and call either `sip_sub_accept()` or `sip_sub_cancel()` as a result.

EV_SIP_SUBSCRIPTION_NOTIFICATION

This event is raised on a SIP subscription when a subscriber receives a `NOTIFY` request on an existing subscription. The application should examine `sip_details()` to determine its contents.

EV_SIP_SUBSCRIPTION_REFRESH

This event is raised on a SIP subscription when a notifier receives a `SUBSCRIBE` request on an existing subscription. The application should examine `sip_details()` and call `sip_sub_notify()` in order to conform to RFC 3265.

EV_SIP_SUBSCRIPTION_REFRESH_FAILED

This event is raised on a SIP subscription when a subscriber receives an unsuccessful response to a refresh attempt on an existing subscription, which is ***not*** one of the following: 404, 405, 410, 416, 480, 481, 482, 483, 484, 485, 489, 501 and, 604. The application should examine `sip_details()`. There will be no further attempts to automatically refresh the subscription, but the subscription will be valid until it eventually expires.

Appendix B: Raw SDP usage

The structure ACU_MEDIA_OFFER_ANSWER contains a field called raw_sdp. When present in the call's details, this field will always be populated if there is some new SDP *received* in a SIP message. However, when this structure is used to populate some SDP prior to *transmission,* then usage of the raw_sdp is strictly optional. For the vast majority of SIP/SDP applications, the structures defined in this API are adequate and in need of no further refinement.

In the event of needing to set an SDP feature not specified in the API, the application may populate the SDP body itself and set the raw_sdp element with a char* holding this body. To do this correctly it is recommended that the application writer refer to the IETF documents: RFC 2327 and RFC 3264, as, when raw SDP mode is used the sipserv executes no checking or validation code. Note that when the sipserv detects a non-NULL value for raw_sdp it assumes that the application wishes to use purely the raw mode for this call and ignores any SDP representation structures present.

It is possible for the application to use a combination of raw SDP and abstracted SDP structures. For example, an application calls sip_openout() using the abstracted structures to initially place a call and subsequently use raw SDP for a complex re-INVITE sent using sip_media_propose(). Note that when using raw SDP, either alone or accompanying function calls made with the abstracted structures, it is very important that the application increment the 'SDP session version', a field held within the SDP body, at the correct junctures. The above RFCs provide guidance in achieving this.

e.g. Application defines the SDP used for a sip_openout() using the raw mode of the structure:

```
SIP_OUT_PARMS outx;
INIT_ACU_STRUCT(&outx);

// full setup of open out structure omitted for brevity

ACU_MEDIA_OFFER_ANSWER mo1;
memset(&mo1, 0, sizeof(ACU_MEDIA_OFFER_ANSWER));

char* sdpUser =
"v=0\r\n"
"o=VoIP-media-session 123 123 IN IP4 192.168.15.202\r\n"
"s=SIP Call\r\n"
"c=IN IP4 192.168.0.2\r\n"
"t=0 0\r\n"
"m=audio 5006 RTP/AVP 18 121\r\n"
"a=ptime:80\r\n"
"a=fmtp:121 0-16\r\n"
"a=rtpmap:121 telephone-event/8000/1\r\n";

// populate open out structure with raw SDP
mo1.raw_sdp = sdpUser;
outx.media_offer_answer=&mo1;

ACU_ERR rc = sip_openout(&outx);
```

Appendix C: Receipt of raw SIP messages

The Extended SIP API provides applications with the opportunity to inspect raw SIP messages as received by the protocol stack. Since the internal queuing of these messages may give rise to a performance overhead, their delivery to the application is switched off by default. Furthermore, different applications may wish to examine different message types. To permit the developer to select which SIP messages, if any, should be queued then passed up the application, a set of bit flags is provided:

```
typedef enum acu_sip_message_notification_masks
{
   ACU_SIP_INITIAL_INVITE_NOTIFICATION        = 0x00000001, (see note 1)
   ACU_SIP_REINVITE_NOTIFICATION              = 0x00000002, (see note 1)
   ACU_SIP_TRANSFER_INVITE_NOTIFICATION       = 0x00000004, (see note 1)
   ACU_SIP_INFO_NOTIFICATION                  = 0x00000008,
   ACU_SIP_NOTIFY_NOTIFICATION                = 0x00000010,
   ACU_SIP_REGISTER_NOTIFICATION              = 0x00000020, (see note 2)
   ACU_SIP_SUBSCRIBE_NOTIFICATION             = 0x00000040, (see note 2)
   ACU_SIP_OPTIONS_NOTIFICATION               = 0x00000080,
   ACU_SIP_BYE_NOTIFICATION                   = 0x00000100, (see note 1)
   ACU_SIP_MESSAGE_NOTIFICATION               = 0x00000200,
   ACU_SIP_UPDATE_NOTIFICATION                = 0x00000400,
   ACU_SIP_PRACK_NOTIFICATION                 = 0x00000800, (see note 1)
   ACU_SIP_REFER_NOTIFICATION                 = 0x00001000, (see note 1)
   ACU_SIP_INITIAL_ACK_NOTIFICATION           = 0x00002000, (see note 1)
   ACU_SIP_REINVITE_ACK_NOTIFICATION          = 0x00004000, (see note 1)
   ACU_SIP_TRANSFER_ACK_NOTIFICATION          = 0x00008000, (see note 1)
 } ACU_SIP_MESSAGE_NOTIFICATION_MASKS;
```

NOTE

1. These message types are not valid for use with `enable_response_mask`.
2. These message types are only valid for use with the out of dialog API.

A combination of these flags may be written to the *`request_notification_mask`*, *`response_notification_mask`* and *`enable_response_mask`* field elements of the *`SIP_OUT_PARMS`* and *`SIP_IN_PARMS`* API structures. (Note that the application can *separately* configure a requirement to be notified on receipt of request or response messages, in addition to message type based selection.)

Here are some examples of configuration of a call in order that it may be notified on receipt of given messages:

For example, call requires notification of `INFO` request receipt

```
SIP_OUT_PARMS outx;
INIT_ACU_STRUCT(&outx);

outx.request_notification_mask = ACU_SIP_INFO_NOTIFICATION;
```

For example, call requires notification of initial `INVITE` response receipt

```
SIP_IN_PARMS inx;
INIT_ACU_STRUCT(&inx);

inx.response_notification_mask = ACU_SIP_INITIAL_INVITE_NOTIFICATION;
```

(It is possible to bitwise OR the above flags into a mask, to specify interest in more than one message type.)

On receipt of a message whose type was specified in one of the above fields, the SIP service will queue the message and raise `EV_DETAILS` to notify its presence. The message can be collected by the application using the `sip_details()` function call

and will be presented in the *sip_message* field of SIP_DETAIL_PARMS structure.

Note that not calling sip_details() to retrieve SIP messages flagged by EV_DETAILS results in these messages remaining in memory until the call handle is released.

How the SIP service responds to a given request depends on whether the relevant mask has been set in the *request_notification_mask* and *enable_response_mask* fields. This is most easily explained with an example. Here we will assume that the SIP service has just received a mid call INFO request. There are then the following possibilities:

- If INFO notification has not been requested, the SIP service will silently accept the request and respond with a 200 OK.
- If INFO notification has been requested but the enable_response_mask has not been set, the SIP service will silently respond with a 200 OK and notify the application of the INFO request by raising an EV_DETAILS.
- Finally, if both request_notification_mask and enable_response_mask have been set, an EV_DETAILS will be raised and no response will be sent. It is then down to the application to formulate its own response in a timely fashion through a call to sip_send_response().

Appendix D: Using TLS to provide security

TLS, transport layer security, (SSL as standardised by the IETF) is available as a transport mechanism when using the Aculab SIP service in combination with the SIP_TLS software. This protocol encrypts the SIP messages prior to transmission over a TCP stream. The ability to exploit this form of transport security is announced by SIP entities by the 'sips', as opposed to 'sip' URI scheme.

NOTE

The default IP network port for secure SIP is `5061`.

The application writer is advised that some existing background knowledge in IP security and TLS may be useful in configuring the Aculab SIP service for TLS and trouble-shooting any issues may arise.

TLS draws upon a wealth of cryptographic routines and techniques: very briefly, it permits the creation of a symmetric cryptography session by use of public key techniques to negotiate encryption parameters. In order for creation of this session the parties involved must supply a minimum set of parameters, here follows a very brief description of the configuration information required by the application in order that those parameters are present.

In order to support TLS, it must be configured via the configuration file sipserv.cfg. Supply the following option to the SIP service:

```
UseTLS = 1
```

For a minimal TLS configuration, at least one trusted third party, who is often known as a 'Certification Authority' (CA), must be specified. This is in the form of a digital certificate. A TLS application may have a list of trusted third parties. To configure this list use the setting below:

```
TLS_TrustedCertificates = <path to file containing a list of certificates>
```

Below is an example of the possible content of such a file:


```
-----BEGIN CERTIFICATE-----
MIICoDCCAgmgAwIBAgIBADANBgkqhkiG9w0BAQQFADBFMQswCQYDVQQGEwJBVTET
MBEGA1UECBMKU29tZS1TdGF0ZTEhMB8GA1UEChMYSW50ZXJuZXQgV2lkZ2l0cyBQ
dHkgTHRkMB4XDTA2MDcxMjA3NDc1MloXDTA2MDgxMTA3NDc1MlowRTELMAkGA1UE
BhMCQVUxEzARBgNVBAgTClNvbWUtU3RhdGUxITAfBgNVBAoTGEludGVybmV0IFdp
ZGdpdHMgUHR5IEx0ZDCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEAlX/aSX2G
7VRbQgEvDYO2ALFrwAUF3Ptc4IBY4ouY9oEEFYOpsuQcZUAfEMutnQ8+n5/hwKKG
+F1+Tz7u06q+zkN2sE9EvU5xBVCoIM9GP+mIgOqSgNSdA1zf7c3bnJXxsrGKljJg
B7DCZoaKqn6udhVwSxtJAbB4TfEiI+0jiHsCAwEAAaOBnzCBnDAdBgNVHQ4EFgQU
WN7cMk4ie3WI44p2j4SdSv+xqvgwbQYDVR0jBGYwZIAUWN7cMk4ie3WI44p2j4Sd
Sv+xqvihSaRHMEUxCzAJBgNVBAYTAkFVMRMwEQYDVQQIEwpTb21lLVN0YXRlMSEw
HwYDVQQKExhJbnRlcm5ldCBXaWRnaXRzIFB0eSBMdGSCAQAwDAYDVR0TBAUwAwEB
/zANBgkqhkiG9w0BAQQFAAOBgQAsPBL+SyO4tytnjDx874qap21SiBj6ha8DrLLB
TY1W2KHzKdjv1tAweF9z914HNQU594fB7XtDa1VMT3VOZ1EKJk+NOd3Wnbyn6Chy
KjZ4EOcE9BZjuSmvJlGIVtRrN2LrU7cSHFxoY8FUtDYpox1lUoiiZmVVuTrxv/1u
DVnk8A==
-----END CERTIFICATE-----
```


The above file contains one certificate; additional certificates would be concatenated to the end of this.

In order that the TLS application may act as a 'server' in a TLS transaction, that is receive simple requests to establish secure channels from clients, a path to a server certificate must be configured:

```
TLS_ServerCertificate = <path to server certificate>
```

The following is an example of the possible server certificate file:


```
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQD2lvP2blSTmEl+6hBr+zVL8PukYDJty4yPRlnsTqaEmOz1ryCt
c3fXfGpxvXKoZ42Z8BjE4oE7IrJDIoOVJkjAhW1JAxawt18Jo0nh59aAJMMiudkl
wnMkLx2WLq79MRs/bl6epIQMcWrUvrqEwvL+vpwwtwwql5/xFimPqlPtFwIDAQAB
AoGAPhv5aNmTTfWulVvpNPl6DB2vlFBygzsgtTm4DpAk2wMVtDAfH/EUf18kBG/+
QDKM9PgHlRekCzwLAGPiFqAk9HJQKwrxCWI2OSwAAE5zMHJJSHD3aJoU6TKRurhp
G1lKDn/Efn+u3FVjl14vkhgbxDWil99xIcO0zrHiCUd3DRkCQQD/07Y90BtnqFey
sFHeITaA57z+H8NszATKaOpKesjCFbNIcMHc0BD3iwySiG89LI68PwWtEnrxa93B
FKx6lFP1AkEA9sGkVxIQ8R4Ru5q4lfm6hQPBQtJYF2DBgzZViZsDzuOuFWmh2EK5
FHry7ugejbPCofc47yFCxBxPZt/EoHmhWwJBAP8w0xOp6gxmssR+ecvY2aOQhsgA
K1L12LmIGl4dSPHB78sNH3UC4EnuHKZ3Dm+5aNCDFUhlrNnyPYyC8OJ935kCQQDx
FRQZNb9ztCsjHfFGJi1Dk/2X6abDgHbQWZ+Mx/Uah3wn04KapeXpyo3bONHRJFX+
pGntyJNQw1AYdHvJRcyNAkB9Ulyxvubh/xfVIHliKFh1RHkf212W3q3s4WlFRDip
GImb4H6wHQwwyIbwnG5gRaOYMelMjHErB8nHv/+4vd62
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
MIIB+TCCAWICAQEwDQYJKoZIhvcNAQEFBQAwRTELMAkGA1UEBhMCQVUxEzARBgNV
BAgTClNvbWUtU3RhdGUxITAfBgNVBAoTGEludGVybmV0IFdpZGdpdHMgUHR5IEx0
ZDAeFw0wNjA3MTIwNzQ5NTBaFw0wNjA4MTEwNzQ5NTBaMEUxCzAJBgNVBAYTAkFV
MRMwEQYDVQQIEwpTb21lLVN0YXRlMSEwHwYDVQQKExhJbnRlcm5ldCBXaWRnaXRz
IFB0eSBMdGQwgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAPaW8/ZuVJOYSX7q
EGv7NUvw+6RgMm3LjI9GWexOpoSY7PWvIK1zd9d8anG9cqhnjZnwGMTigTsiskMi
g5UmSMCFbUkDFrC3XwmjSeHn1oAkwyK52SXCcyQvHZYurv0xGz9uXp6khAxxatS+
uoTC8v6+nDC3DCqXn/EWKY+qU+0XAgMBAAEwDQYJKoZIhvcNAQEFBQADgYEANK35
BNKwrQuxLcAu406siXOQ5SU1d7zFB4Jp9LWcR2CXxngztmUryTk4zWEWOpQzy/bT
+tjRZLhtGQqUypNOo2wESjlEh1+sLiSBHErIaXpRHaVoWlq+yD2zMtdNuB3IFcId
wGLQkvMV8OigZhDGfCtsvGYFog1xmKk18SZHcR8=
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIICoDCCAgmgAwIBAgIBADANBgkqhkiG9w0BAQQFADBFMQswCQYDVQQGEwJBVTET
MBEGA1UECBMKU29tZS1TdGF0ZTEhMB8GA1UEChMYSW50ZXJuZXQgV2lkZ2l0cyBQ
dHkgTHRkMB4XDTA2MDcxMjA3NDc1MloXDTA2MDgxMTA3NDc1MlowRTELMAkGA1UE
BhMCQVUxEzARBgNVBAgTClNvbWUtU3RhdGUxITAfBgNVBAoTGEludGVybmV0IFdp
ZGdpdHMgUHR5IEx0ZDCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEAlX/aSX2G
7VRbQgEvDYO2ALFrwAUF3Ptc4IBY4ouY9oEEFYOpsuQcZUAfEMutnQ8+n5/hwKKG
+F1+Tz7u06q+zkN2sE9EvU5xBVCoIM9GP+mIgOqSgNSdA1zf7c3bnJXxsrGKljJg
B7DCZoaKqn6udhVwSxtJAbB4TfEiI+0jiHsCAwEAAaOBnzCBnDAdBgNVHQ4EFgQU
WN7cMk4ie3WI44p2j4SdSv+xqvgwbQYDVR0jBGYwZIAUWN7cMk4ie3WI44p2j4Sd
Sv+xqvihSaRHMEUxCzAJBgNVBAYTAkFVMRMwEQYDVQQIEwpTb21lLVN0YXRlMSEw
HwYDVQQKExhJbnRlcm5ldCBXaWRnaXRzIFB0eSBMdGSCAQAwDAYDVR0TBAUwAwEB
/zANBgkqhkiG9w0BAQQFAAOBgQAsPBL+SyO4tytnjDx874qap21SiBj6ha8DrLLB
TY1W2KHzKdjv1tAweF9z914HNQU594fB7XtDa1VMT3VOZ1EKJk+NOd3Wnbyn6Chy
KjZ4EOcE9BZjuSmvJlGIVtRrN2LrU7cSHFxoY8FUtDYpox1lUoiiZmVVuTrxv/1u
DVnk8A==
-----END CERTIFICATE-----
```


NOTE

The above file contains the server's private key, certificate and (the final block) the digital certificate of the trusted entity that 'signed' the server's certificate.

Configuration of the 3 settings mentioned above are sufficient in order that the SIP service may negotiate secure sessions with a variety of remote TLS applications. For such a simple configuration, the digital certificates used should be created using the RSA public key algorithm as this provides authentication, encryption, and secrecy. New certificates may be loaded at runtime through a call to `sip_load_tls_configuration()`.

Additional parameters, beyond the simple configuration exemplified above, may be provided to further tailor the cryptographic characteristics of the negotiated sessions. Again, note that the 3 basic settings mentioned above are quite adequate for most secure sessions. The additional, 'advanced', parameters are briefly explained below:

```
TLS_VerifyPeer = <0 or 1>
```

This setting forces the TLS software to attempt to authenticate the remote peer. Additionally, an application running as a server would insist on presentation of a certificate by the client application, and fail the handshake on it's absence.

```
TLS_VerifyDepth = <n>
```

Where *n* is a positive integer or zero.

This setting configures the maximum depth to which certificates may be chained. To explain further: a server certificate may be signed by `certificaten` where `certificaten` itself is not a certificate trusted by the peer; provided that `certificaten` is signed by a trusted party. This process, that is the delegation of certificate signing to an intermediary, may be repeated. Hence, the above setting 'TLS_VerifyDepth' is provided in order that the extent of this delegation be limited to a specified number of intermediaries. Zero for this parameter yields default behaviour: no verify depth checking.

```
TLS_DH512File = <path to 512-bit Diffie-Hellman file>
TLS_DH1024File = <path to 1024-bit Diffie-Hellman file>
```

The above settings permit the TLS application to provide Diffie-Hellman (DH) parameter files. There are 2 reasons why the application writer may wish to support DH parameters. Firstly, in the case of the certificates being based upon DSA keys, no encryption mechanism is provided by the DSA keys: the DH parameters may be used to provide this mechanism which is essential in the negotiation of keys used for the symmetric exchange. Secondly, when the certificates are based on RSA keys, where an encryption mechanism is available, the DH parameters provide additional security in the form of 'forward secrecy'.

```
TLS_PostConnectionCheck = <0 or 1>
```

Attempting to authenticate the peer through use of the `TLS_VerifyPeer` options above does not always provided the required level of security. Firstly, a NULL certificate is considered valid by the verify peer mechanism and secondly, any certificate signed by the CA will also be considered valid. Nothing prevents an attacker from acquiring their own certificate signed by the CA and then masquerading as a genuine peer. This may be prevented by performing a DNS lookup on the fully qualified domain name (FQDN) contained within the certificate and comparing it with the known IP address of the remote party. If `TLS_PostConnectionCheck` is set, both of these additional checks will be performed.

NOTE

This product includes cryptographic software written by Eric Young (eay@cryptsoft.com).

Appendix E: Using the Tel URI scheme

The Tel URI scheme is another scheme supported in the formation of the To and From headers based on RFC 3966. This scheme is used to support telephone numbers and is used mainly in the API commands `sip_openout()` and `sip_send_out_of_dialog_request()`.

A `Route` header must be included in each request. If `ipt_set_sip_proxy()` has been used to configure a global outbound proxy then this happens automatically.Otherwise the `Route` header should be included as one of the `custom_headers`.

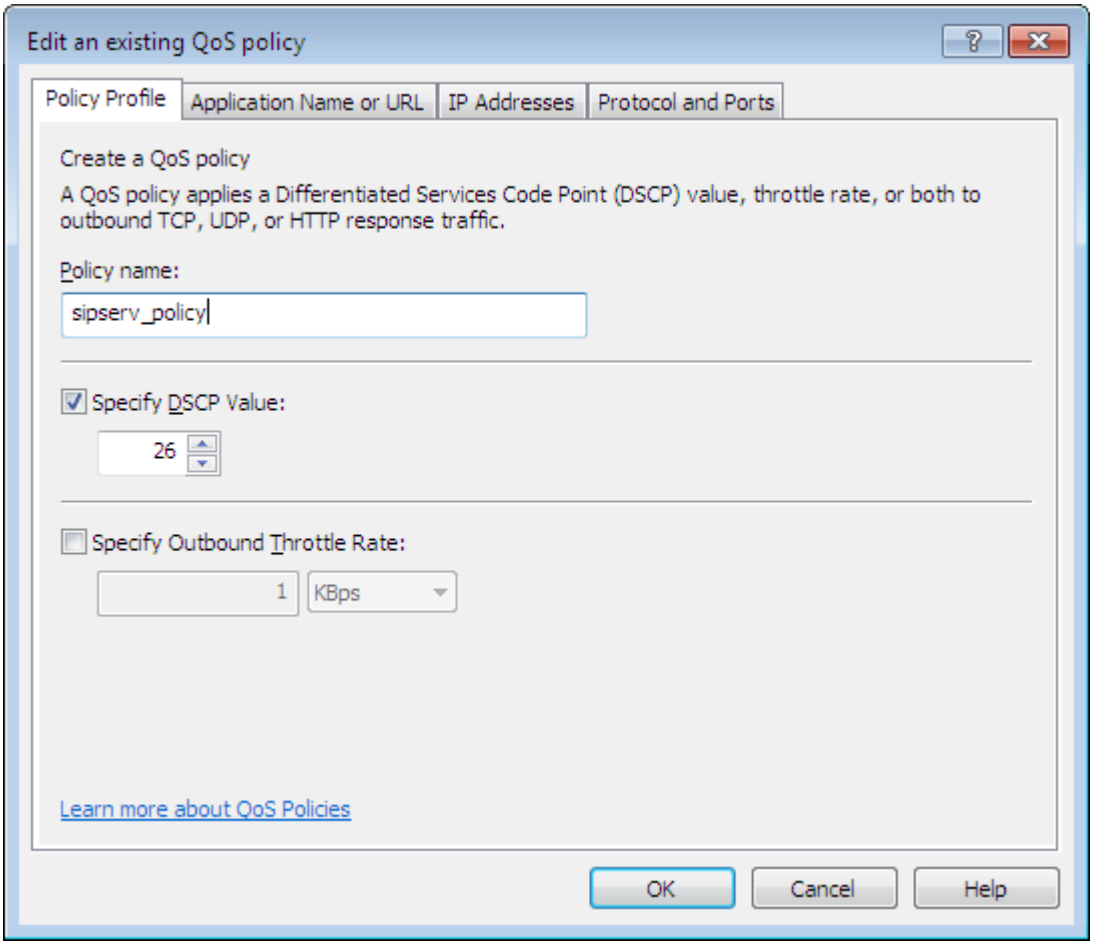
For example:

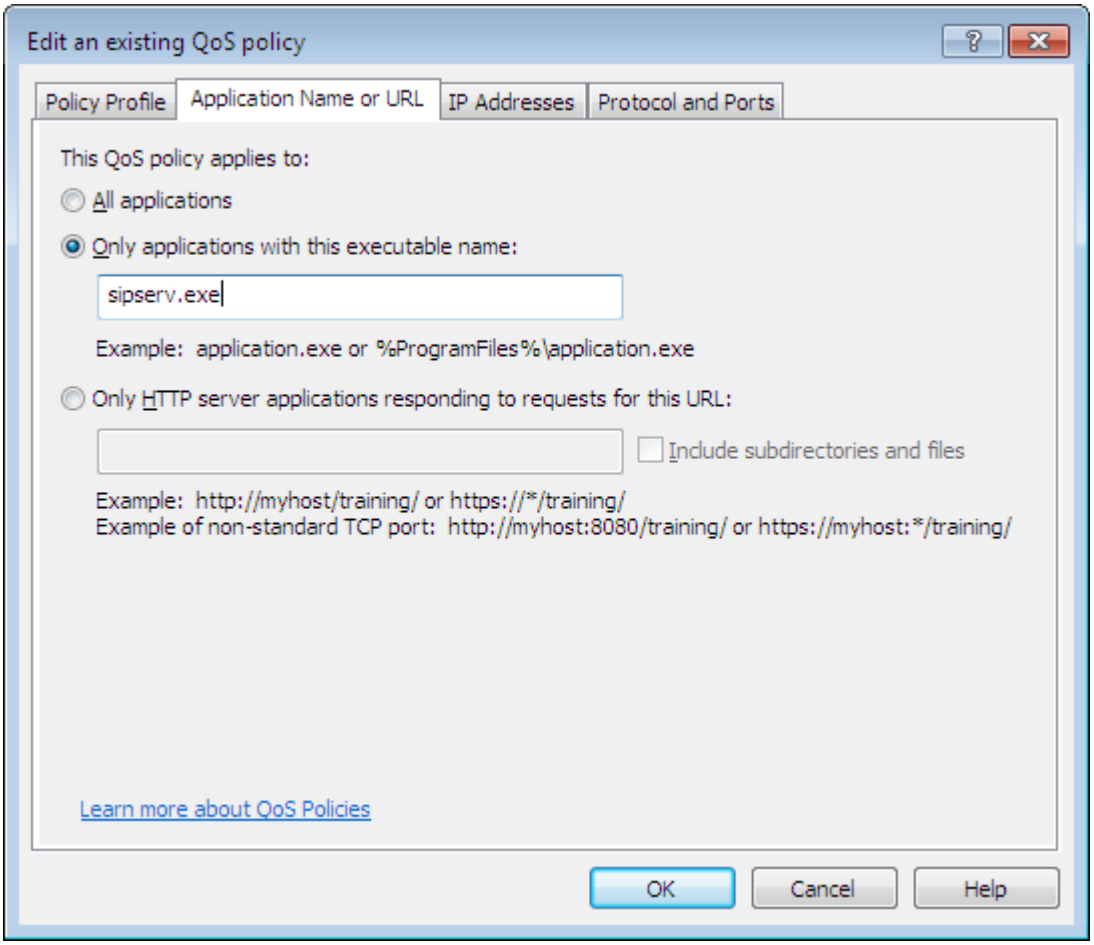
```
SIP_OUT_PARMS outx;
INIT_ACU_STRUCT(&outx);

outx.net = sipPort;
outx.destination_addr = "tel:+1-555-01234";
outx.originating_addr = "tel:863-6789;phone-context=+1-914-555";

outx.custom_headers = "Route: <sip:10.202.205.219:5060;lr>";
// This field must be null terminated

// setup SIP_OUT_PARMS

rc = sip_openout(&outx);
```

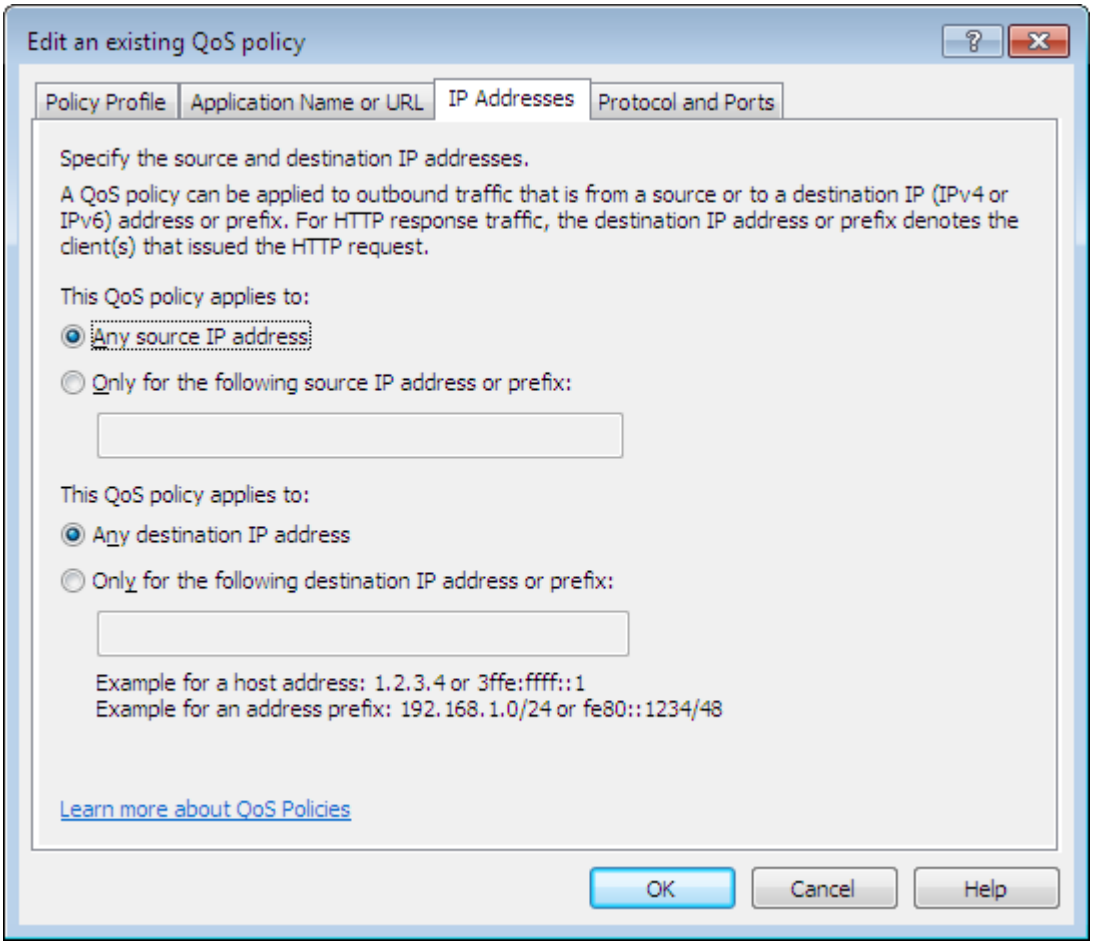
Appendix F: Quality Of Service for Windows(DSCP)

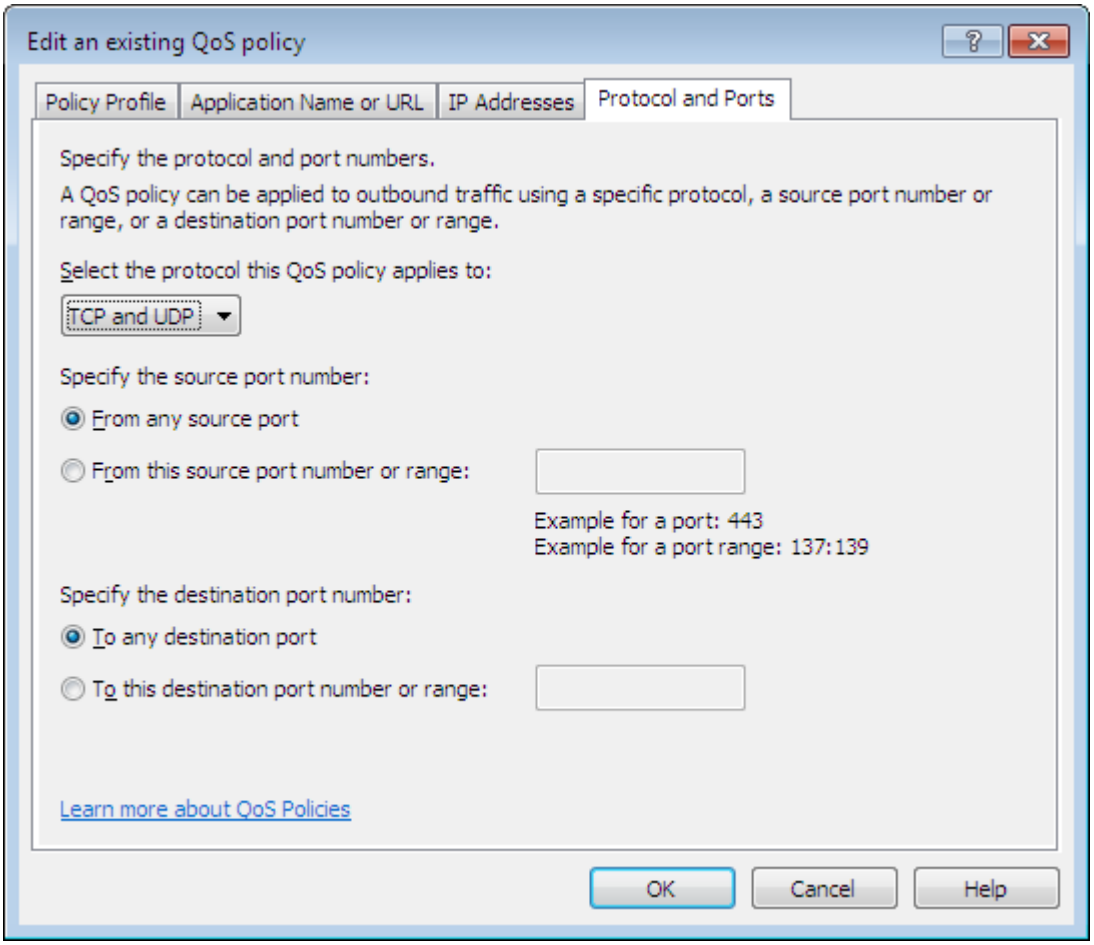
For Windows, consult the latest Technet documentation for Policy-based Quality of Service for the version of your operating system.

Here are examples of how you would configure sipserv.exe to use a particular DSCP value.




Edit an existing QoS policy
Policy Profile
Application Name or URL
IP Addresses
Protocol and Ports
This QoS policy applies to:
All applications
Only applications with this executable name:
sipserv.exe
Example: application.exe or %ProgramFiles%\application.exe
Only HTTP server applications responding to requests for this URL:
Include subdirectories and files
Example: http://myhost/training/ or https://*/training/
Example of non-standard TCP port: http://myhost:8080/training/ or https://myhost:*/training/
Learn more about QoS Policies
OK
Cancel
Help


Edit an existing QoS policy
Policy Profile
Application Name or URL
IP Addresses
Protocol and Ports
Specify the source and destination IP addresses.
A QoS policy can be applied to outbound traffic that is from a source or to a destination IP (IPv4 or IPv6) address or prefix. For HTTP response traffic, the destination IP address or prefix denotes the client(s) that issued the HTTP request.
This QoS policy applies to:
Any source IP address
Only for the following source IP address or prefix:
This QoS policy applies to:
Any destination IP address
Only for the following destination IP address or prefix:
Example for a host address: 1.2.3.4 or 3ffe:ffff::1
Example for an address prefix: 192.168.1.0/24 or fe80::1234/48
Learn more about QoS Policies
OK
Cancel
Help


Edit an existing QoS policy
Policy Profile
Application Name or URL
IP Addresses
Protocol and Ports
Specify the protocol and port numbers.
A QoS policy can be applied to outbound traffic using a specific protocol, a source port number or range, or a destination port number or range.
Select the protocol this QoS policy applies to:
TCP and UDP
Specify the source port number:
From any source port
From this source port number or range:
Example for a port: 443
Example for a port range: 137:139
Specify the destination port number:
To any destination port
To this destination port number or range:
Learn more about QoS Policies
OK
Cancel
Help