

ACULAB.COM

# Aculab extended SIP API guide



## Revision 6.8.2

MAN1762

## **PROPRIETARY INFORMATION**

The information contained in this document is the property of Aculab plc and may be the subject of patents pending or granted, and must not be copied or disclosed without prior written permission. It should not be used for commercial purposes without prior agreement in writing.

All trademarks recognised and acknowledged.

Aculab plc endeavours to ensure that the information in this document is correct and fairly stated but does not accept liability for any error or omission.

The development of Aculab's products and services is continuous and published information may not be up to date. It is important to check the current position with Aculab plc.

Copyright © Aculab plc. 2018 all rights reserved.

## **Document Revision**

Rev	Date	Ву	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	
6.4.10 0	05.09.08	SP	Updated with new sipserv.cfg options sip_send_response added	
6.4.12 6	07.04.09	SP	Added sip_load_tls_configuration. Removed TLS plugin. Added sip_send_invite_response, new configuration options.	
6.4.15 2	21.01.10	NC	Added DRSS section	
6.4.15 3	11/03/10	EBJ	Updated to corporate fonts	
6.4.15 4	01/04/10	NC	Added extra features to sip_disconnect, fixed errors in sip_incoming_ringing and sip_send_out_of_dialog_request	
6.4.16 1.8	12/08/10	NC	Updated DRSS information	



6.4.16 1.13	20.10.10	DF	Changed name of document	
6.4.16 1.14	17.12.10	NC	Extra features to sip_disconnect(), fixes and updates to configuration section, aligned with current cl_lib.h	
6.4.16 3	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	
6.4.16	15.07.11	NC	Update – no MHP compatibility with DRSS.	
6			Update – DRSS for ringing calls	
			Clarify procedure when EV_MEDIA_PROPOSE is received	
6.4.16 7	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()	



6.7.0	10.03.17	NMC	ACU_RAISE_FORKED_EVENT introduced to <i>sip_openout</i> in <i>call_options</i> and new event EV_FORKED	
6.7.1	13.02.18	NMC	User added to sip_out_parms	
6.7.2	06.06.18	NMC	Correction – DisableTrying when set disables the automatic sending of '100 trying' response	
6.7.3	02.07.18	NMC	PayloadParams option added to sipserv.cfg	
6.7.4	20.05.22	KGB	sip_free_message add cause to SIP_READ_MESSAGE_PARMS	
			Clarifications to TLS certificate use	
			Correct notes for ACU_SIP_MESSAGE_NOTIFICATION_MASKS	
			Add note to sip_send_response to mention sip_send_invite_response	
6.7.5	15.05.22	ACP	LegacySetSipProxy & LegacyQuerySipProxy	
6.7.6	02.12.22	ACP	Additional notes on use of <i>sip_details</i>	
6.7.7	16.02.23	ACP	Additional examples of contact_address	
6.7.8	14.03.23	ACP	OutboundThreads functionality changed	
6.8.0	07.08.23	ACP	CipherList, CipherSuites & DHGroups	
6.8.1	04.10.23	ACP	Corrected notes on use of <i>sip_details</i> re. <i>events</i>	
6.8.2	13.03.24	ACP	Format only	



## CONTENTS

1	Intr	stroduction				
1.1 Scope			9	. 10		
	1.2	Overl	ap of the extended SIP API and generic call control API	. 10		
		1.2.1	Extended versions of existing generic API functions	10		
		1.2.2	SIP/SDP specific routines targeted towards third party call control and	re-		
			INVITE handling			
		1.2.3	Routines supporting the extended architecture	11		
		1.2.4	Miscellaneous	11		
		1.2.5	Out of dialog messages	12		
		1.2.6	Subscription API	12		
		1.2.7	Transports	13		
		1.2.8	IPv6			
		1.2.9	Asynchronous calls	13		
2	Со	nfiguri	ing the SIP service	16		
-			ervice logging configuration			
			on timer configuration			
			eader configuration			
			configuration			
			buted SIP settings			
			n Parameters			
			IPv6 Listen Addresses			
	2.8		oad Monitor			
			ellaneous global settings			
			a Handler Plugin Settings			
2	Inte	rfaco	definition (APIs)	20		
З			pen_port() – open a SIP call control port			
			penout() - open a SIP call control port penout() - open for outgoing call			
			penin() - open for incoming call			
			ccept() - accept incoming call			
			etails() - get call details			
			ree_details() - return memory allocated by sip_details()			
			<pre>coc_details() = retain memory anocated by sip_details() ncoming_ringing() = send incoming ringing</pre>			
	3.8	sin n	rogress() - send progress information	56		
			end_invite_response() - Send response to initial INVITE			
			eature_send() - sending feature information			
			nedia_propose() – send a media proposal			
			nedia_accept() – accept a media proposal			
			nedia_request_proposal() – request a media proposal			
			nedia_reject_proposal() – reject a media proposal			
	3.15	5sip s	end_request() - send a mid call SIP request	. 68		
			end_response() - send a mid call SIP response			
			et reason phrase() – modify a SIP response reason phrase			
			dd_answer_challenge_credentials() – provide authentication crede			
		•				
	3.19	9sip_re	emove_answer_challenge_credentials() - remove authentication			
			entials	.75		
	3.20	)sip_d	isconnect() - send a 3xx response, CANCEL or BYE or response to	) BYE		
			•	.77		

		1 sip_recall() – call an alternative address	
	3.22	2sip_set_tls_private_key_password() – pass a password for a TLS pri	vate key
			80
	3.23	3sip_load_tls_configuration() - load a new set of TLS certificates	81
	3.24	4sip_set_message_notification() - declare an interest in out of dialog	messages
			83
	3.25	5sip_send_out_of_dialog_request() - send an out of dialog request	85
	3.26	6sip_send_out_of_dialog_response() - send an out of dialog response	e87
	3.27	7 sip_read_request() - collect an out of dialog request	88
	3.28	Bsip_read_response() - collect an out of dialog response	89
	3.29	9sip_read_out_of_dialog_failure () - collect out of dialog failure notifi	cation90
	3.30	Dsip_free_message()- free memory associated with out of dialog notif	ication 92
	3.31	1 sip_sub_subscriber() – SUBSCRIBE to an event package	93
	3.32	2sip_sub_notifier() – wait for SUBSCRIBE requests to a specific even	t package
		· ·	
	3.33	3sip_sub_accept() – accept or acknowledge a SUBSCRIBE request	98
		4 sip_sub_notify() – NOTIFY a subscriber of a state change in an even	
			•
	3.35	5sip_sub_cancel() – cancel an existing subscription	102
		6 sip_sub_release() – release the internal resources associated with a	
		subscription	
	3.37	7 sip sub fetch() – request an immediate fetch of subscription state	105
		Bsip_set_global_tos() – Change the ToS (Type of Service)	
4		specific structures	
		ACU_RAW_MESSAGE_BODY	
		ACU_IP_ADDRESS	
		ACU_PAYLOAD	
		ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE	
	4.5	ACU_MEDIA_DESCRIPTION	
		ACU_MEDIA_OFFER_ANSWER	
		ACU_MEDIA_SESSION	
		ACU_SIP_MESSAGE	
		ACU_STRING_LIST	
	4.11	1 ACU_SUBSCRIPTION_INFO	125
5	Dua	al redundant SIP service (DRSS)	127
-		Description	
	5.2	Terms used during this section	
	5.3	Pre-requisites, restrictions and usage information	128
		How it works	
		5.4.1 Example	
	5.5	Configuration	
		5.5.1 Application configuration	
		5.5.2 SIP server configuration	
		5.5.2.1 IPC configuration	
		5.5.2.2 Fault tolerance configuration options	
	5.6	API for resilient SIP service	
	2.0	5.6.1 sip_rss_get_server_details() – get resilient server status	
		5.6.2 sip_rss_maintenance() – initiate a takeover from an application	
		5.6.3 sip_target_refresh() – update the route set for a call	
	5.7	Events for resilient SIP service	
	5.7	5.7.1 Global events	
		5.7.2 Call events	

## aculab

Appendix A: SIP specific events	139
Appendix B: Raw SDP usage	142
Appendix C: Receipt of raw SIP messages	143
Appendix D: Using TLS to provide security	145
Appendix E: Using the Tel URI scheme	148
Appendix F: Quality Of Service for Windows(DSCP)	149

## 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 <u>www.aculab.com</u>. 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  $\underline{sip\_openout}()$ , it is necessary to use  $call\_hold()$  as  $\underline{sip\_hold}()$  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 <code>call\_ and sip\_</code> instances of the same function, for example, with <code>call/sip\_feature\_send()</code>, it is necessary to choose the routine consistent with the method of opening. So, if a call is created using <code>call\_openout()</code> then use <code>call\_feature\_send()</code>, else if the call is created using <code>sip\_openout()</code> then <code>use sip\_feature\_send()</code>.

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 <u>sip\_openout()/sip\_openin()</u>.



## **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 <code>port\_id</code> is required to facilitate this, not to be confused with those <code>port\_ids</code> discovered using Aculab's standard hardware-centric port discovery mechanism (as used by the generic API). <u>sip open port()</u> provides this <code>port\_id</code>.

As the size of the details information of a SIP call cannot be, generally, known in advance of a details call, a fully expressive  $\underline{sip\_details}()$  function is only possible if the implementation dynamically allocates the structures required after the query to the protocol stack. A function,  $\underline{sip\_free\_details}()$  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_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 credential s(), 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.

 $\underline{sip\_recall}()$  and  $\underline{sip\_disconnect}()$  implement redirection in SIP. These may only be called on a call created by  $\underline{sip\_openout}()$  (in the case of  $\underline{sip\_recall}()$ ) and  $\underline{sip\_openin}()$  (in the case of  $\underline{sip\_disconnect}()$ ).



## 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 164, whereas the GCC uses 11.

## 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 <u>sip sub\_subscriber()</u> or <u>sip\_sub\_notifier()</u> both of which return an ACU\_CALL\_HANDLE which is to be used in subsequent API calls relating to that particular subscription. <u>sip\_details()</u> 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 <u>sip details()</u> 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 <code>\$ACULAB\_ROOT/cfg</code> directory. It is not provided with the distribution, but a default configuration file may be generated by running <code>sipserv -g</code> 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 <code>\$ACULAB\_ROOT/log/sipserv.log</code> 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. TLS requires the AIT component SIP\_TLS ; if this is not present please contact support@aculab.com for further information. 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.

For loopback testing purposes this can be a key-cert.pem file. This is a combination of a key and certificate, generated by merging a key and a .pem certificate file, with the private key at the top of the file, using tools available on the Internet.

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 to use, obtained from the server the application is going to interact with. However, for test purposes, it is possible to generate these certificates using tools available on the Internet where a cacert.pem file could be used. 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 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.

NOTE

If no certificate from the server is available TLS is not possible.



CipherList = <TLS v1.2 valid list of ciphers>

The value should be a space or colon separated list of supported ciphers and associated parameters – leave empty for default settings. For example: @SECLEVEL=0 kEECDH+ECDSA kEECDH kEDH HIGH MEDIUM +3DES +SHA !RC4 !aNULL !eNULL !LOW !MD5 !EXP

CipherSuites = <TLS v1.3 valid list of cipher suite collections>

The value should be a space or colon separated list of supported collections of cipher suites – leave empty for default settings. For example;

TLS\_AES\_256\_GCM\_SHA384:TLS\_CHACHA20\_POLY1305\_SHA256:TLS\_AES\_12 8\_GCM\_SHA256.

DHGroups = <valid list of Diffie-Hellman groups>

The value should be a list of Diffie-Hellman groups supported for shared secret exchange – leave empty for default settings. Not recommended in normal use – it should be possible to achieve the desired settings via the <code>CipherList</code> or <code>CipherSuites</code> options.

TLS\_DH512File = <absolute path to file>

The value should be an absolute path to a 512-bit Diffie-Hellmann file. Not recommended for use with OpenSSL 3. 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. Not recommended for use with OpenSSL 3. Please refer to Appendix D for details.



## 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 <code>\$ACULAB\_ROOT/cfg</code> 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 "endianness", 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 ' $_{29hG4bK}$ ') 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. <u>user@domain</u>) rather than just the user.

RaiseAcceptForProgress - By default, EV\_WAIT\_FOR\_ACCEPT is not raised after <u>sip\_progress()</u> 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.

#### $DelayStart = \langle N \rangle$

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 = $\langle N \rangle$

The service uses separate threads to resolve remote addresses asynchronously. This results in less blocking in the associated API calls in the application. This parameter

determines the number of threads. If zero is specified here, the number of threads will be one.

Asyncoop – 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.

DisableTrying - when set to 1 prevents sipserv from automatically sending the '100 Trying' response; the application will be responsible for sending this response using sip\_send\_invite\_response() with sip\_code = 100 (all other parameters are set to 0). When set to 0 (the default setting) 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.

PayloadParams - when set to 1 allows the entire payload name string to be returned if it contains more than one '/'. For example:

a=rtpmap:100 opus/8000/2

then ACU PAYLOAD.rtp payload name='opus/8000/2'

Note: for traditional codec types the payload name is returned in the normal fashion:

a=rtpmap:8 PCMA/8000

then ACU PAYLOAD.rtp\_payload\_name='PCMA'

When set to 0 (the default setting) has a behaviour exhibited as before whereby only the name which occurs before the first '/' is returned in rtp\_payload\_name:

a=rtpmap:100 opus/8000/2

then ACU\_PAYLOAD.rtp\_payload\_name='Opus'

a=rtpmap:8 PCMA/8000

then ACU PAYLOAD.rtp payload name='PCMA'

LegacySetSipProxy - when set to 1 this enables legacy behaviour for the ipt\_set\_sip\_proxy() method of the Aculab IP Telephony API: if the proxy port is set to zero then 5060 will be used and if the proxy transport\_type is set to zero then ACU\_SIP\_TRANSPORT\_UDP will be used. When set to 0 (the default setting) if the proxy port is set to zero then no port will be specified in the SIP URL representing the outbound proxy and if the proxy transport\_type is set to zero then ACU\_SIP\_TRANSPORT\_ANY will be used - enabling a search for NAPTR and/or SRV records during DNS lookup of the proxy address.

LegacyQuerySipProxy – when set to 1 this enables legacy behaviour for the ipt\_query\_sip\_proxy() method of the Aculab IP Telephony API: the proxy address returned may be a numeric IP address even if the current outbound proxy address had previously been specified as a FQDN in a call to ipt\_set\_sip\_proxy(): an attempt will be made to resolve the FQDN to an IP address. When set to 0 (the default setting) the outbound proxy address will be returned without modification.

## 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:

 $\label{eq:tingtrace} \ensuremath{\texttt{TingTrace}} = < \ensuremath{\texttt{N}} > - \ensuremath{\texttt{If}} \ensuremath{\texttt{tingTrace}} \ensuremath{\texttt{If}} \ensuremath{\texttt{IingTrace}} \ensuremath{\texttt{IingTrace}} \ensuremath{\texttt{Iog}} \ensuremath{\texttt{will}} \ensuremath{\texttt{be}} \ensuremath{\texttt{cott}} \ensuremath{\texttt{Iog}} \ensuremath{\texttt{fingTrace}} \ensuremath{\texttt{Iog}} \ensuremath{\texttt{mill}} \ensuremath{\texttt{log}} \ensuremath{\texttt{IingTrace}} \en$ 

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



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



API	Description
<pre>sip_sub_fetch()</pre>	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  $\texttt{sip_open_port()}$  may be closed using <code>call\_close\_port()</code> .

#### **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 <code>call\_openout()</code> for SIP makes many assumptions on behalf of the application in the setup of the outbound <code>INVITE</code> message. In the extended API, the application is able to affect the format of the <code>INVITE</code> 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
```

l		
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	<pre>app_context_token;</pre>	/* IN */
ACU_EVENT_QUEUE	queue_id;	/* IN */
ACU_UINT	request_notification_mask;	/* IN */
ACU_UINT	response_notification_mask;	/* IN */
ACU_UINT	<pre>enable_midcall_response_mask;</pre>	/* IN */
char*	request_uri;	/* IN */
char*	custom_headers;	/* IN */



	ACU RAW MESSAGE BODY*	me
	ACU MEDIA OFFER ANSWER*	me
	ACU UINT	са
	char	tr
	char*	ca
	char*	us
}	SIP_OUT_PARMS;	

message\_bodies; media\_offer\_answer; call\_options; transport\_type; callid; user;

/ ^	ΤIN	~ /
/*	ΙN	*/
/*	ΙN	*/
/*	ΙN	*/
/*	IN	*/
/*	ΙN	*/

/+ TNT +/

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 <u>sip open port()</u>. 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 <code>destination\_addr</code> 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
```

```
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 Contact header format.

#### For example:

```
outx.contact_address = "sip:user@192.168.0.9";
outx.contact_address = "<sip:user@192.169.0.9>"
outx.contact_address = "<sip:user@192.169.0.9;transport=tcp>"
outx.contact_address = "<sip:user@192.169.0.9;transport=tcp>;expires=5600
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 <u>sip\_details()</u> 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

$= 0 \times 00000001$ , (see note 1)
$= 0 \times 00000002$ , (see note 1)
$= 0 \times 00000004$ , (see note 1)
$= 0 \times 0 0 0 0 0 0 0 8$ ,
= 0x0000010,



ACU SIP REGISTER NOTIFICATION	$= 0 \times 00000020$ , (see note 2)
ACU SIP SUBSCRIBE NOTIFICATION	$= 0 \times 00000040$ , (see note 2)
ACU_SIP_OPTIONS_NOTIFICATION	$= 0 \times 0 0 0 0 0 0 80$ ,
ACU_SIP_BYE_NOTIFICATION	$= 0 \times 00000100$ ,
ACU_SIP_MESSAGE_NOTIFICATION	$= 0 \times 00000200$ ,
ACU_SIP_UPDATE_NOTIFICATION	$= 0 \times 0 0 0 0 0 4 0 0$ ,
ACU_SIP_PRACK_NOTIFICATION	$= 0 \times 00000800$ , (see note 1)
ACU_SIP_REFER_NOTIFICATION	$= 0 \times 00001000$ , (see note 1)
ACU_SIP_INITIAL_ACK_NOTIFICATION	$= 0 \times 00002000$ , (see note 1)
ACU_SIP_REINVITE_ACK_NOTIFICATION	$= 0 \times 00004000$ , (see note 1)
ACU_SIP_TRANSFER_ACK_NOTIFICATION	$= 0 \times 00008000$ , (see note 1)
} ACU SIP MESSAGE NOTIFICATION MASKS;	

note 1. These message types are not valid for use with <code>enable\_midcall\_response\_mask</code>. note 2. These message types are only valid for use with out of dialog API functions.

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.

#### 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 notes 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 <u>sip\_send\_response()</u>. 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 <u>sip\_disconnect()</u>. 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 <u>section 4</u> 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 <u>sip_send_request()</u> for further
details
outx.message_bodies = &arm;
```

#### media\_offer\_answer

This field is a pointer to an <u>ACU\_MEDIA\_OFFER\_ANSWER</u> 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 <u>ACU\_MEDIA\_OFFER\_ANSWER</u> 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 ACU_RAISE_HOLD_EVENT_FOR_C_EQUAL_ZERO ACU_USE_C_EQUAL_ZERO_FOR_HOLD ACU_DISABLE_EARLY_MEDIA ACU_RAISE_HOLD_EVENT_FOR_INACTIVE ACU_USE_INACTIVE_FOR_HOLD ACU_DISABLE_TRANSFER_ON_REFER ACU_USE_MEDIA_DESCRIPTION_FOR_SESSION ACU_DO_NOT_USE_PROXY_IF_SET ACU_USE_ASYNC_MODE ACU_NO_HOLD_EVENTS ACU_RAISE_FORKED_EVENT	=0x0000001, =0x0000002, =0x00000004, =0x00000008, =0x00000020, =0x00000020, =0x00000040, =0x00000100, =0x00000100, =0x00000200, =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 '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 <code>ipt\_set\_sip\_proxy()</code> 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 <code>INVITE</code> will be sent to the configured outbound proxy. When this flag is enabled, the initial <code>INVITE</code> 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 <code>request\_uri</code> or if a <code>Route</code> header is supplied in <code>custom\_headers</code>). 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 <u>sip\_details()</u> 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 client 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.

### user

If a set of credentials has multiple users for the same realm, this optional field allows the client to specify the preferred username in an Authentication header when responding to a challenge with a realm with multiple users. Otherwise if one is not specified the user matching the one obtained from the 'From' header would be used else the first user in the list matching the realm will be selected. Note if a user is defined but is not in the set of credentials the call will just fail with '401 Unauthorized' response and sipserv will not attempt to send a reINVITE with an Authentication header.

### **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 mol;
memset(&mol, 0, sizeof(ACU MEDIA OFFER ANSWER));
// setup the offer answer - then assign to outx
outx.media offer answer=&mol;
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_OLONG
ACU_CALL_HANDLE
                                         handle;
                                                                                                      /* OUT */
       ACU_PORT_ID
                                          net;
                                                                                                      /* IN */
       ACU_INTnet,/* IN */ACU_INTcnf;/* IN */ACU_EVENT_QUEUEqueue_id;/* IN */ACU_ACTapp_context_token;/* IN */ACU_UINTrequest_notification_mask;/* IN */ACU_UINTresponse_notification_mask;/* IN */ACU_UINTenable_midcall_response_mask;/* IN */ACU_UINTcall_options;/* IN */
       ACU_UINT
ACU_UINT
                                                                                                     /* IN */
        ACU UINT
                                            call options;
     } 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	$= 0 \times 00000002$ , (see note 1	)



ACU SIP TRANSFER INVITE NOTIFICATION	= 0x00000004, (see note 1)
ACU SIP INFO NOTIFICATION	$= 0 \times 00000008$ ,
ACU SIP NOTIFY NOTIFICATION	$= 0 \times 00000010$ ,
ACU_SIP_REGISTER_NOTIFICATION	= 0x00000020, (see note 2)
ACU_SIP_SUBSCRIBE_NOTIFICATION	$= 0 \times 00000040$ , (see note 2)
ACU_SIP_OPTIONS_NOTIFICATION	$= 0 \times 0 0 0 0 0 0 80$ ,
ACU_SIP_BYE_NOTIFICATION	$= 0 \times 00000100$ ,
ACU_SIP_MESSAGE_NOTIFICATION	$= 0 \times 0 0 0 0 0 2 0 0$ ,
ACU_SIP_UPDATE_NOTIFICATION	$= 0 \times 0 0 0 0 0 4 0 0$ ,
ACU_SIP_PRACK_NOTIFICATION	$= 0 \times 00000800$ , (see note 1)
ACU_SIP_REFER_NOTIFICATION	$= 0 \times 00001000$ , (see note 1)
ACU_SIP_INITIAL_ACK_NOTIFICATION	$= 0 \times 00002000$ , (see note 1)
ACU_SIP_REINVITE_ACK_NOTIFICATION	$= 0 \times 00004000$ , (see note 1)
ACU_SIP_TRANSFER_ACK_NOTIFICATION	$= 0 \times 00008000$ , (see note 1)
<pre>} ACU_SIP_MESSAGE_NOTIFICATION_MASKS;</pre>	

note 1. These message types are not valid for use with <code>enable\_midcall\_response\_mask</code>. note 2. These message types are only valid for use with out of dialog API functions.

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.

#### $response\_notification\_mask$

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 notes 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 <u>sip\_send\_response()</u>. 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 <u>sip\_disconnect()</u>. 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
                                     =0x0000001,
 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
ACU_DO_NOT_USE_PROXY_IF_SET
                                         =0 \times 00000040,
                                         =0 \times 00000100,
 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 <a href="mailto:sip\_details(">sip\_details()</a>) 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 2000K response.

In the generic API, the implementation of  $call_accept()$  for SIP builds the SDP body of the 2000K. 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 <u>sip\_details()</u> subsequent to the EV\_INCOMING\_CALL\_DET. However, the invocation of <u>sip\_accept()</u> may be delayed until after <u>sip\_incoming\_ringing()</u> or <u>sip\_progress()</u> has been called.

With the exception of handle, media\_offer\_answer is the only mandatory field in <a href="mailto:sip\_accept">sip\_accept</a>(). 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
 ACU\_CALL\_HANDLE
 char\*
 char\*
 ACU\_RAW\_MESSAGE\_BODY\*
 ACU\_MEDIA\_OFFER\_ANSWER
} 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 <u>ACU\_MEDIA\_OFFER\_ANSWER</u> 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 <u>section 4.6</u> 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 mol;
memset(&mol, 0, sizeof(ACU_MEDIA_OFFER_ANSWER));
// setup the offer answer - then assign to accx
accx.media_offer_answer=&mol;
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 function should be regarded as *not re-entrant* in the context of a particular call handle – ie. a call to this function in response to an event must complete fully before a further call to this function is made in response to a subsequent event on the same call handle. This function must be called in response to the following events only, once per event:

• 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 <u>sip\_openout()</u> and <u>sip\_openin()</u>. 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 <u>Appendix C</u> for more details on the collection of raw SIP messages.

- EV\_MEDIA (see Appendix A)
- EV\_MEDIA\_PROPOSE (see Appendix A)
- 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). Upon return from *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 <u>sip\_feature\_send()</u> 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. Upon return from *sip\_details redirect\_info* will be populated.
- EV\_SIP\_SUBSCRIBED (see Appendix A)
- EV\_SIP\_SUBSCRIPTION\_PENDING(see Appendix A)
- EV\_SIP\_SUBSCRIPTION\_CANCELLED (see Appendix A)
- EV\_SIP\_SUBSCRIPTION\_REQUEST (see Appendix A)
- EV\_SIP\_SUBSCRIPTION\_NOTIFICATION (see Appendix A)
- EV\_SIP\_SUBSCRIPTION\_REFRESH (see Appendix A)
- EV\_SIP\_SUBSCRIPTION\_REFRESH\_FAILED (see Appendix A)

# 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
ACU CALL HANDLE
ACU INT
ACU INT
char*
ACU ACT
char
ACU SIP MESSAGE*
ACU MEDIA OFFER ANSWER*
ACU MEDIA SESSION*
char
char
ACU REDIRECT INFO*
ACU SUBSCRIPTION INFO*

```
size:
                                               /* IN */
handle;
valid;
                                              /* OUT */
calltype;
                                              /* OUT */
destination_addr;
                                              /* OUT */
                                              /* OUT */
originating_addr;
                                              /* OUT */
connected addr;
destination display name;
                                              /* OUT */
originating_display_name;
                                              /* OUT */
app_context_token; /* OUT*/
reliable_provisional_response_supported; /* OUT */
sip_message;
media offer answer;
                                              /* OUT */
media session;
                                              /* OUT */
                                             /* OUT */
transfer_refer_received;
transport type;
                                              /* OUT */
redirect info;
                                              /* OUT */
                                              /* OUT */
subscription info;
```

} SIP\_DETAIL\_PARMS;

# Input parameters

The <u>sip\_details()</u> 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  $\tt From:$  and  $\tt To:$  headers of the call setup  $\tt INVITE$  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 <u>ACU\_SIP\_MESSAGE</u> structure. This will be populated by the <u>sip\_details()</u> function if the call has queued a received SIP message as indicated by the <u>EV\_DETAILS</u> event. Please refer to <u>section 4.8</u> for further information on this field.

# media\_offer\_answer

This field is a pointer to an <u>ACU\_MEDIA\_OFFER\_ANSWER</u> 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 OF

 ${\tt ev}\_{\tt ext}\_{\tt transfer}\_{\tt information}.$  Please refer to section 4.6 for further information on this field.

### media\_session

This field points to an <u>ACU\_MEDIA\_SESSION</u> structure. This is populated by <u>sip\_details()</u> subsequent to receipt of the <u>EV\_MEDIA</u> event. An application should use this information to affect how to control media streams relevant to the call. Please refer to <u>section 4.7</u> 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 <u>ACU\_REDIRECT\_INFO</u> structure. This is populated by <u>sip\_details()</u> subsequent to receipt of the <u>EV\_EXTENDED\_DIVERSION</u> event. An application should use this information to supply the input parameters for <u>sip\_recall()</u>. Please refer to <u>section 4.10</u> for further information on this field.

### $subscription_info$

This field points to an <u>ACU\_SUBSCRIPTION\_INFO</u> structure. This is populated by <u>sip\_details()</u> when subscription information is available. For more information on

when subscription events can occur and when to use  $\underline{sip_details}()$  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  $\underline{sip}$  details(). It is essential that this routine is called on any  $\underline{sip}$ \_DETAIL\_PARMS\* received by  $\underline{sip}$  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 <u>sip free details</u>() function takes a pointer,  $detail_parms$ , to a structure SIP\_DETAIL\_PARMS. The structure has previously been populated by a successful call to <u>sip details</u>().

# **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 */
ACUINT	use 183 response for early media;	/* IN */
ACUINT	send reliable provisional response;	/* IN */
char*	contact address;	/* IN */
ACU MEDIA OFFER ANSWER*	media offer answer;	/* IN */
<pre>} SIP INCOMING RINGING PARMS;</pre>		

# Input parameters

The <u>sip\_incoming\_ringing()</u> 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  $\underline{sip\_incoming\_ringing}$  () 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 <u>ACU\_MEDIA\_OFFER\_ANSWER</u> specifying the settings relevant to starting an early media session with the caller. Please refer to <u>section 4.6</u> 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 <u>EV\_WAIT\_FOR\_ACCEPT</u> state, whereupon <u>sip\_accept()</u> will connect the call. The application can therefore control the number of ring cadences by delaying the <u>sip\_accept()</u> function.

# Example usage

```
SIP_INCOMING_RINGING_PARMS ringx;
INIT_ACU_STRUCT(&ringx);
ringx.handle = hndl;
ringx.send_early_media = 1;
ACU_MEDIA_OFFER_ANSWER mol;
memset(&mol, 0, sizeof(ACU_MEDIA_OFFER_ANSWER));
// setup the offer answer - then assign to ringx
ringx.media_offer_answer = &mol;
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

ł	
	ACU_ULONG
	ACU_CALL_HANDLE
	ACU INT
	char*
	ACU MEDIA OFFER ANSWER*
}	SIP PROGRESS PARMS;

size; handle; /\* IN \*/ send\_reliable\_provisional\_response; /\* IN \*/ contact\_address; /\* IN \*/ media\_offer\_answer; /\* IN \*/

### Input parameters

The <u>sip progress</u>() 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 <u>ACU\_MEDIA\_OFFER\_ANSWER</u> specifying the settings relevant to starting an early media session with the caller. Please refer to <u>section 4.6</u> 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 mol;
memset(&mol, 0, sizeof(ACU\_MEDIA\_OFFER\_ANSWER));
// setup the offer answer - then assign to progx
progx.media\_offer\_answer = &mol;
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 */
    ACU_STRING_LIST* 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 <u>ACU\_MEDIA\_OFFER\_ANSWER</u> specifying the settings relevant to starting a media session with the caller. Please refer to <u>section 4.6</u> 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 <u>section 4.1</u> 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 mol;
memset(&mol, 0, sizeof(ACU_MEDIA_OFFER_ANSWER));
// setup the offer answer - then assign to responsex
responsex.media_offer_answer = &mol;
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
ACU_ULONG
                           handle;
                                                         /* IN */
                       nanure,
feature_type;
                                                         /* IN */
 union
 {
   struct
   {
    ACU INT
                                command;
                                                         /* IN */
   } hold;
   struct
   {
                                failure code;
     ACU INT
                                                         /* IN */
     ACU MEDIA OFFER ANSWER*
                                media_offer_answer;
                                                         /* IN */
   } transfer;
  } feature;
} SIP FEATURE DETAIL PARMS;
```

# Input parameters

The <u>sip feature send()</u> 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 2000K 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 <u>ACU\_MEDIA\_OFFER\_ANSWER</u> 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 2000K 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;
 <u>ACU\_MEDIA\_OFFER\_ANSWER media\_o</u>
 char\* custom\_
} SIP MEDIA PROPOSE PARMS;

slze; handle; /\* IN \*/ media\_offer\_answer; /\* IN \*/ custom\_headers; /\* IN \*/

# 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 <u>section 4.6</u> 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

### $custom\_headers$

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 2000K, 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_ACCEP	I_PARMS	
{		
ACU_ULONG	size;	
ACU_CALL_HANDLE	handle;	/* IN */
ACU_MEDIA_OFFER_ANSWER	<pre>media_offer_answer;</pre>	/* IN */
<pre>} SIP_MEDIA_ACCEPT_PARMS;</pre>		

# 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 <u>section 4.6</u> 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 2000k 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

 $\label{eq:media_request_proposal_parms} \ to \ a \ \texttt{SIP_MEDIA_REQUEST_PROPOSAL_PARMS} \ structure.$  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 2000K 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);

/\* IN \*/

```
typedef struct tSIP_MEDIA_REJECT_PROPOSAL_PARMS
{
    ACU_ULONG size;
    ACU_CALL_HANDLE handle; /* IN */
    union
```

{
 ACU\_INT sip\_code;
} 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 2000K; the call will be cleared by a subsequent BYE/2000K 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);
```

<pre>typedef struct tSIP_SEND_REQ {</pre>	QUEST_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*	<pre>media_offer_answer;</pre>	/*	IN */	
<pre>} SIP_SEND_REQUEST_PARMS;</pre>				

### 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
ACU_SIP_MESSAGE_NOTIFY
                                       =1,
                                       =2,
     ACU SIP MESSAGE REGISTER
                                       =3,
     ACU_SIP_MESSAGE_SUBSCRIBE
                                       =4,
                                       =5,
     ACU_SIP_MESSAGE_OPTIONS
     ACU_SIP_MESSAGE_UPDATE
ACU_SIP_MESSAGE_MESSAGE
                                       =6.
                                       =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 out of dialog API functions.

### 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 <u>section 4.1</u> for further details in the setup of this structure.

media\_offer\_answer

This field is a pointer to an <u>ACU\_MEDIA\_OFFER\_ANSWER</u> 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 <u>ACU\_MEDIA\_OFFER\_ANSWER</u> 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 <code>upDATE</code> method the <code>sip\_send\_response()</code> can be called during call set-up before the <code>INVITE</code> 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;	/*	<pre>mandatory */</pre>
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*	<pre>media_offer_answer;</pre>	/*	IN */

} SIP\_SEND\_RESPONSE\_PARMS;

# NOTE

To send a SIP response to an initial INVITE see sip\_send\_invite\_response()

# 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 <u>section 4.1</u> for further details in the setup of this structure.



### media\_offer\_answer

This field is a pointer to an <u>ACU MEDIA OFFER ANSWER</u> 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 <u>ACU MEDIA OFFER ANSWER</u> 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, 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_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

ι							
	ACU_ULONG	size;		/*	IN	*/	
	char*	realm;		/*	IN	*/	
	char*	user;		/*	IN	*/	
	char*	passwo	ord;	/*	IN	*/	
	ACU INT	retrie	es;	/*	IN	*/	
}	SIP_ANSWER_	CHALLENGE	CREDENTIALS	PARI	MS;		

# 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.

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  ${\tt SIP}_{\tt ANSWER}_{\tt CHALLENGE}_{\tt CREDENTIALS}_{\tt PARMS}$  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 */
    ACU_INT generic_cause; /* 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 <u>section 4.1</u> 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.



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.

# Input parameters

The sip\_recall () function takes a pointer <code>recall\_parms</code> to a <code>SIP\_RECALL\_PARMS</code> 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  $\underline{sip} \ openout()$  (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:.

# 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.

```
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 <code>new\_trusted.pem</code> and <code>new\_client.pem</code> 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:

# 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 **Of** <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 qsiqBody;
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

ι.			
	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_N	RESPONSE_PARMS;	

# Input parameters:

The function takes a pointer to a <u>sip\_send\_out\_of\_dialog\_response\_parms</u> structure. The structure must be set up in the following way.

#### sip\_code

Mandatory. Valid SIP response code, such as 200 for 2000K 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.

```
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*/
	ACUUINT	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.

```
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.

```
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
{
                         size;
message;
port_id;
transaction_id;
 ACU LONG
                                                /*NOT USED*/
/*IN*/
  ACU SIP MESSAGE
 ACU_PORT_ID
                                                /*OUT*/
 ACU POINTER
                                                 /*OUT*/
 ACU UINT
                           cause;
} 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	=	Ο,
	ACU_OOD_REQUEST_TIMEOUT	=	1,
	ACU_OOD_TCP_CONNECT_FAILED	=	2,
	ACU_OOD_SSL_ERROR	=	З,
	ACU_OOD_SSL_PEER_CERT_NOT_TRUST	ED=	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.

```
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
	ACU_SIP_MESSAGE
	ACU_PORT_ID
	ACU POINTER
	ACUUINT

size; message; /\*NG port\_id; /\*IN transaction\_id; /\*OG cause; /\*OG

/\*NOT USED\*/ /\*IN\*/ /\*OUT\*/ /\*OUT\*/} SIP READ MESSAGE PARMS;

# Input parameters

This function takes a pointer to a <code>sip\_read\_message\_parms</code> structure, which has previously been populated by a call to either <code>sip\_read\_request()</code> or <code>sip\_read\_response()</code>.

## **Return values**

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

```
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

ί.			
	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*/
	ACUINT	response notification;	/*IN*/
	ACU INT	enable response;	/*IN*/
	ACUINT	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*/
1	ATD AUD AUDOADTDED DADM		

} 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 <code>subscribe</code> requests sent by this subscriber. On receipt of a <code>subscribe</code> response, an application that has set this value to 1 will receive an <code>EV\_DETAILS</code> event and a subsequent call to <code>sip\_details()</code> will collect the entire message. There is no equivalent for request notifications as <code>NOTIFY</code> 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 <u>sip send response()</u>.

#### 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 <u>section 4</u> 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 <u>sip_send_request()</u> 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 <code>AsyncSubscribe</code> 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 <code>EV\_SIP\_SUBSCRIPTION\_CANCELLED</code> indicates that an error occurred.

```
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 ACU_CALL_HANDLE ACU_PORT_ID	size; handle; net;	/*IN*/ /*OUT*/ /*IN*/
ACU_EVENT_QUEUE	queue_id;	/*IN*/
ACU_ACT	<pre>app_context_token;</pre>	/*IN*/
ACU_INT	<pre>request_notification;</pre>	/*IN*/
ACU_INT	response_notification;	/*IN*/
ACU_INT	<pre>enable_response;</pre>	/*IN*/
char*	event_package_token	/*IN*/
	PN(2	

} 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 <u>sip details</u>() 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
                                               /*IN*/
                           size:
  ACU CALL HANDLE
                           handle;
                                               /*IN*/
  ACU INT
                           expires;
                                               /*IN*/
  char*
                           contact address;
                                               /*IN*/
  char*
                           custom headers;
                                               /*IN*/
                           event id param;
                                               /*IN*/
  char*
                           message bodies;
  ACU RAW MESSAGE BODY*
                                               /*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 <u>sip\_details()</u> 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 <u>section 4.1</u> for further details of the setup of this structure <u>ACU\_RAW\_MESSAGE\_BODY</u>.

#### For example:

```
ACU_RAW_MESSAGE_BODY arm;
memset(&arm, 0, sizeof(ACU_RAW_MESSAGE_BODY));
// setup the message body structure - see <u>sip send request()</u> 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.

```
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
```

```
ł
```

ACU_ULONG ACU_CALL_HANDLE	size; handle;	/*IN*/ /*IN*/
char*	custom_headers;	/*IN*/
ACU_RAW_MESSAGE_BODY*	<pre>message_bodies;</pre>	/*IN*/
char	pending;	/*IN*/
char*	event_id_param;	/*IN*/
STP SUB NOTIFY PARMS:		

```
} 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 <u>ACU RAW MESSAGE BODY</u>.

#### For example:

```
ACU_RAW_MESSAGE_BODY arm;
memset(&arm, 0, sizeof(ACU_RAW_MESSAGE_BODY));
// setup the message body structure - see <u>sip_send_request()</u> 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  $\underline{sip \ sub \ accept ()}$  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.

```
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 <u>sip_send_request()</u> 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*/
                                              /*IN*/
ACU CALL HANDLE
                         handle;
ACU INT
                         dialog id;
                                              /*IN*/
ACU INT
                         response_code;
                                              /*TN*/
char*
                         reason;
                                             /*IN*/
char*
                         custom headers;
                                             /*IN*/
                                             /*IN*/
                         event id param;
char*
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 <u>ACU\_SUBSCRIPTION\_INFO</u> 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 <u>section 4.1</u> 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 <u>sip_send_request()</u> 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.

```
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 <u>sip_send_request()</u> 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 <code>call\_release()</code> 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 <u>ACU\_SUBSCRIPTION\_INFO</u> 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
```

{ AC

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

٢		
٢.		
c.		

ACU_ULONG	size;	/*IN*/
ACU_CALL_HANDLE ACU_INT	handle; dialog_id;	/*IN*/ /*IN*/
char*	custom_headers;	/*IN*/
char*	<pre>event_id_param;</pre>	/*IN*/
ACU_RAW_MESSAGE_BODY*	<pre>message_bodies;</pre>	/*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 <u>section 4.1</u> 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 <u>sip_send_request()</u> 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.

```
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 <code>isup</code> and <code>qsig</code> (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 (<u>ACU\_MEDIA\_OFFER\_ANSWER</u>) 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.

#### 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 <u>ACU\_MEDIA\_DESCRIPTION</u>, 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 <u>ACU\_MEDIA\_DESCRIPTION</u>. For example, for a linked list of *audio\_video* payloads the containing <u>ACU\_MEDIA\_DESCRIPTION</u> structure should have *media\_type* set as ACU\_AUDIO OF ACU\_VIDEO.

<pre>typedef struct acu_payload {     union     {        </pre>	
struct	
char* ACU_INT ACU_INT ACU_INT ACU_INT	<pre>rtp_payload_name; rtp_payload_number; packet_length; clock_rate;</pre>
char* } audio_video; struct	payload_specific_options;
<pre>{     char* } image; struct </pre>	<pre>image_payload_name;</pre>
ACU_INT } control; struct	dummy;
ACU_INT } application; } payload;	dummy;
<pre>struct acu_payload* } ACU_PAYLOAD;</pre>	next;

### 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

Note that if the user wishes to transmit a payload name with additional encoding parameters then the entire string including the clock rate may be presented as the rtp\_payload\_name. In the following example the clock rate defined in the string would overwrite any clock\_rate defined separately. The additional encoding parameters are presented as a multiple channel option '/2'

### For example:

"opus/8000/2" would produce an SDP offer containing "a=rtpmap:100 opus/8000/2"

When sip\_details() is called with PayloadParams set in the sipserv.cfg (See 2.9 Miscellaneous Global Settings) the entire payload string is returned in rtp\_payload\_name if more than one '/' occurs in the string thus for the example above.

### For example:

```
"a=rtpmap:100 opus/8000/2"
rtp_payload_name = "opus/8000/2"
but
"a=rtpmap:8 PCMA/8000"
rtp payload name = "PCMA"
```

### 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 <u>ACU PAYLOAD</u> structure may be an element in a list. This is enabled by one element being a pointer to another <u>ACU PAYLOAD</u>, as follows:



### next

This is a pointer to the next element in a list of <u>ACU PAYLOAD</u> 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 m=t 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	<pre>packet_length;</pre>
ACU_MISCELLANEOUS_MEDIA_ATTRIBUTE*	miscellaneous_attributes;
ACU_PAYLOAD*	payloads;
struct acu_media_description*	next;
<pre>} ACU_MEDIA_DESCRIPTION;</pre>	

### $connection\_address$

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 <u>acu\_media\_offer\_answer</u> 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" 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: udpt1.

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 <u>ACU\_MISCELLANEOUS\_MEDIA\_ATTRIBUTE</u> 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 mdl;
 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;
mdl.connection address.address="10.202.10.17";
mdl.media_direction=ACU SEND ONLY;
mdl.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;
 mdl.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 mdl;
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;
mdl.connection address.address="10.202.10.17";
mdl.media direction=ACU SEND ONLY;
mdl.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;
mdl.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));
              mmal.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 <code>offer/answer</code> is conveyed by SIP as an SDP body and the role of the <u>ACU\_MEDIA\_OFFER\_ANSWER</u> 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 <u>sip\_openout()</u>; it will also be presented these structures by <u>sip\_details()</u>.

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
    ACU_MEDIA_DESCRIPTION* media_des
    char* raw_sdp;
} ACU_MEDIA_OFFER_ANSWER;
```

connection\_address; media\_descriptions; raw\_sdp;

When the <u>ACU\_MEDIA\_OFFER\_ANSWER</u> 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 <u>ACU\_MEDIA\_DESCRIPTION</u> 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 <u>Appendix B</u> for further information on using raw SDP.

# 4.7 ACU\_MEDIA\_SESSION

An instance of <u>ACU\_MEDIA\_SESSION</u> 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 <u>sip\_details()</u> 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 <u>ACU\_MEDIA\_OFFER\_ANSWER</u> 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 <u>sent\_media</u> element of the <u>ACU\_MEDIA\_SESSION</u> 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_si	p_message
{	
unsigned char*	message;
ACU_INT	<pre>message_length;</pre>
ACU_UINT	message_handle;
<pre>} ACU_SIP_MESSAGE;</pre>	

#### 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  $\underline{sip\_details}$  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 <u>ACU\_SIP\_MESSAGE</u> and <u>ACU\_REDIRECT\_INFO</u> provides <u>sip\_details()</u> 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  $\underline{sip\_sub\_accept}()$ . Note that  $\underline{ERR\_PARM}$  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 <u>sip\_sub\_release()</u> 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  $\underline{sip\_sub\_notify}()$  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.







# 5.5 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 <u>Distributed SIP Settings</u>. 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 <code>sipplugin.cfg</code> 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 <u>Configuring the SIP service</u>.

### 5.5.2.1IPC configuration

The SIP server must be configured to listen for connections from remote applications, as described in <u>Distributed SIP Settings</u>, and these settings must correspond with the settings that have been chosen for the application.

### 5.5.2.2Fault 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
                                                                        /* OUT */
         ACU CHAR
                                     address[ACU MAX IP ADDRESS];
         ACU USHORT
                                    poil;
state;
heartbeat_present;
server_alive;
calls;
                                                                        /* OUT */
                                    port;
        ACU_UINT
ACU_CHAR
                                                                        /* OUT */
                                                                        /* OUT */
                                                                        /* OUT */
         ACU CHAR
                                                                        /* OUT */
         ACU UINT
                                                                        /* OUT */
        ACU UINT
                                     error;
        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

### 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:

Infinite timeout - takeover will only happen when all calls in progress have reached the connected state.
 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 <code>RSS\_NOT\_CONFIGURED</code> state to a <code>RSS\_ACTIVE</code> OF <code>RSS\_PASSIVE</code>

### 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

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. The application must now call <u>sip\_details()</u>. This function will return a pointer to an <u>ACU\_MEDIA\_SESSION</u> structure. This structure contains three elements, sent SDP, received SDP and a flag indicating which of these was the <u>answer</u> 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 <u>answer</u> SDP is the payload to be used, initially, for both parties to send/receive media with.

### 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 (in contrast to the classic inbound call setup in which EV\_INCOMING\_CALL\_DET will flag the presence of this offer). This event may also be raised if an initial 2000K is received for an outgoing call, which was initiated by an INVITE with no SDP. The application must call <u>sip\_details()</u> 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() Of 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 preconfigured 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 preconfigured memory limit to indicate high levels of traffic has been exceeded by the SIP service. See configuring the <u>Overload Monitor</u> 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 <u>Overload Monitor</u> 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().

### EV\_SIP\_WAIT\_FOR\_NOTIFIER

This event is raised on a SIP subscription after a new subscriber has been created using sip\_sub\_subscriber().

### EV\_SIP\_SUBSCRIBED

sip\_details() must 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 <u>sip\_sub\_accept().sip\_sub\_notify()</u> must be called in order to conform to RFC 3265.
- The application accepted an initial SUBSCRIBE with a 202 Accepted and subsequently called <u>sip sub notify()</u> 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() must always be called here.

This event is raised on a SIP subscription given the following conditions:

- The application has accepted an incoming <code>subscribe</code> request with a call to <a href="mailto:sip\_sub\_accept">sip\_sub\_accept</a>() with the <code>acknowledge</code> field set to 1. <a href="mailto:sip\_sub\_notify">sip\_sub\_notify</a>() 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() must 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. <u>sip sub\_release()</u> should be called here if the *dialog\_count* is 0.
- A subscriber has received a non-2xx response to the initial SUBSCRIBE request. <u>sip sub release()</u> should be called here if the <u>dialog\_count</u> 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.
- <u>sip sub release()</u> should be called here if the *dialog\_count* is 0.



- A notifier has received a SUBSCRIBE with an expires value of 0. <u>sip sub notify</u>() 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  $\underline{sip sub release}()$  on receipt of this event.

EV\_SIP\_SUBSCRIPTION\_REQUEST

sip\_details() must always be called here.

This event is raised on a SIP subscription when a notifier receives an initial subscribe request. The application must call either  $\underline{sip\ sub\ accept}()$  or  $\underline{sip\ sub\ cancel}()$  in response.

### EV\_SIP\_SUBSCRIPTION\_NOTIFICATION

sip details() must always be called here.

This event is raised on a SIP subscription when a subscriber receives a NOTIFY request on an existing subscription.

### EV\_SIP\_SUBSCRIPTION\_REFRESH

sip\_details() must always be called here.

This event is raised on a SIP subscription when a notifier receives a subscription request on an existing subscription. The application must call sip sub notify() here in order to conform to RFC 3265.

### EV\_SIP\_SUBSCRIPTION\_REFRESH\_FAILED

sip details() must always be called here.

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. 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 <u>ACU\_MEDIA\_OFFER\_ANSWER</u> contains a field called <u>raw\_sdp</u>. 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 <u>raw\_sdp</u> 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 <code>raw\_sdp</code> 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 <code>sipserv</code> executes no checking or validation code. Note that when the <code>sipserv</code> detects a non-NULL value for <code>raw\_sdp</code> 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 <code>sip\_openout()</code> using the abstracted structures to initially place a call and subsequently use raw SDP for a complex re-INVITE sent using <code>sip\_media\_propose()</code>. 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  $\mathtt{sip\_openout}()$  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 mol;
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
mol.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
                                                                           = 0 \times 00000010,
    ACU_SIP_REGISTER_NOTIFICATION
ACU_SIP_SUBSCRIBE_NOTIFICATION
                                                                    = 0x00000020, (see note 2)
= 0x00000040, (see note 2)
= 0x0000080,
                                                                           = 0 \times 00000020, (see note 2)
    ACU SIP OPTIONS NOTIFICATION
    ACU SIP BYE NOTIFICATION
                                                                         = 0 \times 00000100,
    ACU_SIP_MESSAGE_NOTIFICATION
                                                                         = 0 \times 00000200.
    ACU_SIP_UPDATE_NOTIFICATION
ACU_SIP_PRACK_NOTIFICATION
                                                                           = 0 \times 00000400,
                                                                         = 0 \times 00000800, (see note 1)
    ACU SIP REFER NOTIFICATION
                                                                         = 0x00001000, (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 <code>enable\_midcall\_response\_mask</code>. note 2. These message types are only valid for use with out of dialog API functions.

A combination of these flags may be written to the *request\_notification\_mask*, *response\_notification\_mask* and *enable\_incall\_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_OUT_PARMS outx;
INIT_ACU_STRUCT(&outx);
outx.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 must be collected by the application using the <u>sip\_details()</u> function call and will be presented in the *sip\_message* field of SIP\_DETAIL\_PARMS structure.

Note that not calling <u>sip\_details</u>() 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
MBEGA1UECBMKU29tZS1TdGF0ZTEhMB8GA1UEChMYSW50ZXJuZXQgV21kZ210cyBQ
dHkgTHRkMB4XDTA2MDcxMjA3NDc1MloXDTA2MDgxMTA3NDc1MlowRTELMAkGA1UE
BhMCQVUxEZARBgNVBAgTC1NvbWUtU3RhdGUxITAfBgNVBAoTGEludGVybmV0IFdp
ZGdpdHMgUHR5IEx0ZDCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEA1X/aSX2G
7VRbQgEvDY02ALFrwAUF3Ptc4IBY4ouY9oEEFYOpsuQcZUAfEMutnQ8+n5/hwKKG
+F1+Tz7u06q+zkN2sE9EvU5xBVCoIM9GP+mIgOqSgNSdA1zf7c3bnJXssrGK1jJg
B7DCZoaKqn6udhVwSxtJAbB4TfEiI+0jiHsCAwEAAaOBnzCBnDAdBgNVHQ4EFgQU
WN7cMk4ie3WI44p2j4SdSv+xqvgwbQYDVR0jBGYwZIAUWN7cMk4ie3WI44p2j4Sd
Sv+xqvihSaRHMEUxCzAJBgNVBAYTAkFVMRMwEQYDVQQIEwpTb211LVN0YXR1MSEw
HwYDVQQKExhJbnR1cm51dCBXaWRnaXRzIFB0eSBMdGSCAQAwDAYDVR0TBAUwAwEB
/zANBgkqhkiG9w0BAQQFAAOBgQAsPBL+SyO4tytnjDx874qap21SiBj6ha8DrLLB
TY1W2KHzKdjv1tAweF9z914HNQU594fB7XtDa1VMT3V021EKJk+NOd3Wnbyn6Chy
KjZ4EOcE9BZjuSmvJIGIVtRrN2LrU7cSHFxoY8FUtDYpox11UoiiZmVVuTrxv/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-----
MIICXQIBAAKBqQD21vP2blSTmE1+6hBr+zVL8PukYDJty4vPRlnsTqaEmOz1ryCt
c3fXfGpxvXKoZ42Z8BjE4oE7IrJDIoOVJkjAhW1JAxawt18Jo0nh59aAJMMiudkl
wnMkLx2WLq79MRs/bl6epIQMcWrUvrqEwvL+vpwwtwwql5/xFimPqlPtFwIDAQAB
AoGAPhv5aNmTTfWulVvpNPl6DB2vlFBygzsgtTm4DpAk2wMVtDAfH/EUf18kBG/+
ODKM9PgH1RekCzwLAGPiFgAk9HJOKwrxCWI2OSwAAE5zMHJJSHD3aJoU6TKRurhp
G11KDn/Efn+u3FVj114vkhgbxDWi199xIcO0zrHiCUd3DRkCQQD/07Y90BtnqFey
sFHeITaA57z+H8NszATKaOpKesjCFbNIcMHc0BD3iwySiG89LI68PwWtEnrxa93B
FKx61FP1AkEA9sGkVx108R4Ru5g41fm6h0PB0tJYF2DBgzZViZsDzuOuFWmh2EK5
FHry7ugejbPCofc47yFCxBxPZt/EoHmhWwJBAP8w0xOp6gxmssR+ecvY2aOQhsgA
K1L12LmIG14dSPHB78sNH3UC4EnuHKZ3Dm+5aNCDFUhlrNnyPYyC80J935kCQQDx
FRQZNb9ztCsjHfFGJi1Dk/2X6abDqHbQWZ+Mx/Uah3wn04KapeXpyo3bONHRJFX+
pGntyJNQw1AYdHvJRcyNAkB9Ulyxvubh/xfVIHliKFh1RHkf212W3q3s4WlFRDip
GImb4H6wHQwwyIbwnG5gRaOYMelMjHErB8nHv/+4vd62
----END RSA PRIVATE KEY----
----BEGIN CERTIFICATE----
MIIB+TCCAWICAQEwDQYJKoZIhvcNAQEFBQAwRTELMAkGA1UEBhMCQVUxEzARBqNV
BAqTClNvbWUtU3RhdGUxITAfBqNVBAoTGEludGVybmV0IFdpZGdpdHMqUHR5IEx0
ZDAeFw0wNjA3MTIwNzQ5NTBaFw0wNjA4MTEwNzQ5NTBaMEUxCzAJBgNVBAYTAkFV
MRMwEQYDVQQIEwpTb211LVN0YXR1MSEwHwYDVQQKExhJbnR1cm51dCBXaWRnaXRz
IFB0eSBMdGQwgZ8wDQYJKoZIhvcNAQEBBQADqY0AMIGJAoGBAPaW8/ZuVJOYSX7q
EGv7NUvw+6RgMm3LjI9GWexOpoSY7PWvIK1zd9d8anG9cqhnjZnwGMTiqTsiskMi
g5UmSMCFbUkDFrC3XwmjSeHn1oAkwyK52SXCcyQvHZYurv0xGz9uXp6khAxxatS+
uoTC8v6+nDC3DCqXn/EWKY+qU+0XAqMBAAEwDQYJKoZIhvcNAQEFBQADqYEANK35
BNKwrQuxLcAu406siXOQ5SU1d7zFB4Jp9LWcR2CXxngztmUryTk4zWEWOpQzy/bT
+tjRZLhtGQqUypNOo2wESjlEh1+sLiSBHErIaXpRHaVoWlq+yD2zMtdNuB3IFcId
wGLQkvMV8OigZhDGfCtsvGYFog1xmKk18SZHcR8=
----END CERTIFICATE----
----BEGIN CERTIFICATE-----
MIICoDCCAgmgAwIBAgIBADANBgkqhkiG9w0BAQQFADBFMQswCQYDVQQGEwJBVTET
MBEGA1UECBMKU29tZS1TdGF0ZTEhMB8GA1UEChMYSW50ZXJuZXOqV21kZ210cvB0
dHkqTHRkMB4XDTA2MDcxMjA3NDc1MloXDTA2MDqxMTA3NDc1MlowRTELMAkGA1UE
BhMCQVUxEzARBqNVBAqTClNvbWUtU3RhdGUxITAfBqNVBAoTGEludGVybmV0IFdp
ZGdpdHMgUHR5IEx0ZDCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEAlX/aSX2G
7VRbQgEvDY02ALFrwAUF3Ptc4IBY4ouY9oEEFY0psuQcZUAfEMutnQ8+n5/hwKKG
+F1+Tz7u06q+zkN2sE9EvU5xBVCoIM9GP+mIgOqSqNSdA1zf7c3bnJXxsrGKljJg
B7DCZoaKqn6udhVwSxtJAbB4TfEiI+0jiHsCAwEAAaOBnzCBnDAdBgNVHQ4EFgQU
WN7cMk4ie3WI44p2j4SdSv+xqvgwbQYDVR0jBGYwZIAUWN7cMk4ie3WI44p2j4Sd
Sv+xqvihSaRHMEUxCzAJBqNVBAYTAkFVMRMwEQYDVQQIEwpTb211LVN0YXR1MSEw
HwYDVQQKExhJbnRlcm5ldCBXaWRnaXRzIFB0eSBMdGSCAQAwDAYDVR0TBAUwAwEB
/zANBgkqhkiG9w0BAQQFAAOBgQAsPBL+SyO4tytnjDx874qap21SiBj6ha8DrLLB
TY1W2KHzKdjv1tAweF9z914HNQU594fB7XtDa1VMT3VOZ1EKJk+NOd3Wnbyn6Chy
KjZ4EOcE9BZjuSmvJlGIVtRrN2LrU7cSHFxoY8FUtDYpox11UoiiZmVVuTrxv/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 <code>ipt\_set\_sip\_proxy()</code> has been used to configure a global outbound proxy then this happens automatically. Otherwise the Route header should be included as one of the <code>custom\_headers</code>.

### 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
Create a QoS policy A QoS policy applies a Differentiated Services Code Point (DSCP) value, throttle rate, or both to outbound TCP, UDP, or HTTP response traffic.
Policy name:
sipserv_policy
Specify DSCP Value:
Specify Outbound Throttle Rate:
1 KBps v
Learn more about QoS Policies
OK Cancel Help



Edit an existing QoS policy
Policy Profile Application Name or URL IP Addresses Protocol and Ports
This QoS policy applies to:
Only applications with this executable name:     sipserv.exe
Example: application.exe or %ProgramFiles%\application.exe
Only <u>H</u> TTP 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	
$\bigcirc$ 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:	
Erom any source port	
© F <u>r</u> om this source port number or range:	
Example for a port: 443 Example for a port range: 137:139	
Specify the destination port number:	
I o any destination port	
⑦ To this destination port number or range:	
Learn more about QoS Policies	
OK Cancel Help	



# Contact us

Phone +44 (0)1908 273800 (UK) +1(781) 352 3550 (USA)

Email Info@aculab.com Sales@aculab.com Support@aculab.com

Socials





Certificate number IS 722024 ISO 27001:2013

