aculab

ACULAB.COM

Aculab digital telephony software

Call control API guide

MAN1781 Revision 6.8.7

PROPRIETARY INFORMATION

Document Revision

Rev	Date	By	Detail
6.6.0	09.06.15	mjw	Add fw_extension to the PORT_INFO_PARMS struct.
6.6.1	06.06.16	mjw	Fix copy and paste error in call_maint_port_block()
6.6.2	22.03.17	df/acp	Bug-3036-Removed call_get_system_notification() and call_get_system_notification_wait_object(). Updated call_get_global_notification_wait_object() and updated info on call_event() and event queues.
6.6.3	18/04/17	ebj	Reformatted
6.6.4	31/10/17	ml	Fixed call_feature_details()
6.6.5	11/10/18	df/ebj	Fixed typos in 4.23
6.8.0	25/04/22	dsl	Align with 6.8.0 software release.
6.8.7	13/02/23	dsl	Change title page.

CONTENTS

1 Introduction

This call control guide is designed to help application developers and system integrators understand and use the API provided by Aculab for use with Aculab cards. The call control functions defined in this guide are independent of the operating system used on the host system, enabling applications developed for Aculab cards to be used with several different operating systems.

Aculab support a wide range of international signalling systems and this is reflected in the generic data structures employed by the functions described in this guide. The generic data structures provide a superset of compatible data structures with signalling system specific data structures embedded within the generic data structures. The use of these generic data structures allows applications to be written for use with the range of public and private signalling systems supported by Aculab.

This version of the document is for use with version 6 (V6) drivers.

1.1 Scope

This guide should be used with Aculab call control driver Version 6 or later, for the development of applications that initiate and control outgoing calls and receive and control incoming calls. The **'Aculab switch control API guide'** describes the control and switching of timeslots.

1.2 Using the guide

This guide describes the interfaces for the API functions and the device driver. Each function is described in terms of its calling parameters and the values that the function will return. No particular operating system or signalling system is assumed and function definitions apply equally to all combinations of operating system and signalling protocols. It should be noted that some features and calls are not applicable to all signalling systems.

See Also

- The **'Aculab call, switch and speech driver installation guide',** which describes how to install the software under different operating systems.
- The **'Aculab Switch Control API Guide',** which describes the switching and clock control interface.
- The **'Aculab example code',** which comprises a number of sample applications that illustrate various aspects of the Aculab call and switch API's.

The latest versions of these documents are available from the company web site at http://www.aculab.com

The latest release notes for each signalling system, containing the latest information and describe the use of configuration switches, are also available from the company web site.

2 Compatibility

2.1 Forward Compatibility

Enhancements to the Aculab API are generally made by extending the structures that are passed as parameters to Aculab API calls. To eliminate the 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);
```

Applications, that correctly set this field up, will work correctly against any Aculab library built subsequent to the version with which the application is compiled.

NOTE

Even following this scheme it is not possible for an application built with a particular version of the Aculab libraries to be used against an earlier release of the Aculab drivers.

2.2 Writing portable and forward compatible applications using the Aculab API

Applications should *always* initialise Aculab data structures to 0 using `INIT_ACU_STRUCT()` or an equivalent. This sets unpopulated fields to a safe default and makes forward compatibility much easier. All Aculab functions assume that unused values will be initialised to 0.

Applications must always initialize the *`size`* member of Aculab fields to the size of the structure. Call control APIs can also use `INIT_ACU_CL_STRUCT()`.

Always release resources you allocate. Each `card_id`, `port_id`, `call handle`, etc. represents a finite system resource. Failure to de-allocate them may lead to unexpected behaviour or outright system failure.

Always check the return code of Aculab API calls, even if you are convinced that they cannot fail.

Do not rely on undocumented or platform specific behavior. For example, do not rely on apparently predictable values for call handles, etc. Do not rely on the format of Aculab internal structures. Do not rely on driver filenames, driver directories, the number of drivers, registry entries or values, etc. These are all things that can change arbitrarily between releases.

Do not use undocumented functions. Some Aculab API libraries export functions that are not documented. These are for Aculab internal use and should not be used by customers. There is no guarantee that these functions will continue to exist in future versions or that their semantics will remain the same.

Where possible do not hard code items such as card serial numbers, firmware filenames, firmware configuration switches, etc. It is much easier to edit a configuration file associated with an application in the field than it is to rebuild the application.

Do not make assumptions about what constitutes a valid card serial number.

Always use the appropriate Aculab type – e.g. `ACU_CALL_HANDLE` for a call handle, `ACU_PORT_ID` for a port id. Where there is an Aculab-defined constant for something (e.g. variable lengths, error codes), use it.

Never make IOCTL calls directly to the Aculab driver. The internal API-driver interface is liable to change at any time. If the Aculab API lacks some functionality that would be useful, please contact us at support@aculab.com to discuss your requirements.

Similarly, do not attempt to use the Aculab API in an environment for which it was not designed, for instance, Aculab's API libraries are not designed for kernel mode operation.

3 Declaration types

These types must be used in all applications written using the V6 API instead of native C types. All API calls in the V6 API use these types in their declarations.

Some examples of their use can be found in this document, and in some instances, in the V6 resource management API guide.

3.1 ACU_RESOURCE_ID – Generic Resource ID

This type is used when any of the Aculab resource ID types can be passed into an API call.

3.2 ACU_CALL_HANDLE - Call handle

This type is used to represent a single call. Values used for this are returned by functions such as `call_openout()` and `call_openin()`. Many Aculab API calls take these values as parameters. The call handle is not meaningful in itself – no attempt should be made to decompose it. Call handles are allocated by the system – you cannot specify your own handle (see `call_openout()` for information on how to associate your own data with a call). Additionally, any regularity or commonality in the values returned should not be relied upon.

Call handles are associated with a particular process and cannot be passed between different processes.

3.3 ACU_CARD_ID – Card id

This type is used to represent an instance of an Aculab card. Values used for this are returned by `acu_open_card()`. A number of Aculab API calls take these values as parameters. The card id is not meaningful in itself.

Card ids are associated with a particular process and cannot be passed between different processes.

3.4 ACU_PORT_ID – Port id

This type is used to represent an instance of a port on an Aculab card. Values used for this are returned by `call_open_port()`. A number of Aculab API calls take these values as parameters. The port id is not meaningful in itself.

Port ids are associated with a particular process and cannot be passed between different processes.

3.5 ACU_WAIT_OBJECT – Wait Object

This type is used to allow platform specific wait functions to be used with the Aculab API. It represents a platform-specific wait object (e.g. a `HANDLE` on Windows, and a file descriptor on Unix-like operating systems). It can be used in functions such as `poll()` or `WaitForMultipleObjects()`.

3.6 ACU_ACT – Application token

This type is used to allow an application to associate data with a call. This is passed in as an additional parameter to `call_openin()` and `call_openout()`. The value is returned by `call_event()` and `call_details()`.

3.7 ACU_ERR – Error code

This is the return type of Aculab API calls.

3.8 ACU_EVENT_QUEUE – Event queue

This type is a reference to an event queue created with `acu_alloc_event_queue()`. This type is used to allow an application to associate a particular resource with a queue.

4 Call control functions

4.1 System configuration

These functions configure the card or provide information about the system:

API	Description
`call_card_dsp_config()`	This API call returns information about the DSPs present on the specified card.
`call_driver_version()`	This function returns the version of the driver operating for a port. As different ports on the same card can be controlled by different drivers, the driver version is returned on a per-port basis.
`call_type()`	Used to return a system wide signalling system reference value for a given network port number. This function is useful for applications that support multiple signalling systems
`call_line()`	Used to return a reference value describing the type of trunk supported by the card. This function is useful if support of multiple interface types is being considered for the application

4.2 Line control

These functions control the 'Physical' or 'Layer 1'connection:

API	Description
`call_watchdog()`	Used to start, stop and refresh the layer 1 watchdog. The aim of this feature is to signal to the network that the Aculab card can no longer take calls. Reasons for this could be an application or operating system failure.
`call_send_alarm()`	Used to send a layer 1 alarm to the network. This may be useful in systems where receipt of an alarm on one trunk may require transmission of that alarm on another trunk. Care should be taken when using this function as transmission of some alarms may cause the network receiving the alarm to cease operation.
`call_port_info()`	Get information about a port
`call_set_port_name()`	This function is used to set the name of a port. The name of a port can be used to provide a more descriptive identifier to each port in the system.
`call_get_card_info()`	Return information about a card.
`call_get_fmw_dl_parms()`	This diagnostic function returns the parameters that were specified when call control firmware was downloaded.
`port_init()`	This function is used to initialise a port before calls are made on it. This function calls `idle_net_ts()` for each of the timeslots on a port.

4.3 Basic call control

This section of the guide covers the library functions available for basic call control. These allow calls to be made or received, details of the call to be received and the clearing of calls:

API	Description
`call_open_port()`	This function is used to open a port for use. The *`port_id`* returned by this function is used as a parameter in many API calls.
`call_close_port()`	This function is used to close a port that was previously opened with `call_open_port()`.
`call_openout()`	Allows an application to initiate an outgoing call. The function registers the outgoing call requirement with the device driver, which, if satisfied with the calling parameters, will return a unique call identifier, the handle.
`call_openin()`	Allows an application to initiate a wait for an incoming call. The function registers the incoming call requirement with the device driver. If the driver is satisfied with the calling parameters, it will return a unique call identifier, the call handle for that call.
`call_send_overlap()`	Used to send the destination address of an outgoing call one or more digits at a time. The function may also be used any time that a valid outgoing call handle is available and the state of the handle is `EV_WAIT_FOR_OUTGOING`. The outgoing call handle would have been obtained from the `call_openout()` function.
`call_event()`	Used to return an event that may have occurred on any incoming or outgoing call. The device driver issues an event when a change of state occurs. The function may be used any time.
`call_details()`	Used to gather the details of the current call, either incoming or outgoing, connected through the device driver.
`call_incoming_ringing()`	Used to send the ringing message to the network causing the caller to hear the ring tone. This function is used after an incoming call has been detected but before the call has been accepted. Use of the function will stop further call details, such as DDI digits, from being received.
`call_accept()`	Used to accept the call after an incoming call has been indicated. This function would typically be used once the application had determined that the DDI digits received by the `call_details()` function are correct and can be supported.
`call_getcause()`	Used to return the clearing cause when an incoming or outgoing call clears. The returned clearing cause will only be valid when the call state is either `EV_IDLE` or `EV_REMOTE_DISCONNECT`.

(Basic call control continued)

API	Description
`call_disconnect()`	Used to disconnect an incoming or outgoing call currently routed through the driver. If the `call_disconnect()` function is successful, the driver will start the disconnect procedure and will return immediately to the calling process. When the call has been disconnected, the state will be `EV_IDLE`. The `call_release()` function must be used to give back the handle to the driver.
`call_release()`	Used to relinquish ownership of a call handle in response to call termination (`CS_IDLE` or `EV_IDLE`) or any error condition that may cause the application to abandon the call. The handle may no longer be used by the application.

4.4 Advanced call control

These functions complement the 'Basic Call' functions and provide extra functionality for the user including the use of 'Supplementary Services':

API	Description
`call_feature_openout()`	It is possible to transmit feature information in a call setup message. An enhanced version of `call_openout()` is needed to include extra parameters.
`call_feature_enquiry()`	During the process of call transfer, this function allows an application to make an enquiry call, i.e. an outgoing call to a third party, with feature information
`call_feature_details()`	Supplementary service information may arrive at different stages during the lifetime of a call. An indication of the availability of this information is found in the `feature_information` field, after a call to `call_details()`. To retrieve the information a second function, `call_feature_details()` should be used.
`call_feature_send()`	`call_feature_send()` is used to transmit feature information at different stages during the lifetime of a call.
`call_setup_ack()`	Used on an incoming call to send a Q.931 SETUP_ACKNOWLEDGE message to the calling party.
`call_proceeding()`	Used on an incoming call to send a message to the calling party to indicate that sufficient information has been obtained to proceed with the call.
`call_progress()`	Used to send call progress information to the network.
`call_get_originating_addr()`	Used to obtain the originating address of an incoming call. It is primarily intended for use in some Channel Associated Signalling (CAS) systems where the application must explicitly request the originating address from the network

(Advanced call control continued)

API	Description
call_answercode()	Some protocols allow an incoming call to be answered with information about how the call is to be handled during the EV_CALL_CONNECTED state. This function is used to pass the answer code to the driver for use during call connection.
call_get_charge	Used to obtain information regarding the cost of a call. The function may be used any time that a valid call handle is available, however, the call charge information may not be valid until the call has cleared and the call has gone to the EV_IDLE state.
call_put_charge	Used to send call charging information on the network and may be used any time that a valid call handle is available and the call is in the EV_CALL_CONNECTED state.
call_notify()	Used on a call to send a message to the network to indicate an appropriate call related event during the active state of a call (such as user suspended).
call_send_keypad_info()	Used to send keypad information during a call. This function is only supported in Q.931
call_send_connectionless()	Some supplementary services require the use of a connectionless network service to transmit FACILITY messages. call_send_connectionless() allows these messages to be transmitted.
call_get_connectionless()	Some supplementary services require the use of a connectionless network service to transmit FACILITY messages. call_get_connectionless() will retrieve the latest message of this kind
call_enable_connectionless()	Some protocols require that actions be taken for messages that are not understood.
call_maint_port_block()/ call_maint_port_unblock()	The block function is used to block a group of timeslots within a port, preventing these timeslots from being used for subsequent calls. Conversely, unblock is used on a blocked port to unblock a group of timeslots for that port, bringing the timeslots back into service.
call_maint_ts_block()/ call_maint_ts_unblock()	The block function is used to block (take out of service) a specific timeslot, preventing it from being used to set up a call. Conversely, unblock is used on a blocked timeslot to unblock the timeslot and bring it back into service.
call_maint_port_reset()	Used to reset the status of a port. This will clear any calls in progress on the port. (Currently SS7/ISUP support only)
call_maint_port_status	Used to protocol specific status information.
call_set_handle_event_queue()	This function is used to associate a call handle with an event queue. All call events that occur for the specified call handle will be notified via this event queue

(Advanced call control continued)

API	Description
`call_get_handle_event_wait_object()`	This function is used to get a wait object that is associated with a specific call handle. This wait object will be signalled while there are call events pending for that call handle. The wait object returned by this function can be used with operating system specific wait functions such as `WaitForMultipleObjects()` or `poll()`.
`call_set_handle_app_context_token()`	This function is used to associate application-defined data with a call handle. This data is returned as the *context* field by `acu_get_event_from_queue()`
`call_get_handle_app_context_token()`	This function is used to retrieve application defined data that was associated with a call handle by an earlier call to `call_openin()`, `call_openout()`, or `call_set_handle_app_context_token()`.
`call_set_port_default_handle_event_queue()`	This function is used to set the default event queue for calls made on a particular port. This allows an application to easily group calls by port.
`call_set_port_app_context_token()`	This function is used to associated application defined data with a port. This data is returned as the *context* field by `acu_get_event_from_queue()`.
`call_get_port_app_context_token()`	Retrieve application-defined data that is associated with a port. The data can be set using `call_set_port_app_context_token()`.
`call_set_port_notification_queue()`	Associate a port with a queue. All port notification events for this port will be notified via this queue.
`call_get_port_notification()`	Retrieve the next pending notification event for the port.
`call_get_port_notification_wait_object()`	Get the wait object that is signalled when there is a pending notification event for the port.
`call_get_global_event_wait_object()`	Get the wait object that is signalled when there is a pending call event on any call associated with the global call event queue (see `call_event()`).
`call_set_global_notification_queue()`	Associate global notifications with a queue. All global notification events will be notified via this queue.
`call_get_global_notification()`	Retrieve the next pending global notification event.

(Advanced call control continued)

API	Description
call_get_global_notification_wait_object()	Get the wait object that is signalled when there is a pending global notification event.
call_open_iptel_port()	There are times when an application may wish to have a greater level of control over the media resources on an IP Telephony call than the basic API offers

These functions are used to implement call transfer:

API	Description
call_hold()	An incoming or outgoing call can be put on hold by use of the call_hold() function. To allow enhancements to the driver API, this call will in turn call xcall_hold(), which may be called directly.
call_reconnect()	A call that is in the held state can be reconnected by use of the call_reconnect() function. To allow enhancements to the driver API, this call will in turn call xcall_reconnect(), which may be called directly. This call will cause the call to return to the connected state. It is possible for a switch to reject a reconnect message sent by the application. If this happens the call will move to the state CS_RECONNECT_REJECT.
call_enquiry()	During the process of call transfer, this function allows an application to make an enquiry call, that is, an outgoing call to a third party.
call_transfer()	After a successful enquiry call, the calls may be transferred by use of the call_transfer() function call. The enquiry call is deemed successful if the state of the enquiry call has reached CS_CONNECTED or CS_OUTGOING_RINGING (for 'blind' transfer).

4.5 call_card_dsp_config()

This API call returns information about the DSPs present on the specified card.

This API call returns information about DSPs associated with call control activities. A card may carry additional DSPs for other purposes. Applications must use different API calls to find out about those.

This API call is primarily intended to allow applications to determine whether a PMX-based card can support tone-based CAS signalling or SS7 signalling.

Synopsis

```
ACU_ERR call_card_dsp_config(CALL_CARD_DSP_CONFIG_PARMS* config_parms);

#define MAX_CARD_DSP_COUNT 5

typedef struct tCALL_DSP_INFO
{
   ACU_UINT       dsp_type;
   ACU_CHAR       status[MAX_HW_DESC];
   ACU_CHAR       dsp_desc[MAX_HW_DESC];
} CALL_DSP_INFO;

typedef struct tCALL_CARD_DSP_CONFIG_PARMS
{
   ACU_ULONG           size;                              /* IN */
   ACU_CARD_ID         card_id;                           /* IN */
   ACU_UINT            dsp_count;                         /* OUT */
   CALL_DSP_INFO       dsp_info[MAX_CARD_DSP_COUNT];      /* OUT */
   ACU_UCHAR           port_dsp_map[MAXPPC];              /* OUT */
   ACU_UCHAR           port_res_map[MAXPPC];              /* OUT */
} CALL_CARD_DSP_CONFIG_PARMS;
```

Input parameters

`call_card_dsp_config()` takes a pointer, *`config_parms`*, to a structure, `CALL_CARD_DSP_CONFIG_PARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`Size`
This is the size of the structure (use `INIT_ACU_CL_STRUCT` to initialise).

`card_id`
This is the card id of the card to query for DSP information.

Return values

`dsp_count`
This is the number of DSP positions available on the card.

`dsp_info`
This array contains information about each of the DSP positions.

`dsp_type`
This is the type of DSP fitted. This is one of the following values:

Constant	Description
`ACU_DSP_TYPE_PMX_HIGH_CAP`	DSP suitable for SS7 signalling or 8 ports of CAS tone generation/detection
`ACU_DSP_TYPE_PMX_LOW_CAP`	DSP suitable for 8 ports of CAS tone generation/detection

Status
This is a free-format text field describing the status of the DSP. For PMX-based cards this field will contain information about which firmware is downloaded (if any).

dsp_desc
This is a textual description of the DSP.

port_dsp_map
This is for diagnostic purposes only

port_res_map
This is for diagnostic purposes only

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

4.6 call_driver_version()

This function returns the version of the driver operating for a port. As different ports on the same card can be controlled by different drivers, the driver version is returned on a per-port basis.

Synopsis

```
call_driver_version(CALL_DRIVER_VERSION_PARMS* version_parms);

typedef struct tCALL_DRIVER_VERSION_PARMS
{
  ACU_ULONG      size;                          /* IN */
  ACU_PORT_ID    port_id;                       /* IN */
  ACU_UINT       ver_maj;                       /* OUT */
  ACU_UINT       ver_min;                       /* OUT */
  ACU_UINT       ver_rev;                       /* OUT */
  ACU_CHAR       ver_desc[MAX_VER_DESC +1];      /* OUT */
} CALL_DRIVER_VERSION_PARMS;
```

This function returns the version of the driver operating for a port. The driver version is returned on a per-port basis as different ports on the same card can be controlled by different drivers.

Input Parameters

size:
The size of the `CALL_DRIVER_VERSION_PARMS` struct.

port_id:
The port id of the port to be queried

Return values

ver_maj:
The major component of the driver version

ver_min:
The minor component of the driver version

ver_rev:
The revision component of the driver version

ver_desc:
An ASCII string containing the driver version plus any additional version information (such as beta status). e.g. "v6.3.2B21".

On successful completion a value of zero is returned. Otherwise, `ERR_NET` will be returned if an invalid port is specified instead.

4.7 call_type() - Get signalling system type

This function may be used to return a system wide signalling system reference value for a given network port number. This function will be useful for applications that support multiple signalling systems.

Synopsis

```
ACU_ERR call_type(ACU_PORT_ID portnum);
```

Input parameters

The input parameter *portnum* identifies the network port number for which the reference value is required and will have a valid port id as returned from `call_open_port()`.

Return values

On successful completion the function will return a signalling system reference value. The value zero will be returned if the signalling system name is not known; otherwise a negative value will be returned indicating the type of error.

The reference value returned will be one of the set defined in the header file under `Signalling Protocol Identifiers`.

4.8 call_line() - Get line type

This function may be used to return a reference value describing the type of trunk supported by the card. This function will be useful if support of multiple interface types is being considered for the application.

Synopsis

```
ACU_ERR call_line (ACU_PORT_ID portnum);     /* IN */
```

Input parameters

The input parameter *portnum* identifies the network port number for which the line type is required and will have a valid port id as returned from `call_open_port()`.

Return values

On successful completion, the function will return a line reference value. The value zero will be returned if the line reference is not known; otherwise a negative value will be returned indicating the type of error.

The reference value returned will be one of the set defined in the header file under `Line Interface Types`:

`L_E1`	for an E1 (PRI) ISDN line
`L_T1_CAS`	for a T1 CAS line
`L_T1_ISDN`	for a T1 (PRI) ISDN line
`L_BASIC_RATE`	for a basic rate (BRI) ISDN line
`L_PSN`	for packet switched network (IP Telephony H.323 or SIP)

4.9 call_watchdog() - Watchdog

The API call `call_watchdog()` is used to start, stop and refresh the layer 1 watchdog. Once set, the watchdog must be refreshed before the time specified by the user has elapsed. If the timer in the firmware expires before a refresh signal has been received then the firmware will set the user specified Layer 1 alarm on the link. The aim of this feature is to signal to the network that the Aculab card can no longer take calls. Reasons for this could be an application or operating system failure.

NOTE

Not supported for IP Telephony or ISUP

Library function

Synopsis

```
ACU_ERR call_watchdog(WATCHDOG_XPARMS *watchp);

typedef struct watchdog_xparms
{
  ACU_ULONG           size;     /* IN */
  ACU_PORT_ID         net;      /* IN */
  ACU_INT             alarm;    /* IN */
  ACU_LONG            timeout;  /* IN */
} WATCHDOG_XPARMS;
```

Input parameters

`call_watchdog()` takes a pointer, *`watchp`*, to a structure, `WATCHDOG_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`net`
must contain the *`port_id`* on the Aculab card on which call watchdog is to be run, as returned from `call_open_port()`.

`alarm`
must contain the alarm code to be sent to the network upon the timeout expiring and will be one of the standard set of alarms codes described below.

`ALARM_NONE`	will clear an existing alarm
`ALARM_AIS`	will cause AIS to be sent
`ALARM_RRA`	will cause Remote Alarm
`ALARM_CML`	will cause CAS Multi Frame Alarm

`timeout`
must contain the time, in milliseconds, for the watchdog to wait before setting the layer 1 alarm. A zero value for the timeout will disable the watchdog feature. If `call_watchdog()` is not called again before the time out period then the firmware will set the alarm with the alarm value stated in the previous `call_watchdog()`.

Return values

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

4.10 call_send_alarm() - Sending alarms to the network

This function may be used to send a layer 1 alarm to the network. This may be useful in systems where receipt of an alarm on one trunk may require transmission of that alarm on another trunk. Care should be taken when using this function as transmission of some alarms may cause the network receiving the alarm to cease operation.

The alarm condition will persist as until changed by subsequent use of the `call_send_alarm()` function.

NOTE

Not supported for IP Telephony.

Synopsis

```
ACU_ERR call_send_alarm(ALARM_XPARMS *alarmp);

typedef struct alarm_xparms
{
  ACU_ULONG           size;     /* IN */
  ACU_PORT_ID         net;      /* IN */
  ACU_INT             alarm;    /* IN */
} ALARM_XPARMS;
```

Input parameters

`call_send_alarm()` takes a pointer, `alarmp`, to a structure, `ALARM_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`net`
must contain the *`port_id`* on the Aculab card on which the alarm is to be sent. As returned from `call_open_port()`.

`alarm`
must contain the alarm code to be sent to the network and will be one of the standard set of alarms codes described below.

`ALARM_NONE`	will clear an existing alarm
`ALARM_AIS`	will cause AIS to be sent
`ALARM_RRA`	will cause Remote Alarm
`ALARM_CML`	will cause CAS Multi Frame Alarm

The `ALARM_xxx` definitions have the same value as the `LSTAT_xxx` definitions allowing the layer 1 status of one port to be sent unchanged as an alarm on another port.

Return values

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

4.11 call_port_info() - get information about a port

This function returns information about a port.

Synopsis

```
ACU_ERR call_port_info(PORT_INFO_PARMS* port_info_xparms);

typedef struct _PORT_INFO_PARMS
{
  ACU_ULONG           size;                       /* IN */
  ACU_PORT_ID         port_id;                    /* IN */
  ACU_ULONG           valid_vector;               /* OUT */
  ACU_UINT            connection_type;            /* OUT */
  ACU_UINT            channel_allocation;         /* OUT */
  ACU_UINT            channel_count;              /* OUT */
  ACU_UINT            port_type;                  /* OUT */
  ACU_UINT            lim_type;                   /* OUT */
  ACU_UINT            physical_index;             /* OUT */
  ACU_CHAR            name[MAXNAME];              /* OUT */
  ACU_CHAR            sig_sys[MAXSIGSYS];         /* OUT */
  ACU_CHAR            hw_ver[MAXHWVERSION];       /* OUT */
  ACU_CHAR            hw_desc[MAX_HW_DESC];       /* OUT */
  ACU_CHAR            fw_desc[MAX_FW_DESC];       /* OUT */
  ACU_CHAR            fw_extension[MAX_FW_EXTN];  /* OUT */
} PORT_INFO_PARMS;
```

Input parameters

The `call_port_info()` function takes a pointer, *`port_info_xparms`*, to a structure, `PORT_INFO_PARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`port_id`
The input parameter *`port_id`* identifies the network port for which information is to be obtained.

Return values

NOTE

Before firmware is downloaded to a port, this function will return default information for the port.

`valid_vector` (Not applicable for IP Telephony)

The 32-bit vector returned in *`valid_vector`* describes the available timeslots on the network stream. Bit 0 refers to timeslot 0 and bit 31 refers to timeslot 31. A bit set will indicate that the timeslot is valid. Thus a vector:

```
31                                          0
1111 1111 1111 1110 1111 1111 1111 1110
```

Indicates that timeslots 0 and 16 are barred and may not be used and is typical of E1. Other vectors will be returned for other line types.

For example ISDN on T1 will return:

```
31                                          0
0000 0000 0111 1111 1111 1111 1111 1111
```

If a barred timeslot is specified during call control an error will be returned by that function (see error codes).

NOTE

Before firmware is downloaded, the valid_vector will show that no timeslots are valid. For SS7, valid_vector is not valid until both the firmware is downloaded and the SS7 stack has been started.

connection_type

The *connection_type* field will contain a mask consisting of a combination of the following values, depending on the type of port:

#define	Description
ACU_CIRCUIT_SWITCHED	This port is circuit switched
ACU_PACKET_SWITCHED	This port is packet switched

channel_allocation

The *channel_allocation* field describes how channels are allocated on this port.

This information can be useful for deciding how many calls to make on it. The value in this field will be made up of a combination of the following values, depending on the type of the port:

#define	Description
ACU_CHANNEL_FIXED	Timeslots on this port are fixed (e.g. ISDN protocols)
ACU_CHANNEL_DYNAMIC	Timeslots on this port are dynamically allocated
ACU_CHANNEL_DYNAMIC_UNLIMITED	This port can provide an unlimited quantity of channels

On a port that has channels dynamically allocated, it is generally not possible to determine which timeslot a call will use until the call has been made. For ports where the timeslots are fixed, it is possible to specify the timeslot that a call should take. A port of type ACU_CHANNEL_DYNAMIC_UNLIMITED can provide an unlimited number of channels to an application.

channel_count

The *channel_count* field contains the maximum number of channels available on the port. For protocols that have a *channel_allocation* of ACU_CHANNEL_FIXED this count is absolute. Protocols where the *channel_allocation* is ACU_CHANNEL_DYNAMIC may support a varying number of channels depending on other factors such as load, options specified for the port, and options specified for each call. The value in this field may be ignored for ports where *channel_allocation* is ACU_CHANNEL_DYNAMIC_UNLIMITED, as they are capable of providing of an unlimited number of channels.

NOTE

The reported channel count for SIP ports opened using sip_open-port() may indicate lower than expected numbers if the VoIP board is still booting when call_port_info() is called. Once the VoIP firmware us fully initialised, the count will be fixed at a value equal to the total capacity of the installed hardware.

port_type

The *port_type* field is a combination of the following values, describing the capabilities of the port. Making use of a particular capability may require a particular type of firmware to be downloaded.

#define	Meaning
ACU_PORT_CAP_E1_ISDN	Port is capable of E1 ISDN protocols (e.g. ETS 300, DASS2)
ACU_PORT_CAP_T1_ISDN	Port is capable of T1 ISDN protocols (e.g. NI2, AT&T)
ACU_PORT_CAP_E1_CAS	Port is capable of E1 CAS protocols (e.g. E1 Lineside)
ACU_PORT_CAP_T1_CAS	Port is capable of T1 CAS protocols (e.g. T1 Robbed Bit)
ACU_PORT_CAP_E1_ISDN_MONITOR	Port is capable of monitoring E1 ISDN calls
ACU_PORT_CAP_T1_ISDN_MONITOR	Port is capable of monitoring T1 ISDN calls
ACU_PORT_CAP_IP	Port is capable of running IP Telephony
ACU_PORT_CAP_IP_OFFLOAD	Port is capable of running H.323 IP Telephony, with call control, on the board.
ACU_PORT_CAP_E1_ISUP	Port is capable of running E1 ISUP.
ACU_PORT_CAP_T1_ISUP	Port is capable of running T1 ISUP
ACU_PORT_CAP_SS7	Port is capable of running SS7

lim_type (not applicable for IP telephony)

The *lim_type* field describes the actual type of LIM module that the port is on. Values for this field are defined in the header file under PM module types.

NOTE

Aculab advise against coding checks for specific LIM types as it may limit forward compatibility. This field should be used for information only.

physical_index (not applicable for IP telephony)

The physical_index field is the physical index of the port on its parent card.

name

The name field holds the name of the port. The name can be set using call_set_port_ name().

sig_sys

The driver will return a null terminated ASCII string containing the signalling system type that is supported by the device driver in the character array *sig_sys*. This string will contain one of the following:

- E1 - ISDN signaling systems
  ETS300 ETSNET FETX150 FETXNET
  DASS2 DASSNE DPNSS QSIG
- E1 - CAS signaling systems
  R2B2P CAS BTCU BTCN PTVU PTVN PD1D
  PD1U PD1N R2L P8 EM BEZEQ
- E1 - CAS tone signaling systems - requires DSP
  R2T R2T1 ALSU ALSN BELGU BELGN EFRAT
  EEMA PD1 PD1DD PD1UD PD1ND BTMC OTE2
  FMFS SMFS I701 SS5
- T1 - ISDN signaling systems
  NI2 NI2NET INS_T1 INT1NET ATT1 ATT1NET ETST1U ETST1N
  DMS1 DMS1NET
- T1 - CAS tone signaling systems - requires DSP
  F12 T1RB
- SS7
  ISUP
- Passive Monitor
  E1 - MONE
  T1 – MONT
- IP Telephony
  H323 SIP

NOTE

When firmware has not been downloaded to a port, the string returned in sig_sys is "NO F/W".

hw_ver (not applicable for IP telephony)

The *hw_ver* field is a null terminated string containing the hardware version of the LIM that the port is on.

hw_desc (not applicable for IP telephony)

A string that describes the hardware of a port. It is provided for informational purposes only and as such provides a textual description of the *port_type* field.

fw_desc (not applicable for IP telephony)

The name of the firmware file currently running on the port, provided for informational purposes only. As firmware filenames vary between different types of hardware and new firmware variants and hardware types might be introduced in the future, it is not recommended that applications be written to depend on specific values appearing in this field.

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

fw_extension (not applicable for IP telephony)

Introduced in v6.6.4.

The extension of the call control firmware file that is expected by the driver. This can be either "pmx" or "pxi".

4.12 call_set_port_name() - set the name of a port

This function is used to set the name of a port. The name of a port can be used to provide a more descriptive identifier to each port in the system. Some Aculab configuration tools also display this name. It can be retrieved using `call_port_info()`.

Synopsis

```
ACU_ERR call_set_port_name(CALL_SET_PORT_NAME_PARMS* name_parms);

typedef struct _CALL_SET_PORT_NAME_PARMS
{
  ACU_ULONG           size;                 /* IN */
  ACU_PORT_ID         port_id;              /* IN */
  ACU_CHAR            name[MAXNAME];        /* IN */
} CALL_SET_PORT_NAME_PARMS;
```

Input Parameters

The `call_set_port_name()` function takes a pointer, *`name_parms`*, to a structure, `CALL_SET_PORT_NAME_PARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`port_id`

`port_id` is the id of the port. This id is obtained from a call to `call_open_port()`.

`name`

`name` is the new name for the port. It's a null terminated string and must be no longer than `MAXNAME` characters including the `NULL` character.

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

Setting the name:

```
ACU_CHAR* port_name = "TELCO A";
SET_PORT_NAME_PARMS set_name_parms;
ACU_PORT_ID; /* port previously opened with call_open_port() */
ACU_INT error = 0;

INIT_ACU_STRUCT(&set_name_parms);
set_name_parms.port_id = port_id;
strncpy(set_name_parms.name, port_name, MAXNAME);

error = call_set_port_name(port_id, port_name);
```

4.13 call_get_card_info() - return information about a card

This function returns information about a card.

Synopsis

```
ACU_ERR call_get_card_info(CARD_INFO_PARMS* info_parms);

typedef struct _CARD_INFO_PARMS
{
  ACU_ULONG      size;                        /* IN */
  ACU_CARD_ID    card_id;                     /* IN */
  ACU_UINT       ports;                       /* OUT */
  ACU_UINT       card_type;                   /* OUT */
  ACU_ULONG      irq_ticks;                   /* OUT */
  ACU_ULONG      clock_ticks;                 /* OUT */
  ACU_CHAR       serial_no[MAXSERIALNO];      /* OUT */
  ACU_CHAR       hw_version[MAXHWVERSION];    /* OUT */
  ACU_ULONG      interrupts_seen;             /* OUT */
  ACU_ULONG      interrupts_claimed;          /* OUT */
} CARD_INFO_PARMS;
```

Input parameters

The `call_get_card_info()` function takes a pointer, *`info_parms`*, to a structure, `CARD_INFO_ PARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`card_id`
The *`card_id`* field must contain the appropriate id for the call driver you want to query. This id must be opened for call control with an earlier call to `acu_open_call()`.

Return values

`ports`
The *`ports`* field will contain the number of network ports on the card.

`card_type`
The *`card_type`* field returns a value denoting the exact type of Aculab board:

#define	Description
ACU_PROSODY_X_CARD	A Prosody X PCI, cPCI and PCIe cards
ACU_PROSODY_S_V3_CARD	A ProsodyS virtual card
ACU_VIRTUAL_BEARER_CARD	A Virtual bearer card

`irq_ticks` (not applicable for IP telephony)

This field contains the number of times the call driver for this card has processed its interrupt queue in response to one or more interrupts.

`clock_ticks`
This field is not used in the V6 call driver.

`serial_no`
The *`serial_no`* field returns an alphanumeric character string that contains the serial number of the Aculab Prosody PCI/E1T1 or Compact PCI Prosody/E1T1 cards installed.

`hw_version`
The *`hw_version`* field contains the revision of the card

`interrupts_seen` (not applicable for IP telephony)

This field contains the number of times the call driver for this card has received an interrupt from the operating system.

`interrupts_claimed` (not applicable for IP telephony)

This field contains the number of times that the call driver for this card has received an interrupt for this card.

NOTE

Aculab does not recommend constantly polling this function.

NOTE

It is normal for the number of interrupts seen to differ from the number of interrupts claimed.

NOTE

call_port_info() can be used to obtain the LIM version.

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

4.14 call_get_fmw_dl_parms – return downloaded call firmware parameters

This function returns the parameters that were specified when call control firmware was downloaded.

NOTE

This function only returns firmware parameters that were specified using call_restart_fmw().

NOTE

This function is provided for diagnostic purposes only.

Synopsis

```
ACU_ERR call_get_fmw_dl_parms(CALL_GET_FMW_DL_PARMS* fmw_dl_parms);

typedef struct tCALL_GET_FMW_DL_PARMS
{
  ACU_UINT       size;                       /* IN */
  ACU_PORT_ID    port_id;                    /* IN */
  ACU_CHAR       parms[CL_MAX_FMW_PARMS];    /* OUT */
} CALL_GET_FMW_DL_PARMS;
```

Input parameters

The `call_get_fmw_dl_parms()` function takes a pointer, *`fmw_dl_parms`*, to a structure, `CALL_GET_FMW_DL_PARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`Size`
This is the size of the structure (initialise using `INIT_ACU_CL_STRUCT`)

`port_id`
This is the port id of the port to query.

`Parms`
On return, this will hold the parameters that were specified.

Return values

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

4.15 port_init() - port initialisation

This function is used to initialise a port before calls are made on it. This function calls `idle_net_ts()` for each of the timeslots on a port.

Synopsis

```
ACU_ERR port_init(ACU_PORT_ID portnum);      /* IN */
```

NOTE

Not applicable for IP telephony

NOTE

This function makes use of the switch API and, as such, will only function correctly if the application has already opened the card for switch API use using the acu_open_switch() function.

Input Parameters

portnum

portnum must be a valid *port_id* returned from an earlier call to `call_open_port()`.

Return Values

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

Basic call control

4.16 call_open_port() - open a port

This function is used to open a port for use. The *port_id* returned by this function is used as a parameter in many API calls.

NOTE

An application must keep track of the ports it has opened.

Synopsis

```
ACU_ERR call_open_port(OPEN_PORT_PARMS* port_parms);
typedef struct tOPEN_PORT_PARMS
{
  ACU_ULONG           size;              /* IN */
  ACU_CARD_ID         card_id;           /* IN */
  ACU_INT             port_ix;           /* IN */
  ACU_PORT_ID         port_id;           /* OUT */
  ACU_EVENT_QUEUE     queue_id;          /* IN */
} OPEN_PORT_PARMS;
```

Input Parameters

This `call_open_port()` function takes a pointer *port_parms*, to a structure,`OPEN_PORT_PARMS`. The structure must be initialised before invoking the function, (see section 2.2).

card_id

The *card_id* field must be set to the id of the card that owns the port to open.

port_ix

The *port_ix* field must be set to the number of the physical port to open. This is the 0 based index of the port. You can determine how many ports a card has using the `call_get_card_info()` API call.

Return Values

port_id

The *port_id* 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.

Error Codes

#define	Description
ERR_NET	Invalid port index specified
ERR_INVALID_CARD	Invalid card_id specified

Example Usage

```
OUT_XPARMS call_parms;
OPEN_PORT_PARMS open_port;
CLOSE_PORT_PARMS close_port;
ACU_CARD_ID card_id;
ACU_INT error = 0;
ACU_CHAR* serial_no = "123456";

INIT_ACU_STRUCT(&open_card);
strcpy(open_card.serial_no, serial_no);

card_id = open_card.card_id;

/* open the first port on this card */
INIT_ACU_STRUCT(&open_port);
open_port.card_id = card_id;
open_port.port_index = 0;

error = call_open_port(&open_port);
if (error == 0)
{
   /* Use the port */
   INIT_ACU_STRUCT(&call_parms);
   call_parms.net = open_port.port_id;
   call_parms.ts = -1;

   /*
    * other call set up here
    */

   error = call_openout(&call_parms);
   if (error != 0)
   {
        /* handle call here */
   }

   INIT_ACU_STRUCT(&close_port);
   close_port.port_id = open_port.port_id;

   error = call_close_port(&close_port);
}
```

4.17 call_close_port() - close a port

This function is used to close a port that was previously opened with `call_open_port()`.

Synopsis

```
ACU_ERR call_close_port(CLOSE_PORT_PARMS* port_parms);
typedef struct tCLOSE_PORT_PARMS
{
  ACU_ULONG           size;          /* IN */
  ACU_PORT_ID         port_id;       /* IN */
} CLOSE_PORT_PARMS;
```

NOTE

The order that you release the resources in is important. You cannot close a port before closing all of the call handles on that port.

Input Parameters

This `call_close_port()` function takes a pointer *port_parms*, to a structure,`CLOSE_PORT_PARMS`. The structure must be initialised before invoking the function, (see section 2.2).

port_id

This function takes the `port_id` of an open port as its only parameter.

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

See the example code for `call_open_port()` shown above.

4.18 call_openout() - Open for outgoing call

This function allows an application to initiate an outgoing call. The function registers the outgoing call requirement with the device driver, which if satisfied with the calling parameters, will return a unique call identifier, the *handle*.

Synopsis

```
ACU_ERR call_openout(OUT_XPARMS *outdetailsp);

typedef struct out_xparms
{
  ACU_ULONG           size;                       /* IN */
  ACU_CALL_HANDLE     handle;                     /* OUT */
  ACU_PORT_ID         net;                        /* IN */
  ACU_INT             ts;                         /* IN */
  ACU_INT             cnf;                        /* IN */
  ACU_INT             sending_complete;           /* IN */
  char                destination_addr[MAXADDR];  /* IN */
  char                originating_addr[MAXADDR];  /* IN */
  ACU_ACT             app_context_token;          /* IN */
  ACU_EVENT_QUEUE     queue_id;                   /* IN */
  union uniqueu       unique_xparms;              /* IN */
} OUT_XPARMS;

  union uniqueu
  {
    /* see protocol specific structures in section 8 */
  } unique_xparms;
```

Input parameters

The `call_openout()` function takes a pointer, *outdetailsp*, to a structure, `OUT_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

net

Specifies the *port_id* on the Aculab card on which the call is to be made, as returned from `call_open_port()`.

ts

NOTE

For IP Telephony the ts field must be set to –1 for any timeslot. The actual timeslot on which the call is proceeding can be identified by calling the `call_details()` function. The timeslot will be valid as soon as the `EV_WAIT_FOR_OUTGOING` event is generated.

ts is used to specify the timeslot on which the call will be made. There are three possible modes for this input parameter:

1. Exclusive Timeslot

The `ts` field must contain the number of a valid timeslot on which the outgoing call will be made. Valid and barred timeslots can be obtained from validvector, returned by the `call_signal_info()` function.

```
struct out_xparms outxp;
outxp.ts = timeslot;
```

The device driver will attempt to make the outgoing call on the ‘exclusive’ timeslot provided. If, during establishment of the call, the network negotiates an alternate timeslot, the call will be discontinued. The call will only continue if the network can provide use of the specified timeslot.

For an exclusive slot map in Q.931 signalling systems that support slotmap, the ts field must be set to `USE_SLOTMAP` (-3). The slot map will be specified in the slotmap

field of the `sig_q931` structure. Slot map calls are currently supported on ETS300 (EuroISDN).

```
struct out_xparms outxp;
outxp.ts = USE_SLOTMAP;
outxp.unique_xparms.sig_q931.slotmap=slotmap;
```

2. Preferred Timeslot (using `CNF_TSPREFER`)

`ts` must contain the number of a valid timeslot on which the outgoing call will be made. Valid and barred timeslots can be obtained from validvector returned by the `call_signal_info()` function. The `CNF_TSPREFER` configuration switch must be set in the `cnf` input parameter.

```
struct out_xparms outxp;
outxp.cnf = CNF_TSPREFER;
outxp.ts = timeslot;
```

The device driver will attempt to make the call out on the 'preferred' timeslot provided. If, during establishment of the call, the network negotiates an alternate timeslot, the call will be allowed to continue.

The actual timeslot on which the call is proceeding may differ from that supplied by the application, but can be ascertained by calling the `call_details()` function.

For a preferred slot map in Q.931 signalling systems that support slot map calls, ts must contain `USE_SLOTMAP` (-3). The preferred slot map will be specified in in the *`slotmap`* field of the `sig_q931` structure. The `CNF_TSPREFER` configuration switch must be set in the cnf input parameter.

```
struct out_xparms outxp;
outxp.cnf = CNF_TSPREFER;
outxp.ts =USE_SLOTMAP;
outxp.unique_xparms.sig_q931.slotmap=slotmap;
```

The actual timeslots on which the slot map call is proceeding may differ from that supplied by the application, but can be ascertained by calling the `call_details()` function. The call will only proceed if a call with the same number of timeslots as requested in the *`slotmap`* field can be negotiated.

3. Any Timeslot

The input parameter `ts` must contain a value of -1.

```
struct out_xparms outxp;
outxp.ts = -1;
```

The device driver will use the first available timeslot on the specified network. The actual timeslot on which the call is proceeding can be identified by calling the `call_details()` function.

`cnf`

NOTE

For H.323, only CNF_REM_DISC and CNF_TSVIRTUAL are supported.

NOTE

For SIP, only CNF_TSVIRTUAL is supported

Is used to request additional functionality from the device driver. By 'OR'ing in configuration switches prior to invoking the function the device driver will modify its behaviour (depending upon the switches set) on a per call basis. The configuration switches currently supported are:

- **CNF_REM_DISC** (remote disconnect)
  This switch controls the drivers response to a 'far end' disconnect. Without the configuration switch, the driver will automatically respond to a 'far end' disconnect by releasing the call, the call state going to EV_IDLE.

  This method of working is compatible with all previous releases of the device driver.

  With the configuration switch set, when a 'far end' disconnect occurs, the event EV_REMOTE_DISCONNECT will be returned by the device driver. The application may now 'tidy up' any resources before finally disconnecting the call using the call_disconnect() function.

  This has two effects that may be useful:

  - Hold the channel until the system is ready to release the channel for further calls.
  - Retain any incoming call progress tones on the channel until released.

NOTE

Dependant on protocol, the disconnecting end may then send a release, because the far end has not received a response to the disconnect that was sent. e.g. For Q931 after T305/T306 has expired. This will result in the call going to the idle state.

- **CNF_CALL_CHARGE** (call charge event)
  This switch controls advice of charging. Without the configuration switch, the driver will neither return call-charging information nor return the EV_CALL_CHARGE event to the application.

  If the application wishes to obtain charging information, the application should set this switch and then either call the call_get_charge() function regularly, or expect the driver to return the EV_CALL_CHARGE event to the application indicating that new information has arrived.
- **CNF_TSPREFER** (preferred timeslot)
  This switch controls the way in which the timeslot parameter *ts* is interpreted. See the below on preferred timeslot for further information.
- **CNF_COMPLETE** (digits complete)
  The functionality of CNF_COMPLETE has been superseded by the *sending_complete* parameter as detailed in the next section.
- **CNF_TSVIRTUAL** (virtual call)
  below is the method used to open a virtual out call.

```
int outgoing_vcall( ACU_PORT_ID port, int *handle )
{
   int err;

   struct feature_out_xparms outgoing_param;

   INIT_ACU_CL_STRUCT( &outgoing_param );

   outgoing_param.net = port;
   outgoing_param.ts = -1;
   outgoing_param.cnf = CNF_TSVIRTUAL;

   outgoing_param.sending_complete = 1;

   strcpy( outgoing_param.destination_addr, DEST_ADDR );
   strcpy( outgoing_param.originating_addr, ORIG_ADDR );
   outgoing_param.feature_information = FEATURE_VIRTUAL;
```

```
    err = call_feature_openout( &outgoing_param );
    *handle = outgoing_param.handle;

    return err;
}
```

The timeslot is equal to -1 and the *cnf* flag is set for virtual (CNF_TSVIRTUAL), apart from these field, the rest is standard. We have also set the *feature_information* field to FEATURE_VIRTUAL, now if call details is done on a handle of a virtual call, the *feature_information* field of the details structure will return FEATURE_VIRTUAL, this way you can tell if the call is virtual or not.

The CNF_TSVIRTUAL option for the cnf flag is supported for SIP. When this option is set, a virtual SIP 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.

- **CNF_RAW_MSG** (ISUP only: enable reception of raw messages)
  When this switch is set, incoming call control messages become available via FEATURE_RAW_MSG. The extended event EV_EXT_RAW_MSG will be generated if call control messages were not previously queued.

sending_complete (not supported by IP telephony)

Is a Boolean value, that indicates if the number provided in the *destination_addr* field is the complete number.

- Overlap sending

```
sending_complete = 0;
```

Indicates that there may be more destination digits to follow, for example by calling the call_send_overlap() function.

- En-bloc sending

```
sending_complete = 1;
```

Indicates that there will be no more destination digits, all digits have been provided. The state of *sending_complete* will have different effects depending upon the protocol in use. If the protocol does not support *sending_complete* then the Boolean will be ignored.

destination_addr
Input character buffer contains a null terminated string of IA5 digits. This field can be:

- The whole of the number to be dialled (en bloc).
- Part of the number to be dialled (overlap sending, not supported for IP telephony).
- Empty, indicating no digits provided (overlap sending, not supported for IP telephony).

For IP Telephony, URI addressing may also 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.

It must be noted that if the address supplied does not conform exactly to the URI format, for example the <scheme>: section missing, the IP protocol will try to determine what has been entered.

For H.323 the supported schemes are h323: (H.323 URI), h323id: (H.323 ID), tel: (E.164 number), mailto: (email address) and http: (web URL).

For SIP the destination addr can be either:

- a valid SIP uri e.g. sip:user@1.2.3.4, sip:gateway.company.
- a numeric string e.g. 1234 that will be converted to sip:1234@proxy by the service. This string will form of the basis of the SIP To: header.
- a SIP call over TCP of the form <sip:user@host;transport=tcp>. Note that the <>s are required to correctly embed the transport=tcp as a URI, and NOT as a header parameter.

originating_addr
The input character buffer *originating_addr* can be supplied with a null terminated string of IA5 digits. This string represents the originating subscriber number. This string will be passed to the signalling system when the outgoing call is made. This provides for *originating_addr* to be specified on a per call basis.

For IP Telephony, URI addressing may also 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.

If the address supplied does not conform exactly to the URI format, for example the <scheme>: section missing, the IP service will try to determine what has been entered.

For SIP this is optional, however if supplied by the user must be of similar to the *destination_addr*. This string will form the basis of the SIP From: header.

NOTE

In the DASS signaling system, originating_addr may contain an ASCII string of extension number digits.

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

queue_id
The event queue to which events for this call should be sent. This must be either a unique event queue identity as returned by acu_allocate_event_queue() or zero to denote the default call event queue for the port that this call is made on.

unique_xparms
The input parameter *unique_xparms* is a union that provides extensions required by specific signalling systems. The union will vary depending upon the signalling system supported by the device driver and Aculab card. Each of the unions is described in chapter 8 which include:

Unique_xparms for Q931
Unique_xparms for DASS2
Unique_xparms for DPNSS
Unique_xparms for CAS
Unique_xparms for ISUP/SS7
Unique_xparms for IPtel

Return values

handle
When requesting to make a call, for example, by using call_openout(), the driver returns a unique call identifier value (non zero) in the handle field. This unique call identifier value must be used for all subsequent operations relating to the call.

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

4.19 call_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 device driver. If the driver is satisfied with the calling parameters, it will return a unique call identifier, the call handle for that call.

Synopsis

```
ACU_ERR call_openin(IN_XPARMS *indetailsp);

typedef struct in_xparms
{
  ACU_ULONG               size;              /* IN */
  ACU_CALL_HANDLE         handle;            /* OUT */
  ACU_PORT_ID             net;               /* IN */
  ACU_INT                 ts;                /* IN */
  ACU_INT                 cnf;               /* IN */
  ACU_EVENT_QUEUE         queue_id;          /* IN */
  ACU_ACT                 app_context_token; /* IN */
  union uniquex           unique_xparms;     /* IN */
} IN_XPARMS;
```

See section 8 for further details on using `uniquex`.

Input parameters

The `call_openin()` function takes a pointer *`indetailsp`*, to a structure,`IN_PARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`net`

Specifies the *`port_id`* on the Aculab card on which the call is to be made, as returned from `call_open_port()`.

`Ts`

NOTE

For IP Telephony the ts field must be set to –1 for any timeslot. The actual timeslot on which the call is proceeding can be identified by calling the call_details() function. The timeslot will be valid as soon as the EV_INCOMING_CALL_DET event is generated.

The *`ts`* field is used to specify the timeslot on which the incoming call will be received. There are three possible modes for this input parameter:

1. Exclusive timeslot

   The *`ts`* field must contain the number of a valid timeslot on which the call will be received. Valid and barred timeslots can be obtained from *`validvector`*, returned by the `call_port_info()` function.

```
struct in_xparms inxp;
inxp.ts = timeslot;
```

   The device driver will wait for any incoming call on the ‘exclusive’ timeslot provided. If, during establishment of an incoming call, the network provides a timeslot that does not match the ‘exclusive’ timeslot provided then the call will be discontinued.

2. Preferred timeslot

   The `ts` field must contain the number of a valid timeslot on which the call will be received. Valid and barred timeslots can be obtained from *`validvector`*, returned by the `call_port_info()` function. The `CNF_TSPREFER` configuration switch must be set in the `cnf` input parameter.

```
struct in_xparms inxp;
inxp.cnf = CNF_TSPREFER;
inxp.ts = timeslot;
```

The device driver will wait for any incoming call on the 'preferred' timeslot provided. If during establishment of an incoming call the network provides a timeslot that does not match the 'preferred' timeslot provided then the call will be allowed to continue.

The actual timeslot on which the call is proceeding may differ from that provided but can be ascertained by calling the `call_details()` function.

3. Any timeslot

   The `ts` field must contain a value of -1.

```
struct in_xparms inxp;
inxp.ts = -1;
```

The device driver will wait for an incoming call on any timeslot. The actual timeslot on which the call is proceeding can be ascertained by calling the `call_details()` function.

4. Any slot map or timeslot

   To open an incoming call to receive slot map calls on Q.931 signalling systems that support slot maps the timeslot (`ts`) field should be set to `USE_SLOTMAP -3`. The actual timeslot or slot map on which the call is proceeding can be ascertained by calling the `call_details()` function. Timeslots calls will also be detected in this mode. Slot map calls are currently supported on ETS300 (EuroISDN).

cnf

NOTE

For H.323 only `CNF_REM_DISC` is supported.

NOTE

For SIP the `cnf` flag is ignored

Is used to request additional functionality from the device driver. By 'OR'ing in configuration switches prior to invoking the function the device driver will modify its behaviour depending upon the switches set, on a per call basis. The configuration switches currently supported are:

- **`CNF_REM_DISC`** (remote disconnect)
  This switch controls the driver's response to a 'far end' disconnect. Without the configuration switch, the driver will automatically respond to a 'far end' disconnect by releasing the call, the call state going to `CS_IDLE`.

  This method of working is compatible with all previous releases of the device driver.

  When the configuration switch is set and a 'far end' disconnect occurs, the event `EV_REMOTE_DISCONNECT` will be returned by the device driver.

  The application may now 'tidy-up' any resources before finally disconnecting the call using the `call_disconnect()` function.

  This has two effects that may be useful:

  a) Holds the channel until the system is ready to release the channel for further calls.

b) Retains any incoming call progress tones on the channel until released.

- **CNF_CALL_CHARGE** (call charging)
  This switch controls advice of charging. Without the configuration switch the driver will neither return any call charging information nor return the EV_CALL_CHARGE event to the application.

  If the application wishes to obtain charging information, it should set this switch and either call the call_get_charge() function regularly or expect the driver will return the EV_CALL_CHARGE event to the application indicating that new information has arrived.
- **CNF_TSPREFER** (preferred timeslot)
  This switch controls how the timeslot parameter ts should be interpreted. See above on preferred timeslot for further information.
- **CNF_TSVIRTUAL** (virtual call)
  This switch opens the incoming side for a virtual call should be be used like this:

```
int incoming_vcall( ACU_PORT_ID port, int *handle )
{
   int err;

   struct in_xparms incoming_param;

   INIT_ACU_CL_STRUCT( &incoming_param );

   incoming_param.net = port;
   incoming_param.ts = -1;
   incoming_param.cnf = CNF_TSVIRTUAL;

   err = call_openin(&incoming_param);
   *handle = incoing_param.handle;

   return err;
}
```

  The timeslot is equal to -1 and the *cnf* flag is set for virtual (CNF_TSVIRTUAL), apart from these field, the rest is standard.

- **CNF_RAW_MSG** (ISUP 6.5.0 or later only: enable reception of raw messages)
  When this switch is set, incoming call control messages become available via FEATURE_RAW_MSG. The extended event EV_EXT_RAW_MSG will be generated if call control messages were not previously queued.
- **CNF_CALL_COLLECT** (Currently only ISUP 6.6.0 or later)
  This switch controls the behaviour of the driver when a call arrives with a "collect call request" parameter coded to indicate, "collect call requested". If the switch is set, an incoming call that has invoked an operator service to request that a call be charged to a called party will be handled normally. If the switch is not set, an incoming reverse-charge call will be rejected. This switch defaults to not set. Refer to the description of the collect_call_request_ind member of unique_xparms to discriminate between normal and reverse-charge calls.

queue_id
The event queue to which events for this call should be sent. This must be either a unique event queue identity as returned by acu_allocate_event_queue() or zero to denote the default call event queue for the port that this call is made on..

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

unique_xparms
The input parameter *unique_xparms* is a union that provides extensions required by specific signalling systems. The union will vary depending upon the signalling system

supported by the device driver and Aculab card. See the unique_xparms section for further details.

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.

4.20 call_send_overlap() - Sending overlap digits

This function may be used to send the destination address of an outgoing call one or more digits at a time. The function may also be used any time that a valid outgoing call handle is available and the state of the handle is `EV_WAIT_FOR_OUTGOING`. The outgoing call handle would have been obtained from the `call_openout()` function.

NOTE

Not supported for IP telephony or ANSI ISUP.

Synopsis

```
ACU_ERR call_send_overlap(OVERLAP_XPARMS *overlapp);

typedef struct overlap_xparms
{
  ACU_ULONG           size;                    /* IN */
  ACU_CALL_HANDLE     handle;                  /* IN */
  ACU_INT             sending_complete;        /* IN */
  char                destination_addr[MAXNUM];   /* IN */
} OVERLAP_XPARMS;
```

Input parameters

The `call_send_overlap()` function takes a pointer *`overlapp`*, to a structure, `OVERAP_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`handle`

This unique call identifier value is used to identify the outgoing call to which the destination digits are to be sent.

`sending_complete`

Is a Boolean value, that indicates if the number provided in the **`destination_addr`** is the complete number. Set to 0 if more destination digits will follow, set to 1 if all the digits have been provided.

The *`sending_complete`* field will have different effects depending upon the protocol in use. If the protocol does not support *`sending_complete`* then this field will be ignored.

`destination_addr`

The input character buffer `destination_addr` should contain a null terminated string of IA5 digits representing all or part of an ISDN destination subscriber number.

This field can be:

- The whole of the number to be dialed, (en bloc).
- Part of the number to be dialed, (overlap sending).
- Empty, indicating no digits provided, (overlap sending).

Return values

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

NOTE

Depending upon the signaling system in use, this function may be subject to an inter-digit timeout.

4.21 call_event() - Get a call event

This function is used to wait for call events to be returned that may have occurred on any incoming or outgoing call. The device driver issues an event when a change of state occurs.

In the V6 driver, events are always queued. Some events, especially `EV_IDLE` and `EV_REMOTE_DISCONNECT` can occur at any time, however, so it is advisable to ensure that applications can handle any event at any time. It is very important that an application always clear up any resources associated with a call when it goes to the idle state.

NOTE

An application can only receive call events for calls it opened.

Call handles are not interchangeable between applications.

The global event model can be used in multiple applications simultaneously without them poaching events from each other.

By default all calls are associated with a global call event queue. This is the queue that is checked when call_event() is used in global mode. Associating a call with any other event queue will prevent events for that call from being reported using the global call event mode.

Synopsis

```
ACU_ERR call_event(STATE_XPARMS *eventp);

typedef struct state_xparms
{
  ACU_ULONG           size;              /* IN */
  ACU_CALL_HANDLE     handle;            /* IN-OUT */
  ACU_LONG            state;             /* OUT */
  ACU_LONG            timeout;           /* IN */
  ACU_LONG            extended_state;    /* OUT */
  ACU_ACT             app_context_token; /* OUT */
  ACU_TOKEN           raw_msg_seq;       /* OUT */
} STATE_XPARMS;
```

Input parameters

The `call_event()` function takes a pointer, *`eventp`*, to a structure, `STATE_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`handle`

Must contain either zero or a valid call handle depending upon the mode of operation required of the `call_event()` function.

This function has two modes of operation:

- Global mode
  This mode must be used to receive call events from calls that are associated with the global call event queue (via the default call event queue for their respective port). To use this mode set *`handle`* to zero.
- Handle mode
  This mode must be used for calls that are associated with an event queue previously created by calling `acu_allocate_event_queue()`. Prior to calling this function the application should wait on the event queue and then set *`handle`* to the call handle retrieved by calling `acu_get_event_from_event_queue()`.

timeout

Specifies the mode of operation of the function and has the following values:

- **0** - The driver will return event information immediately to the application (polling).
- **+ve** - A positive value specifies a time in milliseconds that the application will wait for an event before returning.
- **-1** - The function will block until an event occurs in the driver or will return immediately if an event has occurred since the function was last invoked (blocking). If no event occurs then the process will block forever. To wake up a process blocked in the API a multi-tasking system may use the `call_disconnect()` function with the appropriate handle.

Return values

handle

The *`handle`* field will contain zero if there are no events. Otherwise *`handle`* will contain the call handle of the call on which the event occurred. The `handle` value may be directly used in functions that require a handle as an input parameter.

state

The *`state`* field will only be valid when *`handle`* contains a valid *`handle`* (non-zero) and will contain the current state of the call.

NOTE

It is possible for a number of similar events to be consolidated into a single event notification should an application be slow to process its event queue. This is done where a state indicates additional information is available such as EV_DETAILS, EV_NOTIFY, and EV_CALL_CHARGE.

- `EV_WAIT_FOR_INCOMING`
  Waiting for incoming call. The driver is waiting for an incoming call on this handle.
- `EV_INCOMING_CALL_DET`
  An incoming call has arrived. An incoming call has been detected on this call handle.
- `EV_CALL_CONNECTED`
  A call is connected. The driver has connected a call through to the application.
- `EV_WAIT_FOR_OUTGOING`
  An outgoing call is being made. The application has requested that an outgoing call be made and the driver is proceeding with the request.
- `EV_DETAILS (EV_INCOMING_DETAILS)`
  Incoming/outgoing call details. Additional incoming or outgoing call details are available in the driver. The application may use the `call_details()` function to retrieve the information.

  For incoming calls, an `EV_DETAILS` event occurs whenever new information has arrived. This may be additional DDI or CLI digits available in the `destination_addr` or `originating_addr` fields returned by the `call_details()` function.

  For outgoing calls, an `EV_DETAILS` event occurs as soon as the device driver knows the outgoing timeslot. This allows for the earliest connection of the speech channels. The timeslot value can be recovered by use of the `call_details()` function.
- `EV_OUTGOING_RINGING`
  Far end ringing. The outgoing call requested by the application has terminated on a subscriber.

- EV_REMOTE_DISCONNECT
  The EV_REMOTE_DISCONNECT event will be returned, (if the CNF_REM_DISC option was set), whenever the far end disconnects the call. This event may occur at any time during a call. The normal response to this event would be to disconnect the call using the call_disconnect() function.
- EV_WAIT_FOR_ACCEPT
  The EV_WAIT_FOR_ACCEPT event will be returned in response to the call_incoming_ringing() function to indicate that ringing is in progress. The time between function and event is undefined as some tone based signaling systems may take a few seconds to send the ring tone. The application should respond with the call_accept() function after receiving this event.
- EV_CALL_CHARGE
  The EV_CALL_CHARGE event will be returned (if the CNF_CALL_CHARGE option was set), whenever the device driver receives new call charge information. The information type, that is meter pulse or charge string, may be ascertained by use of the call_get_charge() function.
- EV_IDLE
  The channel is idle, The call has been terminated. The application must return the handle to the driver using the call_release() function.
- EV_HOLD
  The EV_HOLD event will be received in when a call enters the call hold state in response to a call_hold() request.
- EV_HOLD_REJECT
  The EV_HOLD_REJECT event will be received in response to a call_hold() if the request was rejected by the network.
- EV_TRANSFER_REJECT
  The EV_TRANSFER_REJECT event will be received if a call_transfer() request has been rejected.
- EV_RECONNECT_REJECT
  The EV_RECONNECT_REJECT event will be received if a call_reconnect() request has been rejected.
- EV_NOTIFY
  The EV_NOTIFY event is received when information is received pertaining to a call (such as user suspended). This information can be obtained by examining the *notify_indicator* field from the sig_q931 structure returned by the call_details() function.

NOTE

The EV_NOTIFY event is supported on Q.931 protocols only.

- EV_EXTENDED
  Indicates that an extended event has been received. See the description of the *extended_state* field for further information on extended events
- EV_OUTGOING_PROCEEDING
  The called party has all the digits necessary to proceed with the call.

NOTE

The EV_OUTGOING_PROCEEDING event is supported on Q.931 and ISUP protocols only.

- EV_PROGRESS
  A call has received a PROGRESS (protocol) message. A call on ISUP that receives a PROGRESS message may alternatively receive EV_OUTGOING_RINGING depending on the context of the call and contents of the PROGRESS message.

NOTE

The EV_PROGRESS event is supported on Q.931, ISUP, H.323 and SIP protocols.

EV_EMERGENCY_CONNECT

Indicates an incoming call, made in an emergency, has been automatically connected. The application should respond as it would for the EV_CALL_CONNECTED state.

NOTE

The EV_EMERGENCY_CONNECT is supported by certain CAS protocols only. Please see the CAS firmware release notes to find out which protocols it is applicable to.

EV_TEST_CONNECT

Indicates an incoming call, made to establish if the line is operational, has been automatically connected. The application should do nothing in response except wait for the calling party to initiate clearing.

NOTE

The EV_TEST_CONNECT is supported by certain CAS protocols only. Please see the CAS firmware release notes to find out which protocols it is applicable to.

extended_state

A value in the state field, EV_EXTENDED, allows an indication of the availability of feature information. This indicates the availability of information in the *extended_state* field.

Possible values for extended_state are defined in the header file under Extended Events. For example:

- EV_EXT_FACILITY
  Indicates that a FACILITY message has arrived. A call to call_feature_details() will retrieve the information.
- EV_EXT_UUI_PENDING
  Indicates that User-to-User data is waiting to be collected. Data will be in a queue, that must be emptied upon receiving this event. Empty the queue by calling call_feature_details() with details.feature.uui.command set to UU_GET_PENDING_DATA_CMD. Repeat the call to this function until the details.feature.uui.length field returned is 0 (i.e. no more data).
- EV_EXT_UUI_CONGESTED
  Indicates that User-to-User data has been lost due to throughput violation or other error. No further User to User messages should be sent until an EV_EXT_UNCONGESTED event has been received
- EV_EXT_UUI_UNCONGESTED
  Indicates that it is once more possible to transmit User to User.

- EV_EXT_UUS_SERVICE_REQUEST
  Occurs during active call phase if a FACILITY message is received indicating a User to User service 3 request. The reply to this request also generates this event.
- EV_EXT_HOLD_REQUEST
  Indicates that a request to put a call on hold has arrived.
- EV_EXT_RECONNECT_REQUEST
  Indicates that a request to 'reconnect' (or retrieve) call on hold has arrived.
- EV_EXT_TRANSFER_INFORMATION
  Indicates that information has arrived pertaining to call transfer.
- EV_EXT_DIVERSION
  Indicates that a call diversion message has arrived. A call to call_feature_details() can be used to retrieve the information.

NOTE

EV_EXT_DIVERSION will be seen after a call_openout() and before any EV_CALL_CONNECTED event.

- EV_EXT_RAW_DATA
  Indicates that a RAW_DATA message has arrived. A call to call_feature_details() will retrieve the information.

NOTE

An application must be prepared for any event to be returned by the driver and not expect the events to occur in a particular order. Particularly, the EV_REMOTE_DISCONNECT and EV_IDLE events may be received at any time and the application must be prepared for this. Applications should be designed with this in mind

- EV_EXT_NON_STANDARD_DATA
  Indicates that H.323 non-standard data has arrived. A call to call_feature_details() can be used to retrieve the information. This event is only applicable to the H.323 protocol.
- EV_EXT_RAW_MSG (ISUP only)
  Indicates that received call control messages are available for retrieval via FEATURE_RAW_MSG. call_feature_details() can be used to retrieve the information.

NOTE

EV_EXT_RAW_MSG is only returned by the driver when a call's queue of raw messages was previously empty. The event will not be returned for a call if a message is received when earlier messages still await collection.

- EV_EXT_FEATURE_ACTIVATION
  Indicates a call control message containing a 'Feature Activation Information Identifier' has been received. The contents of the identifier is retrieved using call_feature_details() and specifying the FEATURE_FEATURE_ACTIVATION feature type.

- `EV_EXT_INFORMATION_REQUEST`

  Indicates a call control message containing a 'Information Request' Information Identifier has been received. The contents of the identifier is retrieved using call_feature_details() and specifying the `FEATURE_INFORMATION_REQUEST` feature type.
- `EV_EXT_MEDIA_CHANGE_REQUEST` (H323 only)
  Indicates that a request to change the type of media on the call has been received. `call_media_details()` can be used to retrieve the information about this request and `call_change_media_accept()` or `call_change_media_reject()` should be used to accept or reject this proposed change.
- `EV_EXT_MEDIA_CHANGE_ACCEPT` (H323 only)
  Indicates that a request to change the type of media on the call has been accepted by the remote end.
- `EV_EXT_MEDIA_CHANGE_REJECT` (H323 only)
  Indicates that a request to change the type of media on the call has been rejected by the remote end.
- `EV_EXT_MEDIA_CHANGE_TIMEOUT` (H323 only)
  Indicates that a request to change the type of media on the call has timed out due to not having had a response from the remote end.
- `EV_EXT_MEDIA_CHANGE_COMPLETED` (H323 only)
  Indicates that a request to change the type of media on the call has completed and the original media connection has ended.

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

`raw_msg_seq` (ISUP only)

This field contains a sequence number that can be paired with the value provided by the field of the same name in `struct raw_msg_xparms`. If a call control event occurs as a result of a received network message, an application can use this field to match the call control event with the raw message retrieved with `call_feature_details()`. If the call control event did not occur as a result of a network message (e.g. a timeout), then the *`raw_msg_seq`* field will contain the value zero.

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

4.22 call_details() - Get call details

This function is used to gather the details of a current call, either incoming or outgoing, connected through the device driver.

Synopsis

```
ACU_ERR call_details(DETAIL_XPARMS *detailsp);

typedef struct detail_xparms
{
  ACU_ULONG           size;                       /* IN */
  ACU_CALL_HANDLE     handle;                     /* IN */
  ACU_LONG            timeout;                    /* OUT */
  ACU_INT             valid;                      /* OUT */
  ACU_INT             stream;                     /* OUT */
  ACU_INT             ts;                         /* OUT */
  ACU_INT             calltype;                   /* OUT */
  ACU_INT             sending_complete;           /* OUT */
  char                destination_addr[MAXADDR];  /* OUT */
  char                originating_addr[MAXADDR];  /* OUT */
  char                connected_addr[MAXADDR];    /* OUT */
  ACU_COMPAT_ACT      old_app_context_token;      /* OUT */
  ACU_ULONG           feature_information;        /* OUT */
  ACU_ACT             app_context_token;          /* OUT */
   ... padding for backwards compatibility ...
  union uniquex       unique_xparms;              /* OUT */
} DETAIL_XPARMS;
```

See section 8 for further details on using `uniquex`.

Input parameters

The `call_details()` function takes a pointer, *`detailsp`*, to a structure, `DETAIL_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

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

Return values

`timeout`
Not used in V6, but retained in structure for historic reasons

`valid`
Is a Boolean value, that 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

NOTE

When the valid field indicates that there is some information in the structure, this does not mean that all information has been received for this call.
stream.

Will contain the network stream number on which the call was received. For a full description of the stream numbering on Aculab cards consult the Switch Driver API Guide.

`ts`
Will contain the timeslot associated with the call. If a slot map is being used then ts

will contain `USE_SLOTMAP (-3)`. The slot map will then be in the *`slotmap`* field of the `sig_q931` structure.

NOTE

For IP telephony, As there is no physical timeslot for IP telephony calls, ts has no meaning.

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

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

`sending_complete` (not supported for IP telephony)

Is a Boolean, that indicates if the number provided in the *`destination_addr`* field is the complete number.

The `sending_complete` field may not be supported by all protocols.

`sending_complete == 0`

Indicates that there may be more destination digits to follow

`sending_complete == 1`

Indicates that there will be no more destination digits, all digits have been received.

(In Q931, this is an appropriate point in which to use the `call_proceeding()` API call)

`originating_addr and destination_addr`
Will contain the calling line identity (CLI) and direct dial in (DDI) digits respectively if received by the signalling system.

Additional digits may be returned in the *`destination_addr`* field if the calling party uses overlaps sending. If this is the case then the *`sending_complete`* field (if supported) should be used to determine when all digits have arrived. If the application requires more digits, the `call_details()` function may be reused until all the application is satisfied that all extension number digits are available.

`connected_addr`
Will contain the actual number of the party connected to a call. This may differ from the *`destination_addr`* field due to services such as call redirection.

`feature_information`
Contains an indication of whether any supplementary service information has been received and if so what type of information has been received. A separate API call, `call_feature_ details()`, must be used to retrieve this information.

This field can have any combination of the following values defined in the header file under `Feature Indications`.

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

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

`unique_xparms`
The input parameter `unique_xparms` is a union that provides extensions required by specific signalling systems. The union will vary depending upon the signalling system supported by the device driver and Aculab card. Each of the unions is described in

section 8.

NOTE

Be aware that the details returned by this function are not historical. In an application that is slow to process its event queue a number of EV_DETAILS events may be queued. The first call to call_details() will return all of the updated details signaled by the remaining EV_DETAILS events in the event queue.

4.23 call_incoming_ringing() - send incoming ringing

This function may be used to send the ringing message to the network 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. Use of the function will stop further call details, such as DDI digits, from being received. To allow enhancements to the driver API this call will in turn call `xcall_incoming_ringing()` which may be called directly.

Synopsis

```
ACU_ERR call_incoming_ringing(int handle);
ACU_ERR xcall_incoming_ringing(INCOMING_RINGING_XPARMS *ringingp);
typedef struct incoming_ringing_xparms
{
  ACU_ULONG           size;              /* IN */
  ACU_CALL_HANDLE     handle;            /* IN */
  union uniqueu       unique_xparms;     /* IN */
} INCOMING_RINGING_XPARMS;

union uniqueu
{
  /* see protocol specific structures in section 8 */
} unique_xparms;
```

Definitions for the `unique_xparms` parameters are detailed in section 8.

Q931 specific information

```
  struct
  {
    struct
    {
      char            ie[MAXPROGRESS];
      char            last_msg;
    } progress_indicator;
    struct
    {
      char            ie[MAXDISPLAY];
      char            last_msg;
    } display;
  } sig_931;
```

ISUP/SS7 Specific Information

```
  struct
  {
    struct
    {
      char            ie[MAXPROGRESS];
      char            last_msg;
    } progress_indicator;
    ACU_CHAR          charge_ind;
    ACU_CHAR          in_band;
    ACU_UCHAR         dest_category;
    struct
    {
      ACU_UCHAR       valid;
      ACU_UCHAR       value;
    } isdn_access_ind;
    struct
    {
      ACU_UCHAR       valid;
      ACU_UCHAR       value;
    } isdn_userpart_ind;
    struct
    {
      ACU_UCHAR       valid;
```

```
    ACU_UCHAR       value;
  } interworking_ind;
} sig_isup;
```

IP telephony (iptel) Specific Information

```
struct
{
  ACU_CHAR          destination_display_name;
  ACU_CODEC         codecs[MAXCODECS];
  MEDIA_SETTINGS    media_settings;
  union
  {
    struct
    {
      ACU_INT       send_early_media;
      ACU_INT       use_183_response_for_early_media;
      ACU_INT       send_reliable_provisional_response;
      ACU_CHAR      contact_address[MAXADDR];
    } sig_sip;
    struct
    {
      ACU_INT       h245_tunneling;
      ACU_INT       faststart;
      ACU_INT       early_h245;
      ACU_INT       progress_location
      ACU_INT       progress_description;
    } sig_h323;
  } protocol_specific;
} sig_iptel;
```

Input parameters

`call_incoming_ringing()`

The input parameter *`handle`* identifies the call that will send the incoming ringing message.

`xcall_incoming_ringing()`

The `xcall_incoming_ringing()` function takes a pointer, *`ringingp`*, to a structure, `INCOMING_RINGING_XPARMS`. The structure must be initialised before invoking the function.

`handle`

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

`unique_xparms`

See unique_xparms Q931 for `sig_q931` definitions

See unique_xparms ISUP for `sig_isup` definitions

See unique_xparms IP telephony for `sig_iptel` definitions

The following unique parameters are not detailed in *`unique_xparms`* and are specific to `incoming_ringing_xparms`:

Unique parameters for SIP

For SIP, the trivial use of `call_incoming_ringing()` or `xcall_incoming_ringing()` with no parameters set 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`* is set. In this case the codecs and `media_settings` elements may be used to configure the audio characteristics of the call.

`send_early_media`

To send early media in the `xcall_incoming_ringing()` then set this flag to 1. By default, set to 0, early media is `OFF` for `call_incoming_ringing()`, as a SIP ringing message is a 180 and 180s typically do not have SDP bodies (the protocol that negotiates early media). Note additional flags for `xcall_incoming_ringing()` are not

valid, unless this flag is set.

send_reliable_provisional_response
By default, set to 0, 18* messages are sent unreliably. Setting this flag to 1 forces them to be sent reliably. `ERR_PARM` is returned if the caller does not support the protocol extension required for reliable provisional responses.

use_183_response_for_early_media
Setting this will make `call_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.

contact_address
Used to build a non-default contact header. For chassis containing only one NIC card this field should be left blank. It will be in URI address format.

URIs are a well defined international standard for supporting multiple addressing types. They take the form:

```
<scheme>:<scheme-specific-part>
```

Examples of URIs could be:

```
sip:joe.bloggs@aculab.com
h323:joe.bloggs@aculab.com
mailto:joe.bloggs@aculab.com
http://www.aculab.com
tel:+441315610104
```

NOTE

The SIP service currently only supports the sip: uri.

IANA's (Internet Assigned Numbers Authority) current register of registered schemes can be found at http://www.iana.org/assignments/uri-schemes.

If the address supplied does not conform exactly to the URI format, for example, `<scheme>: section missing`, the IP protocol will try to determine what has been entered.

Return values

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

NOTE

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

4.24 call_accept() - Accept incoming call

This function may be used to accept the call after an incoming call has been indicated. This function would typically be used once the application had determined that the DDI digits received by the `call_details()` function are correct and can be supported. To allow enhancements to the driver API this call will in turn call `xcall_accept()`, which may be called directly.

Synopsis

```
ACU_ERR call_accept(int handle);

ACU_ERR xcall_accept(ACCEPT_XPARMS *acceptp);

typedef struct accept_xparms
{
  ACU_ULONG           size;              /* IN */
  ACU_CALL_HANDLE     handle;            /* IN */
  union uniqueu       unique_xparms;     /* IN */
} ACCEPT_XPARMS;

      union uniqueu
      {
        /* see protocol specific structures */
      } unique_xparms;
```

Definitions for the unique xparms parameters are detailed in section 8.

Q931 Specific Information

```
    struct
    {
      struct
      {
        ACU_UCHAR          ie[MAXPROGRESS];
        ACU_UCHAR          last_msg;
      } progress_indicator;
      struct
      {
        ACU_UCHAR          ie[MAXLOLAYER];
        ACU_UCHAR          last_msg;
      } lolayer;
      struct
      {
        ACU_UCHAR          ie[MAXDISPLAY];
        ACU_UCHAR          last_msg;
      } display;
      char                 connected_addr[MAXNUM]
      ACU_UCHAR            conn_numbering_type;
      ACU_UCHAR            conn_numbering_plan;
      ACU_UCHAR            conn_numbering_presentation;
      ACU_UCHAR            conn_numbering_screening;
    } sig_q931;
```

ISUP/SS7 specific Information

```
    struct
    {
      struct
      {
        ACU_UCHAR          ie[MAXPROGRESS];
        ACU_UCHAR          last_msg;
      } progress_indicator;
      struct
      {
        ACU_UCHAR          ie[MAXLOLAYER];
        ACU_UCHAR          last_msg;
      } lolayer;
```

```
  char                connected_addr[MAXNUM]
  ACU_UCHAR           conn_nature_of_address;
  ACU_UCHAR           conn_numbering_plan;
  ACU_UCHAR           conn_numbering_presentation;
  ACU_UCHAR           conn_numbering_screening;
  ACU_CHAR            charge_ind;
  ACU_UCHAR           dest_category;
  struct
  {
    ACU_UCHAR         valid;
    ACU_UCHAR         value;
  } isdn_access_ind;
  struct
  {
    ACU_UCHAR         valid;
    ACU_UCHAR         value;
  } isdn_userpart_ind;
  struct
  {
    ACU_UCHAR         valid;
    ACU_UCHAR         value;
  } interworking_ind;
} sig_isup;
```

IP telephony specific Information

```
struct
{
  ACU_CHAR                destination_display_name[MAXDISPLAY];
  ACU_CHAR                originating_display_name[MAXDISPLAY];
  ACU_CODEC               codecs[MAXCODECS];
  MEDIA_SETTINGS          media_settings;
  union protocol_union    protocol_specific;
} sig_iptel;
```

Input parameters

call_accept()

The input parameter *handle* identifies the call that is to be accepted.

xcall_accept()

The xcall_accept() function takes a pointer, *acceptp*, to a structure,ACCEPT_XPARMS. The structure must be initialised before invoking the function.

handle

Identifies the call that is to be accepted.

unique_xparms

The input parameter *unique_xparms* is a union that provides extensions required by specific signalling systems. The union will vary depending upon the signalling system supported by the device driver and Aculab card. Each of the unions is described in chapter 8 which include:

Unique_xparms for Q931
Unique_xparms for ISUP/SS7
Unique_xparms for IP telephony

Return values

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

4.25 call_getcause() - Get idle cause

This function can be used to return the clearing cause when an incoming or outgoing call clears. The returned clearing cause will only be valid when the call state is either `EV_IDLE` or `EV_REMOTE_DISCONNECT`. To allow enhancements to the driver API this call will in turn call `xcall_getcause()` call, which may be called directly.

Synopsis

```
ACU_ERR call_getcause(CAUSE_XPARMS *causep);

typedef struct cause_xparms
{
  ACU_ULONG           size;          /* IN */
  ACU_CALL_HANDLE     handle;        /* IN */
  ACU_INT             cause;         /* OUT */
  ACU_INT             raw;           /* OUT */
  ACU_INT             location;      /* OUT */
} CAUSE_XPARMS;

ACU_ERR xcall_getcause(DISCONNECT_XPARMS *causep);

struct disconnect_xparms
{
  ACU_ULONG           size;              /* IN */
  ACU_CALL_HANDLE     handle;            /* IN */
  ACU_INT             cause;             /* OUT */
  union uniqueu       unique_xparms;     /* OUT */
} DISCONNECT_XPARMS;
```

The structure of the `disconnect_xparms` is detailed in section 9.

Input parameters

call_getcause()

The `call_getcause()` function takes a pointer, *causep*, to a structure,`CAUSE_XPARMS`.

handle
Is used to identify the call that is to be examined.

Return values

cause_xparms
cause
Will contain the 'generic' reason for the incoming or outgoing call going to the `EV_IDLE` or `EV_REMOTE_DISCONNECT` state and will be one of the standard set of `LC_xxxx` clearing causes.

raw
Will contain the network-supplied cause for the incoming or outgoing call going to the `EV_IDLE` or `EV_REMOTE_DISCONNECT` state and will be dependent upon the protocol in use

location
Allows the retrieval of the cause location for EuroISDN, QSIG and H.323. The possible values are:

```
L_USER
L_PRIV_NET_LOCAL_USER
L_PUB_NET_LOCAL_USER
L_TRANSIT_NET
L_PUB_NET_REMOTE_USER
L_PRIV_NET_REMOTE_USER
L_BEYOND_IW_POINT
```

xcall_getcause()

The `xcall_getcause()` function takes a pointer, `causep`, to a

structure,`DISCONNECT_XPARMS` as detailed in section 9.

On successful completion of `call_getcause()` or `xcall_getcause()`, a value of zero is returned; otherwise, a negative value will be returned indicating the type of error.

Each signalling system will return a `raw` value containing the network supplied cause for the incoming or outgoing call going to the `EV_IDLE` or `EV_REMOTE_DISCONNECT` state and will be dependent upon the protocol in use.

4.26 call_disconnect() - Disconnect call

This function can be used to disconnect an incoming or outgoing call currently routed through the driver. If the `call_disconnect()` function is successful, the driver will start the disconnect procedure and will return immediately to the calling process. When the call has been disconnected, the state will be `EV_IDLE`. The `call_release()` function must be used to give back the handle to the driver. To allow enhancements to the driver API this call will in turn call the `xcall_disconnect()` call, which may be called directly.

Synopsis

```
ACU_ERR call_disconnect(CAUSE_XPARMS *causep);

typedef struct cause_xparms
{
  ACU_ULONG               size;         /* IN*/
  ACU_CALL_HANDLE         handle;       /* IN*/
  ACU_INT                 cause;        /* IN*/
  ACU_INT                 raw;          /* IN*/
  ACU_INT                 location;     /* IN*/
} CAUSE_XPARMS;

ACU_ERR xcall_disconnect(DISCONNECT_XPARMS *causep);

struct disconnect_xparms
{
  ACU_ULONG           size;             /* IN*/
  ACU_CALL_HANDLE     handle;           /* IN*/
  ACU_INT             cause;            /* IN*/
  union uniqueu       unique_xparms;    /* IN*/
} DISCONNECT_XPARMS;
```

The structure of the `disconnect_xparms` is detailed in section 9.

Input parameters

call_disconnect()

The `call_disconnect()` function takes a pointer, *`causep`*, to a structure,`CAUSE_XPARMS`. The structure must be initialised in the following way before invoking the function.

`handle`
Is used to identify the call that is to be disconnected.

`cause`
May be used to provide the device driver with the generic clearing cause for the call. The cause must one from the standard set of generic clearing causes. (See Appendix E for standard clearing causes).

`raw`
The `raw` field may be used to provide the protocol specific clearing cause. *`raw`* must contain a value that is appropriate for the protocol in use.

`location`
Allows the setting of the cause location for EuroISDN, QSIG and H.323. The possible values are:

```
L_USER
L_PRIV_NET_LOCAL_USER
L_PUB_NET_LOCAL_USER
L_TRANSIT_NET
L_PUB_NET_REMOTE_USER
L_PRIV_NET_REMOTE_USER
L_BEYOND_IW_POINT
```

With ISUP/SS7 the location value specified in the ss7 config file will be used.

If a user specified location value is required use the xcall_disconnect() function and specify the location value in the DISCONNECT_XPARMS *unique_xparms* structure.

xcall_disconnect()

This call allows the application to transmit extra information when disconnecting an incoming or outgoing call. The xcall_disconnect() function takes a pointer, *causep*, to a structure,DISCONNECT_XPARMS as detailed in section 9.

Return values

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

NOTE

If there is a call in progress when call_disconnect() is invoked, the driver will initiate the disconnect procedure and will immediately return control to the calling process.

Determining use of cause and raw

The device driver will determine which clearing cause to use, ie. Either *cause* or *raw* by the following:

cause	raw	
0	0	driver uses default clearing
0	x	driver uses raw cause
x	0	driver uses generic cause
x	x	driver uses raw cause

Effects of CNF_REM_DISC on call clearing

The CNF_REM_DISC 'call time' configuration switch may be used for incoming or outgoing calls to change the driver's automatic response to a 'far end' clearing.

default

The driver will automatically respond to the remote disconnect and clear the call. The Application interface will go to the EV_IDLE state.

This procedure maintains compatibility with previous versions of the device driver and ensures backward compatibility with existing applications.

CNF_REM_DISC selected

In this mode the driver does not automatically respond to the remote disconnect but issues the EV_REMOTE_DISCONNECT state to the application.

This has two effects:

1. The channel is held until the application is ready to release it.
2. It retains any incoming call progress tones on the channel until released.

On receipt of this state, the application should tidy up and use the call_disconnect() function when ready.

4.27 call_release() - Release call

This function must be used to relinquish ownership of a call handle in response to call termination (EV_IDLE). If the call_release() function is successful, the driver will disconnect the call and the call handle will be closed. The handle may no longer be used by the application. To allow enhancements to the driver API this call will in turn call xcall_release(), which may be called directly.

Synopsis

```
ACU_ERR call_release(CAUSE_XPARMS *causep);
typedef struct cause_xparms
{
  ACU_ULONG                size;          /* IN */
  ACU_CALL_HANDLE          handle;        /* IN */
  ACU_INT                  cause;         /* IN */
  ACU_INT                  raw;           /* IN */
  ACU_INT                  location;      /* IN */
} CAUSE_XPARMS;
ACU_ERR xcall_release(DISCONNECT_XPARMS *causep);

struct disconnect_xparms
{
  ACU_ULONG           size;               /* IN */
  ACU_CALL_HANDLE     handle;             /* IN */
  ACU_INT             cause;              /* IN */
  union uniqueu       unique_xparms;      /* IN */
}  DISCONNECT_XPARMS;
```

The structure of the DISCONNECT_XPARMS is detailed in section 9.

Input parameters

call_release()

The call_release() function takes a pointer, *causep*, to a structure,CAUSE_XPARMS. The structure must be initialised in the following way before invoking the function.

handle
This is used to identify the call that is to be released. This call must have reached the EV_IDLE state prior to this API function being used.

cause
May be used to provide the device driver with the generic clearing cause for the call. The cause must contain one of the standard set of generic clearing causes (See appendix E for standard clearing causes).

raw
May be used to provide the protocol specific clearing cause. The input parameter raw must contain a value that is appropriate for the protocol in use.

location
Allows the retrieval of the cause location for EuroISDN, QSIG and H.323. The possible values are:

```
L_USER
L_PRIV_NET_LOCAL_USER
L_PUB_NET_LOCAL_USER
L_TRANSIT_NET
L_PUB_NET_REMOTE_USER
L_PRIV_NET_REMOTE_USER
L_BEYOND_IW_POINT
```

xcall_release()

This call allows the application to transmit extra information when relinquishing ownership of a call handle in response to call termination. The xcall_release() function takes a pointer, *causep*, to a structure,DISCONNECT_XPARMS as detailed in

section 9.

Return values

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

If there is a call in progress when `call_release()` is invoked, the calling process will block in the driver until the call has been disconnected by the driver. Control will then be returned to the application. This period is entirely dependent upon the network.

For single threaded applications, this may be a problem. If there is a call in progress, a better procedure would be to use the following method:

```
call_disconnect()
```

```
wait for idle
```

```
call_release()
```

This method will disconnect the call but will not result in the application blocking in the device driver. This ensures that all resources associated with the call, i.e. the handle, are cleared.

NOTE

Calling call_release() while a call is in progress results in the application blocking the driver for a protocol determined time, which can be in excess of two minutes.

Determining use of cause and raw

The device driver will determine which clearing cause to use, i.e. either cause or raw by the following:

cause	raw	
0	0	driver uses default clearing
0	x	driver uses raw cause
x	0	driver uses generic cause
x	x	driver uses raw cause

Advanced Call Control

4.28 call_feature_openout() - Open for outgoing (with features)

It is possible to transmit feature information in a call setup message. An enhanced version of `call_openout()` is needed to include extra parameters.

Synopsis

```
ACU_ERR call_feature_openout (FEATURE_OUT_XPARMS* feature_out);

typedef struct feature_out_xparms
{
  ACU_ULONG           size;                       /* IN */
  ACU_CALL_HANDLE     handle;                     /* OUT */
  ACU_PORT_ID         net;                        /* IN */
  ACU_INT             ts;                         /* IN */
  ACU_INT             cnf;                        /* IN */
  ACU_INT             sending_complete;           /* IN */
  char                destination_addr[MAXADDR];  /* IN */
  char                originating_addr[MAXADDR];  /* IN */
  ACU_ULONG           feature_information;        /* IN */
  ACU_INT             message_control;            /* IN */
  ACU_ACT             app_context_token;          /* IN */
  ACU_EVENT_QUEUE     queue_id;                   /* IN */
  union uniquex       unique_xparms;              /* IN */
  union feature_union feature;                    /* IN */
} FEATURE_OUT_XPARMS;
```

See section 8 for further details on using `uniquex`.

See section 10 for details on using `feature_union`.

Input parameters

The `call_feature_openout()` structure takes a pointer, *`feature_out`*, to a structure,`FEATURE_ OUT_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`net`
Specifies the `port_id` on the Aculab card on which the call is to be made, as returned from `call_open_port()`.

`ts`
Is used to specify the timeslot on which the call will be made.

`cnf`

NOTE

For H.323 only CNF_REM_DISC is supported.

Is used to request additional functionality from the device driver. By 'OR'ing in configuration switches prior to invoking the function the device driver will modify its behaviour (depending upon the switches set) on a per call basis. The configuration switches currently supported are:

- `CNF_REM_DISC` (remote disconnect)
  This switch controls the drivers response to a `'far end'` disconnect. Without the configuration switch, the driver will automatically respond to a 'far end' disconnect by releasing the call, the call state going to `CS_IDLE`.

  This method of working is compatible with all previous releases of the device driver.

  With the configuration switch set, when a `'far end'` disconnect occurs, the state

CS_REMOTE_DISCONNECT or event EV_REMOTE_DISCONNECT will be returned by the device driver. The application may now 'tidy up' any resources before finally disconnecting the call by using call_disconnect().

This has two effects that may be useful:

- hold the channel until the system is ready to release the channel for further calls.
- retain any incoming call progress tones on the channel until released.

- CNF_CALL_CHARGE (call charge event)
  This switch controls advice of charging. Without the configuration switch, the driver will neither return call-charging information nor return the EV_CALL_CHARGE event to the application.

  If the application wishes to obtain charging information, the application should set this switch and then either call the call_get_charge () function regularly, or expect the driver to return the EV_CALL_CHARGE event to the application indicating that new information has arrived.
- CNF_TSPREFER (preferred timeslot)
  This switch controls the way in which the timeslot parameter ts is interpreted. See the below on preferred timeslot for further information.
- CNF_COMPLETE (digits complete)
  The functionality of CNF_COMPLETE has been superseded by the *sending_complete* parameter as detailed in the next section.
- CNF_RAW_MSG (ISUP 6.5.0 or later only: enable reception of raw messages)
  When this switch is set, incoming call control messages become available via FEATURE_RAW_MSG. The extended event EV_EXT_RAW_MSG will be generated if call control messages were not previously queued.

sending_complete (not supported by H.323)
Is a Boolean value that indicates if the number provided in the *destination_addr* field is the complete number.

- Overlap sending

```
sending_complete = 0;
```

  Indicates that there may be more destination digits to follow, for example by calling the call_send_overlap() function.

- En-bloc sending

```
sending_complete = 1;
```

  Indicates that there will be no more destination digits, all digits have been provided. The state of *sending_complete* will have different effects depending upon the protocol in use. If the protocol does not support *sending_complete* then the Boolean will be ignored.

destination_addr
The input character buffer contains a null terminated string of IA5 digits (0-9). This field can be:

- The whole of the number to be dialed (en bloc).
- Part of the number to be dialed (overlap sending, not supported for IP Telephony).
- Empty, indicating no digits provided (overlap sending, not supported for IP Telephony).

The digits supplied are copied, (***not*** concatenated) to *destination_addr*. If, when initiating the outgoing call, the *sending_complate* parameter was set to 1, then the

destination_addr field may contain a '!' to indicate that the number is complete.

For IP Telephony, URI addressing may also 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.

It must be noted that if the address supplied does not conform exactly to the URI format, for example the <scheme>: section missing, the IP service will try to determine what has been entered.

originating_addr

The input character buffer *originating_addr* can be supplied with a null terminated string of IA5 digits. This string represents the originating subscriber number. This string will be passed to the signalling system when the outgoing call is made. This provides for *originating_addr* to be specified on a per call basis. If the *originating_addr* field is empty then *ournum* will be used instead.

For IP Telephony, URI addressing may also 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.

It must be noted that if the address supplied does not conform exactly to the URI format, for example the <scheme>: section missing, the IP service will try to determine what has been entered.

NOTE

In the DASS signaling system, originating_addr may contain a null terminated ASCII string of extension number digits.

feature_information

NOTE

For H.323 only FEATURE_NON_STANDARD, FEATURE_DIVERSION and FEATURE_TRANSFER are supported

Is used to indicate the type of feature that is contained in *feature_union*. Only one feature type can be specified at a time. The following values are valid when making an outgoing call:

```
FEATURE_FACILITY
FEATURE_USER_USER
FEATURE_DIVERSION
FEATURE_TRANSFER
FEATURE_RAW_DATA
FEATURE_NON_STANDARD
FEATURE_RAW_MSG
```

In addition, the following flags control the type of call that is made and can be ORed with another feature type:

```
FEATURE_REGISTER (for ETS300)
FEATURE_VIRTUAL  (for QSIG)
```

For example if the call is to include facility information then the *feature_information* field should have the value FEATURE_FACILITY. If the call is to be a virtual call with Facility information then this field should be set to be FEATURE_FACILITY | FEATURE_VIRTUAL.

NOTE

To make a virtual call the feature_information field must have the

FEATURE_VIRTUAL or FEATURE_REGISTER bit set and the timeslot field ts set to -1.

message_control
Should be set to one of the following values:

`CONTROL_DEFAULT` (call setup message will be sent immediately)
`CONTROL_DEFERRED_SETUP` (wait for further information to send in the call setup message)

If `CONTROL_DEFERRED_SETUP` is used, more feature information to be sent in the call setup message will be provided by subsequent calls to call_feature_send().

NOTE

The functionality provided by the CONTROL_DEFERRED_SETUP value is only supported with ETS300 (EuroISDN), QSIG and H.323.

app_context_token
The *app_context_token* field is set by `acu_set_card_app_context_token()`, which is used to associate application-defined data with a specific card.

queue_id
The event queue to which events for this call should be sent. This must be either a unique event queue identity as returned by `acu_allocate_event_queue()` or zero to denote the default call event queue for the port that this call is made on.

feature
The *feature* union will be used to transmit features based on the value of the field *feature_information*. For example if the *feature_information* field has the value `FEATURE_DIVERSION`, then the `diversion` part of the union will be used.

Return Values

Handle
When this API call is successful, this field contains the handle for this call. This unique call identifier value must be used for all subsequent operations relating to the call.

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

4.29 call_feature_enquiry() - make an outgoing enquiry call with feature information

During the process of call transfer, this function allows an application to make an enquiry call, i.e. an outgoing call to a third party, with feature information.

Synopsis

```
ACU_ERR call_feature_enquiry (FEATURE_OUT_XPARMS* feature_out);

typedef struct feature_out_xparms
{
  ACU_ULONG               size;                    /* IN */
  ACU_CALL_HANDLE         handle;                  /* OUT */
  ACU_PORT_ID             net;                     /* IN */
  ACU_INT                 ts;                      /* IN */
  ACU_INT                 cnf;                     /* IN */
  ACU_INT                 sending_complete;        /* IN */
  char                    destination_addr[MAXADDR]; /* IN */
  char                    originating_addr[MAXADDR]; /* IN */
  ACU_ULONG               feature_information;     /* IN */
  ACU_INT                 message_control;         /* IN */
  ACU_ACT                 app_context_token;       /* IN */
  ACU_EVENT_QUEUE         queue_id;                /* IN */
  union uniquex           unique_xparms;           /* IN */
  union feature_union     feature;                 /* IN */
} FEATURE_OUT_XPARMS;
```

See section 8 for further details on using `uniquex`.

See section 10 for details on using `feature_union`.

Input parameters

The `call_feature_enquiry()` structure takes a pointer, *`feature_out`*, to a structure, `FEATURE_ OUT_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

The function is essentially the same as `call_feature_openout()`, having all of the same call states and events. The function registers the enquiry call requirement with the device driver. If the driver is satisfied with the calling parameters, it will return a unique call handle. The call handle must then be used in all successive call control related operations for this call.

NOTE

When choosing the time slot it is important to be aware of the type of transfer available for the protocol. For ETS300 and VN3 it is possible to use the timeslot of the held call and thus use fewer timeslots. To initiate such an enquiry the timeslot of the held party must be specified in the timeslot field of feature_out_xparms.

Return values

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

4.30 call_feature_details() - Get feature information

Supplementary service information may arrive at different stages during the lifetime of a call. An indication of the availability of this information is found in the *feature_information* field after a call to call_details(). To retrieve the information call_feature_details() should be used.

Synopsis

```
ACU_ERR call_feature_details (FEATURE_DETAIL_XPARMS* feature_detailsp);

typedef struct feature_detail_xparms
{
  ACU_ULONG               size;              /* IN */
  ACU_CALL_HANDLE         handle;            /* IN */
  ACU_PORT_ID             net;               /* IN */
  ACU_ULONG               feature_type;      /* IN-OUT */
  ACU_INT                 message_control;   /* IN */
  union feature_union     feature;           /* OUT */
} FEATURE_DETAIL_XPARMS;
```

See section 10 for details on using feature_union.

Input parameters

The call_feature_details() function takes a pointer, *feature_detailsp*, to a structure,FEATURE_DETAIL_XPARMS. The structure must be initialised before invoking the function.

handle

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

net

Specifies the number of the network outlet on the Aculab card on which the call is to be made as returned from call_open_port().

feature_type

The *feature_type* field must contain one of the following values to indicate to the driver which feature is requested (as there may be more than one present).

```
FEATURE_FACILITY
FEATURE_USER_USER
FEATURE_DIVERSION
FEATURE_HOLD_RECONNECT
FEATURE_TRANSFER
FEATURE_RAW_DATA
FEATURE_NON_STANDARD
FEATURE_RAW_MSG
FEATURE_FEATURE_ACTIVATION
FEATURE_INFORMATION_REQUEST
```

NOTE

Only the following feature types are supported for H.323 :

```
FEATURE_NON_STANDARD
FEATURE_DIVERSION
FEATURE_TRANSFER
FEATURE_CALL_WAITING
FEATURE_HOLD_RECONNECT
```

NOTE

Only the information for one supplementary service can be retrieved at one time. If more than one type of supplementary service information is available then this API call should be used again for each type.

On return, this field contains the feature type returned, or 0 if none (e.g. no more) of the specified type is available.

`message_control`
Used when `raw_data` is required in a message other than the facility message. When raw data is required the `message_control` field should contain `CONTROL_NEXT_CC_MESSAGE`. The raw data will be included in the next call control message. For example, if `call_accept()` is called next, the raw data will be included in the `CONNECT` message.

Return values

`feature_type`

Contains the feature type returned, or 0 if none is available.

`feature`

For further details, `see feature_xparms` - section 10.

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

4.31 call_feature_send() - Sending feature information

`call_feature_send()` can be used to transmit feature information at different stages during the lifetime of a call, or for a feature related to a specific port.

Synopsis

```
ACU_ERR call_feature_send(FEATURE_DETAIL_XPARMS* feature_detailsp);

typedef struct feature_detail_xparms
{
  ACU_ULONG           size;              /* IN */
  ACU_CALL_HANDLE     handle;            /* IN */
  ACU_PORT_ID         net;               /* IN */
  ACU_ULONG           feature_type;      /* IN */
  union feature_union feature;           /* IN */
  ACU_INT             message_control;   /* IN */
} FEATURE_DETAIL_XPARMS;
```

See section 10 for details on using `feature_union`.

Input parameters

The `call_feature_send()` function takes a pointer, *`feature_detailsp`*, to a structure,`feature_detail_xparms`. 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. When `restart_channels_xparms` is used to restart channels on a port this should be set to zero.

`net`

When `restart_channels_xparms` is used this field should be set to the network port. Otherwise it should be set to zero.

`feature_type`

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

```
FEATURE_FACILITY
FEATURE_USER_USER
FEATURE_DIVERSION
FEATURE_HOLD_RECONNECT
FEATURE_TRANSFER
FEATURE_RAW_DATA
FEATURE_NON_STANDARD
FEATURE_RAW_MSG
FEATURE_CALL_WAITING
FEATURE_ADDRESSED_NON_STANDARD_DATA
FEATURE_RESTART_CHANNELS
FEATURE_FEATURE_ACTIVATION
FEATURE_INFORMATION_REQUEST
FEATURE_NAME_PRESENTATION
```

NOTE

Only the following feature types are supported for H.323 :

```
FEATURE_NON_STANDARD
FEATURE_DIVERSION
FEATURE_HOLD_RECONNECT
FEATURE_TRANSFER
FEATURE_CALL_WAITING
FEATURE_ADDRESSED_NON_STANDARD_DATA
```

`message_control`

The *`message_control`* field is used to govern which call control message the

feature should be sent in. The values this field can take are:

CONTROL_DEFAULT
Facility information element is sent immediately by the current call control function

CONTROL_NEXT_CC_MESSAGE
Feature information element is sent in the next call control message that is called (e.g. if call_disconnect() is called, it will be attached to the disconnect message)

CONTROL_DEFERRED
Used with the call_feature_send() function, used to delay sending of the facility message until further feature information elements have been added via subsequent call_feature_send() function calls.

CONTROL_DEFERRED_SETUP
Used with the call_feature_openout() function, used to delay sending of the setup message until further feature information elements have been added via subsequent call_feature_send() function calls.

CONTROL_EXTRA_INFO
Used with the call_feature_send() function. Can be used multiple times after CONTROL_DEFERRED to add further feature information elements to the facility message.

CONTROL_EXTRA_INFO_SETUP
Used in call_feature_send() after a call to the call_feature_openout() function. Can be used multiple times after CONTROL_DEFERRED_SETUP to add further feature information elements to the setup message.

CONTROL_LAST_INFO
Used in call_feature_send() after a previous call to the call_feature_send() function. Used to indicate this is the last feature information element to be added to the facility message

CONTROL_LAST_INFO_SETUP
Used in call_feature_send() after a call to the call_feature_openout() function. Used to indicate this is the last feature information element to be added to the setup message.

Example: This is used when *raw_data* is required in a message other than the facility message. When raw data is required the *message_control* field should contain CONTROL_NEXT_CC_MESSAGE. The raw data will be included in the next call control message. For example, if call_accept() is called next, the raw data will be include in the CONNECT message.

Return values

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

SPECIAL NOTE

For QSIG only there is now the possibility of responding to call reroute requests from the network.

To accept a request it is necessary to send a 'return result' message before clearing the call with call_disconnect(). If this is not done the network will not know if the call reroute has been successful call_feature_send() should be sent with the following parameters:

- *feature_type* set to FEATURE_DIVERSION
- *operation* set to OP_CALL_REROUTE_REQ
- *operation_type* set to RETURN_RESULT

To reject a call reroute request you need to send a 'return result' message. The network may still be able to complete the original call but this could result in a non-optimal route. If the network cannot complete the original call without a reroute then it will disconnect the call. If rejected the request for reroute `call_feature_send()` should be set with the following parameters:

- *`feature_type`* set to `FEATURE_DIVERSION`
- *`operation`* set to `OP_CALL_REROUTE_REQ`

To decline the request for reroute, *`operation_type`* should be set to `RETURN_ERROR` and error should be set to one of :

```
0       userNotSubscribed
3       notAvailable
10      supplementaryServiceInteractionNotAllowed
11      resourceUnavailable
12      invalidDivertedToNr
14      specialServiceNr
15      diversionToServedUserNr
24      numberOfDiversionsExceeded
1008    unspecified
```

After a return error the network can choose to allow the call to continue, otherwise it will disconnect the call.

4.32 call_setup_ack() - Send setup acknowledge

This function may be used on an incoming call to send a Q.931 `SETUP_ACKNOWLEDGE` message to the calling party. The use of this function is supported in conjunction with the `-cSU` switch (see Call, Switch and Speech driver Installation Guide and release notes). This disables the automatic response of the driver to a Q.931 `SETUP` message and allows the application to provide additional information to the network. This function is dependent on the signalling system and reference should be made to the appropriate specification for the protocol.

NOTE

Not supported for IP Telephony.

Synopsis

```
ACU_ERR call_setup_ack(SETUP_ACK_XPARMS *setup_ackp);

typedef struct setup_ack_xparms
{
  ACU_ULONG                size;              /* IN */
  ACU_CALL_HANDLE          handle;            /* IN */
  union
  {
    struct
    {
      struct
      {
        ACU_UCHAR          ie[MAXPROGRESS];
        ACU_UCHAR          last_msg;
      } progress_indicator;                   /* IN */
      struct
      {
        ACU_UCHAR          ie[MAXDISPLAY];
        ACU_UCHAR          last_msg;
      } display;                              /* IN */
    } sig_q931;
  } unique_xparms;
} SETUP_ACK_XPARMS;
```

Input parameters

The `call_setup_ack()` function takes a pointer, *`setup_ackp`*, to a structure,`SETUP_ACK_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`handle`

The *`handle`* field identifies the call that will send the setup acknowledge message.

unique_xparms system-specific fields - Q931 structure

`progress_indicator`

The *`progress_indicator`* field can be used to indicate events pertaining to the call regarding interworking or in-band information. The `ie` field of the `progress_indicator` structure should contain the progress information, that is sent transparently to the other end. The `last_msg` field should be left blank. This information is dependent on the signalling system and reference should be made to the appropriate specification for the protocol.

`Display (MAXDISPLAY = 34)`

The display field can be used to transmit information that may be displayed by the user.

display example (This supplies IA5 information “ABCD”.)

```
display.ie[0] = 0x04 (four bytes follow)
```

```
display.ie[1] = 0x41
display.ie[2] = 0x42
display.ie[3] = 0x43
display.ie[4] = 0x44
```

last_msg

The *last_msg* field should not be used when sending information; set to zero.

Return values

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

4.33 call_proceeding() - Send call proceeding message

This function may be used on an incoming call to send a message to the calling party to indicate that sufficient information has been obtained to proceed with the call. Depending on the protocol this message may already have been sent by the driver. If this is the case, using the function `call_proceeding()` will have no effect on the call. This function is only supported in Q.931 and ISUP protocols. This function is dependent on the signalling system and reference should be made to the appropriate specification for the protocol.

NOTE

Not supported for IP Telephony.

Synopsis

```
ACU_ERR call_proceeding(PROCEEDING_XPARMS *proceedingp);

typedef struct proceeding_xparms
{
  ACU_ULONG           size;              /* IN */
  ACU_CALL_HANDLE     handle;            /* IN */
  Union uniqueu       unique_xparms;     /* IN */
} PROCEEDING_XPARMS;

union uniqueu
{
  /* see protocol specific structures */
} unique_xparms;
```

See section 8 for further details on using unique_xparms.

Q931 protocol specific

```
    struct
    {
      struct
      {
        ACU_UCHAR          ie[MAXPROGRESS];
        ACU_UCHAR          last_msg;
      } progress_indicator;
      struct
      {
        ACU_UCHAR          ie[MAXDISPLAY];
        ACU_UCHAR          last_msg;
      } display;
      struct
      {
        ACU_UCHAR          ie[MAXNOTIFY];
        ACU_UCHAR               last_msg;
      } notify_indicator;
    } sig_q931;
```

ISUP/SS7 protocol specific

```
    struct
    {
      struct
      {
        ACU_UCHAR          ie[MAXPROGRESS];
        ACU_UCHAR          last_msg;
      } progress_indicator;
      ACU_UCHAR            in_band;
      ACU_UCHAR            charge_ind;
      ACU_UCHAR            dest_category;
      struct
      {
```

```
      ACU_UCHAR          valid;
      ACU_UCHAR          value;
    } isdn_access_ind;
    struct
    {
      ACU_UCHAR          valid;
      ACU_UCHAR          value;
    } isdn_userpart_ind;
    struct
    {
      ACU_UCHAR          valid;
      ACU_UCHAR          value;
    } interworking_ind;
  } sig_isup;
```

Input parameters

The `call_proceeding()` function takes a pointer, *`proceedingp`*, to a structure,`PROCEEDING_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`handle`
The *`handle`* field identifies the call that will send the call proceeding message.

`unique_xparms`
See unique_xparms Q931 for `sig_q931` definitions

See unique_xparms ISUP for `sig_isup` definitions

Return values

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

4.34 call_progress() - Send progress information

This function may be used to send call progress information to the network. This function may be used on an incoming call in the event of interworking or to indicate that in-band information is now available. This function is only supported in Q.931, ISUP, SIP and H.323 protocols. This information is dependent on the signalling system and reference should be made to the appropriate specification for the protocol.

NOTE

For the SIP and H323 protocols use of this routine enables early media announcements. It may therefore be necessary to set the codecs and media_Settings structure appropriately to ensure coherent audio.

Synopsis

```
ACU_ERR call_progress(PROGRESS_XPARMS *progressp);

typedef struct progress_xparms
{
  ACU_ULONG           size;              /* IN */
  ACU_CALL_HANDLE     handle;            /* IN */
  union uniqueu       unique_xparms;     /* IN */
} PROGRESS_XPARMS;

union uniqueu
{
  /* see protocol specific structures */
} unique_xparms;
```

See section 8 for further details on using unique_xparms.

Q931 specific protocol

```
    struct
    {
      struct
      {
        ACU_UCHAR          ie[MAXPROGRESS];
        ACU_UCHAR          last_msg;
      } progress_indicator;
      struct
      {
        ACU_UCHAR          ie[MAXDISPLAY];
        ACU_UCHAR          last_msg;
      } display;
      CAUSE      cause;
    } sig_q931;
```

ISUP/SS7 specific protocol

```
    struct
    {
      struct
      {
        ACU_UCHAR          ie[MAXPROGRESS];
        ACU_UCHAR          last_msg;
      } progress_indicator;
      ACU_UCHAR            in_band;
      ACU_UCHAR            charge_ind;
      ACU_UCHAR            dest_category;
      struct
      {
        ACU_UCHAR          valid;
        ACU_UCHAR          value;
      } isdn_access_ind;
```

```
      struct
      {
        ACU_UCHAR          valid;
        ACU_UCHAR          value;
      } isdn_userpart_ind;
      struct
      {
        ACU_UCHAR          valid;
        ACU_UCHAR          value;
      } interworking_ind;
    } sig_isup;
```

IP telephony (iptel) specific protocol

```
struct
{
  ACU_CHAR            destination_display_name;
  ACU_CODEC           codecs[MAXCODECS];
  MEDIA_SETTINGS      media_settings;
  union
  {
    struct
    {
      ACU_INT         send_reliable_provisional_response;
      ACU_CHAR        contact_address[MAXADDR];
    } sig_sip;
    struct
    {
      ACU_INT         h245_tunneling;
      ACU_INT         faststart;
      ACU_INT         early_h245;
      ACU_INT         location;
      ACU_INT         description;
    } sig_h323;
  } protocol_specific;
} sig_iptel;
```

Input parameters

The `call_progress()` function takes a pointer, *`progressp`*, to a structure,`PROGRESS_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`handle`
The *`handle`* field identifies the call that will send the progress message.

`unique_xparms`
See unique_xparms Q931 for `sig_q931` definitions

See unique_xparms ISUP for `sig_isup` definitions

See unique_xparms IP telephony for `sig_iptel` definitions

The following unique parameters are not detailed in *`unique_xparms`* and are specific to `progress_xparms`:

Unique parameters for SIP

`send_reliable_provisional_response`
By default, set to 0, 18* messages are sent unreliably. Setting this flag to 1 forces them to be sent reliably. ERR_PARM is returned if the caller does not support the protocol extension required for reliable provisional responses.

`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 maybe left blank. It will be in URI address format.

Unique parameters for H.323

location

The location field is part of the progress indicator information. Valid values are:

L_USER

Indicates that interworking has occurred directly at the user.

L_PRIV_NET_LOCAL_USER

Indicates that interworking has occurred at the private network serving the local user.

L_PUB_NET_LOCAL_USER

Indicates that interworking has occurred at the public network serving the local user.

L_TRANSIT_NET

Indicates that interworking has occurred at the transmit network.

L_PUB_NET_REMOTE_USER

Indicates that interworking has occurred at the public network service the remote user.

L_PRIV_NET_REMOTE_USER

Indicates that interworking has occurred at the private network serving the remote user.

L_BEYOND_IW_POINT

Indicates that interworking has occurred at the network beyond the interworking point.

description

The description field is part of the progress indicator information. Valid values are:

D_NOT_END_TO_END_ISDN
Indicates that the call is not end-to-end ISDN.

D_DEST_ADDR_NON_ISDN
Indicates that the destination address is non-ISDN.

D_ORIG_ADDR_NON_ISDN
Indicates that the origination address is non-ISDN.

D_CALL_RETURNED_TO_ISDN
Indicates that the call has returned to the ISDN.

D_IW_OCCURRED
Indicates that interworking has occurred.

D_INBAND_INFO_AVAIL
Indicates that inband information is available.

Return values

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

4.35 call_get_originating_addr() - Receiving the originating address

This function may be used to obtain the originating address of an incoming call in some Channel Associated Signalling (CAS) systems and ISUP variants where the application must explicitly request the originating address from the network.

This function may be used after an incoming call has been detected but before the call has been accepted. Under ISUP, this function may be used until either ACM or CON are transmitted.

NOTE

Not supported for IP Telephony.

Synopsis

```
ACU_ERR call_get_originating_addr(int handle);

ACU_ERR xcall_get_originating_addr(GET_ORIGINATING_ADDR_XPARMS*
originating_parms);

typedef struct get_originating_addr_xparms
{
  ACU_ULONG           size;              /* IN */
  ACU_CALL_HANDLE     handle;            /* IN */
} GET_ORIGINATING_ADDR_XPARMS;
```

Input parameters

The `xcall_get_originating_addr()` function takes a pointer *`originating_parms`*, to a structure,`GET_ORIGINATING_ADDR_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`handle`

The *`handle`* field identifies the call that should obtain the originating address.

Return values

Use of the function will result in the `EV_DETAILS` event from the driver when the originating address is available.

The originating address may be inspected by use of the `call_details()` function.

In some circumstances multiple `call_get_originating_addr()` function calls result in further information being requested from the network. This requirement will be documented in the release notes for the particular signalling system.

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

4.36 call_answercode() - Setting the answer code

Some protocols allow an incoming call to be answered with information about how the call is to be handled during the `EV_CALL_CONNECTED` state. This function can be used to pass the answer code to the driver for use during call connection. The answer code is primarily for CAS protocols, however, use of the function generally will not adversely affect those protocols (ISDN) that do not require the answer code.

NOTE

Not supported for IP Telephony.

Synopsis

```
ACU_ERR call_answercode(CAUSE_XPARMS *answerp);

typedef struct cause_xparms
{
  ACU_ULONG                size;              /* IN */
  ACU_CALL_HANDLE          handle;            /* IN */
  ACU_INT                  cause;             /* IN */
  ACU_INT                  raw;               /* IN */
} CAUSE_XPARMS;
```

Input parameters

The `call_answercode()` function takes a pointer, *answerp*, to a structure,`CAUSE_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

handle

The `handle` field is used to identify the call to which the answer code applies.

cause

The `cause` field must contain the answer code to be used during the connection of the incoming call and will be one of the standard set of answer codes described below.

raw

This `raw` field is not used and should be set to zero.

Return values

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

NOTE

It should be noted that call_answercode() simply 'lodges' the answer code with the device driver for subsequent signaling.

If the call_answercode() function is to be used then it must be invoked before use of either the call_incoming_ringing() or call_accept() functions.

Standard answer codes

The following describes the standard set of answer codes provided.

`AC_NORMAL`	default acceptance code
`AC_CHARGE`	answer call with charging
`AC_NOCHARGE`	answer call without charging
`AC_LAST_RELEASE`	last party release
`AC_SPARE1`	spare
`AC_SPARE2`	spare

The `AC_SPAREx` codes are provided for future expansion but in the R2T1 generic R2 protocol these codes may have specific values mapped to them by certain driver configuration switches. This is documented in the R2 application guide supplied with

the software.

4.37 call_get_charge()- Receiving call charge information

This function may be used to obtain information regarding the cost of a call. The function may be used any time that a valid call handle is available, however, the call charge information may not be valid until the call has cleared and the call has gone to the `EV_IDLE` state. The function provides for the receipt of call charging information and/or the accumulation of meter pulses.

NOTE

Not supported for IP Telephony.

Synopsis

```
ACU_ERR call_get_charge(GET_CHARGE_XPARMS *chargep);

typedef struct get_charge_xparms
{
  ACU_ULONG           size;                  /* IN */
  ACU_CALL_HANDLE     handle;                /* IN */
  ACU_INT             type;                  /* OUT */
  char                charge[CHARGEMAX];     /* OUT */
  ACU_UINT            meter;                 /* OUT */
  union
  {
    struct
    {
      ACU_UINT            tariff_type;       /* OUT */
    } sig_isup;
  } unique_xparms;
} GET_CHARGE_XPARMS;
```

Input parameters

The `call_get_charge()` function takes a pointer, *`chargep`*, to a structure,`GET_CHARGE_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`handle`

The *`handle`* field is used to identify the call on which the call charging information is requested.

Return values

`type`

The *`type`* field will contain the type of charging information available and will have one of the following values:

- `CHARGE_NONE`
  There is no valid charging information available in either the charge or meter fields.
- `CHARGE_INFO`
  The information contained in the element charge is valid and may be used.
- `CHARGE_METER`
  The information contained within the meter element is valid and may be used.

`charge`

The charging information of the call if available from the network. This information is presented as received from the network and interpretation of the information is left to the application.

`meter`

The number of meter pulses received from the network. This information is presented as received from the network and the accuracy and timing of the pulse count is dependent upon the network.

tariff_type
An integer specific to Finland, (Finnish ISUP). Set to zero when not in use.

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

NOTE

If INFO and METER charge information are both present, charging information will be made available to the application via CHARGE_METER.

4.38 call_put_charge()- Sending call charge information

This function may be used to send call charging information on the network and may be used any time that a valid call handle is available and the call is in the `EV_CALL_CONNECTED` state. It should be noted that it is normally only possible to send this information from a Network end protocol.

The function provides for sending of call charge information and/or meter pulses. The choice of information is dependent upon the type of signalling system supported by the device driver.

For signalling systems that support the sending of call charge information, the function may be used to pass a null terminated string of IA5 characters of charging information to the driver for transmission on the network. The driver passes the message 'as is' with no interpretation or modification of the character string provided.

For signalling systems that use meter pulses, use of the function will result in one meter pulse being sent on the network, the charging information string being ignored.

NOTE

Not supported for IP Telephony.

Synopsis

```
ACU_ERR call_put_charge(PUT_CHARGE_XPARMS *chargep);

typedef struct put_charge_xparms
{
  ACU_ULONG           size;                 /* IN */
  ACU_CALL_HANDLE     handle;               /* IN */
  char                charge[CHARGEMAX];    /* IN */
  ACU_UINT            meter;                /* IN */
  union
  {
    struct
    {
      ACU_UINT        tariff_type;          /* IN */
    } sig_isup;
  } unique_xparms;
} PUT_CHARGE_XPARMS;
```

Input parameters

The `call_put_charge()` function takes a pointer, *`chargep`*, to a structure,`PUT_CHARGE_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`handle`
The *`handle`* field is used to identify the call on which the call charging information will be sent.

`charge`
The charging information of the call if available from the network. This information is presented as received from the network and interpretation of the information is left to the application.

`meter`
The number of meter pulses received from the network. This information is presented as received from the network and the accuracy and timing of the pulse count is dependent upon the network.

`tariff_type`
An integer specific to Finland, (Finnish ISUP). Set to zero when not in use.

Return values

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

4.39 call_notify() - Send notification information

This function may be used on a call to send a message to the network to indicate an appropriate call related event during the active state of a call (such as user suspended). This function is supported in some Q.931 protocols. This function is dependent on the signalling system and reference should be made to the appropriate specification for the protocol.

NOTE

Not supported for IP Telephony.

Synopsis

```
ACU_ERR call_notify(NOTIFY_XPARMS *notifyp);

typedef struct notify_xparms
{
  ACU_ULONG           size;              /* IN */
  ACU_CALL_HANDLE     handle;            /* IN */
  union
  {
    struct
    {
      struct
      {
        ACU_UCHAR          ie[MAXNOTIFY];
        ACU_UCHAR          last_msg;
      } notify_indicator;                /* IN */
      struct
      {
        ACU_UCHAR          ie[MAXDISPLAY];
        ACU_UCHAR          last_msg;
      } display;                         /* IN */
    } sig_q931;
  } unique_xparms;
} NOTIFY_XPARMS;
```

Input parameters

The `call_notify()` function takes a pointer, *`notifyp`*, to a structure,`NOTIFY_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`handle`

The *`handle`* field identifies the call that will send the notify message.

unique_xparms system-specific fields -Q931 structure

`notify_indicator`

The *`notify_indicator`* field can be used to indicate the information detailing the call related event pertaining to the call. The `ie` field of the *`notify_indicator`* structure should contain the notify information that is sent transparently to the other end. The *`last_msg`* field should be set to zero. This information is dependent on the signalling system and reference should be made to the appropriate specification for the protocol.

`display`

The *`display`* field is used to transmit information that may be displayed by the user. (`MAXDISPLAY = 34`)

Return values

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

4.40 call_send_keypad_info() - Send Keypad Information

This function may be used to send keypad information during a call. This function is only supported in Q.931 and H.323 protocols.

NOTE

Sending of keypad information is only possible in the connected state.

Synopsis

```
ACU_ERR call_send_keypad_info(KEYPAD_XPARMS *keypadp);

typedef struct keypad_xparms
{
  ACU_ULONG           size;              /* IN */
  ACU_CALL_HANDLE     handle;            /* IN */
  ACU_PORT_ID         net;               /* IN */
  union
  {
    struct
    {
      ACU_INT         device;            /* IN */
      KEYPAD          keypad;            /* IN */
      ACU_INT         location;          /* IN */
      DISPLAY         display;           /* IN */
    } sig_q931;
    struct
    {
      union
      {
        struct
        {
          ACU_CHAR    dtmf[MAXNUM];      /* IN */
        } sig_h323;
      } protocol_specific;
    } sig_iptel;
  } unique_xparms;
} KEYPAD_XPARMS;
```

Input parameters

The `call_send_keypad_info()` function takes a pointer, *`keypadp`*, to a structure,`KEYPAD_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`handle`
The *`handle`* field identifies the call that will send the keypad information.

`net`
Reserved for future expansion of the driver.

unique_xparms protocol specific fields

Q931 structure

`device`
Reserved for future expansion of the driver.

`keypad`
The *`keypad`* field is used to transmit any keypad information (such as supplementary service information).

`location`
The *`location`* field must contain the following value. Any other values are reserved for future expansion of the driver.

`KEYPAD_CONNECT`

display
The *display* field can be used to transmit information that may be displayed by the user.

iptel structure

dtmf
Holds the DTMF information that is to be transmitted. For H.323, the DTMF mode is out of band and the information will be sent out on the signalling path using User Input Indications.

Return values

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

4.41 call_send_connectionless() - Call independent signalling

Some supplementary services require the use of a connectionless network service to transmit `FACILITY` messages. `call_send_connectionless()` allows these messages to be transmitted. This message is not associated with a call and no handle is associated with it.

Synopsis

```
ACU_ERR call_send_connectionless(FEATURE_DETAIL_XPARMS* feature_detailsp);

typedef struct feature_detail_xparms
{
  ACU_ULONG           size;              /* IN */
  ACU_CALL_HANDLE     handle;            /* IN */
  ACU_PORT_ID         net;               /* IN */
  ACU_ULONG           feature_type;      /* IN */
  union feature_union feature;           /* IN */
} FEATURE_DETAIL_XPARMS;

union feature_union
{
  struct uui_xparms                 uui;
  struct facility_xparms            facility;
  struct diversion_xparms           diversion;
  struct feature_hold_xparms        hold;
  struct feature_transfer_xparms    transfer;
  struct raw_data_struct            raw_data;
  struct mlpp_xparms                mlpp;
  struct non_standard_data_xparms   non_standard
};
```

Structure for (Raw) Facility Information

```
struct facility_xparms
{
  ACU_INT        command;
  ACU_UCHAR      control;
  ACU_UCHAR      length;
  ACU_UCHAR      data[MAXFACILITY_INFO];
  char           destination_addr[MAXNUM];
  char           originating_addr[MAXNUM];
  ACU_UCHAR      dest_subaddr[MAXNUM];
  ACU_UCHAR      dest_numbering_type;
  ACU_UCHAR      dest_numbering_plan;
  ACU_UCHAR      orig_numbering_type;
  ACU_UCHAR      orig_numbering_plan;
  ACU_UCHAR      orig_numbering_presentation;
  ACU_UCHAR      orig_numbering_screening;
};
```

See section 10.2 for further details on facility xparm definitions

Input parameters

The `call_send_connectionless()` function takes a pointer, *`feature_detailsp`*, to a structure,`FEATURE_DETAIL_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`Handle`
This field is not used in this API call.

`net`
The *`net`* field must be set to the *`port_id`* that will be used to transmit the connectionless information.

feature_type
The *feature_type* field should be used to indicate the type of feature. The only suitable value this field should take is FEATURE_FACILITY.

To include facility information the *feature_type* field should be supplied with the value FEATURE_FACILITY.

Facility Information – facility_xparms

command
EuroISDN allows the possibility of two types of connectionless transmission. The command field selects the mode for this call. This field must contain one of these values

```
FAC_CLESS_DL_DATA_CMD          /* ETS300 196 Section8.3.2.2 */
FAC_CLESS_DL_UNIT_DATA_CMD     /* ETS300 196 Section8.3.2.4 */
```

This value is ignored when using QSIG.

facility_xparms

The *facility_xparms* structure must be used in the following way:

data
The *data* field should be supplied with protocol dependant information.

length
The length of this information should be supplied in the length field.

It may be appropriate to transmit addressing information in this message.

*_addr

The *destination_addr*, *originating_addr* and *called_subaddr* fields may be used where appropriate for the protocol. If these fields are used then the following may also be used as required:

```
dest_numbering_type
dest_numbering_plan
orig_numbering_type
orig_numbering_plan
orig_numbering_presentation
orig_numbering_screening
```

All other fields are not applicable to call_send_connectionless().

Return values

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

NOTE

A successful invocation of call_send_connectionless() is no guarantee that the network side will receive the message. If an error occurs then the FACILITY message may be discarded.

4.42 call_get_connectionless() - Call independent signalling

Some supplementary services require the use of a connectionless network service to transmit `FACILITY` messages. `call_get_connectionless()` will retrieve the latest message of this type.

Synopsis

```
ACU_ERR call_get_connectionless (FEATURE_DETAIL_XPARMS* feature_detailsp);

typedef struct feature_detail_xparms
{
  ACU_ULONG           size;              /* IN */
  ACU_CALL_HANDLE     handle;            /* IN */
  ACU_PORT_ID         net;               /* IN */
  ACU_ULONG           feature_type;      /* IN */
  union feature_union feature;           /* IN */
} FEATURE_DETAIL_XPARMS;

union feature_union
{
  struct facility_xparms       facility;
};
```

Structure for (Raw) Facility Information

```
struct facility_xparms
{
  ACU_INT        command;
  ACU_UCHAR      control;
  ACU_UCHAR      length;                      /* OUT */
  ACU_UCHAR      data[MAXFACILITY_INFO];      /* OUT */
  char           destination_addr[MAXNUM];
  char           originating_addr[MAXNUM];
  ACU_UCHAR      dest_subaddr[MAXNUM];
  ACU_UCHAR      dest_numbering_type;
  ACU_UCHAR      dest_numbering_plan;
  ACU_UCHAR      orig_numbering_type;
  ACU_UCHAR      orig_numbering_plan;
  ACU_UCHAR      orig_numbering_presentation;
  ACU_UCHAR      orig_numbering_screening;
};
```

See section 10.2 for further details on facility xparm definitions

Input parameters

The `call_get_connectionless()` function takes a pointer, *`feature_detailsp`*, to a structure,`FEATURE_DETAIL_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`Handle`
This field is not used in this API call.

`net`
The *`net`* field must be set to the `port id` that will be used to retrieve any connectionless information that has arrived for that port.

`feature_type`
The *`feature_type`* field should be used to indicate the type of feature. The only suitable value this field should take is `FEATURE_FACILITY`

Return values

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

Facility Information – facility_xparms

If connectionless data was available, the driver will fill the *facility_xparms* structure with the following values.

length

The *length* field will contain the length of the information supplied in the data field.

data

The *data* field will contain protocol dependant information.

If any address information was supplied then the following fields may contain data:

```
destination_addr
originating_addr
called_subaddr
dest_numbering_type
dest_numbering_plan
orig_numbering_type
orig_numbering_plan
orig_numbering_presentation
orig_numbering_screening
```

NOTE

Connectionless data is not queued for all protocols and may be overwritten if it is not retrieved by an application before the next message arrives. Rather than poll this function, applications can wait for a call notification event to signal that there is a message to collect. See call_get_port_notification().

4.43 call_enable_connectionless() - Call independent signalling

Some protocols require that actions be taken for messages that are not understood. In order to provide standard-conforming default behaviour the Aculab API requires that applications explicitly enable the reception of these event types. `call_enable_connectionless()` allows the application to control notification of these messages.

Synopsis

```
ACU_ERR call_enable_connectionless (FEATURE_ENABLE_XPARMS *enablep);

typedef struct feature_enable_xparms
{
  ACU_ULONG           size;              /* IN */
  ACU_CALL_HANDLE     handle;            /* IN */
  ACU_PORT_ID         net;               /* IN */
  ACU_ULONG           feature_type;      /* IN */
  ACU_INT             enable;            /* IN */
} FEATURE_DETAIL_XPARMS;
```

Input parameters

The `call_enable_connectionless()` function takes a pointer *`enablep`* to a structure,`FEATURE_ENABLE_XPARMS`. The structure must be initialised before invoking the function (see section 2.2).

`handle`
This field is not used in this API call.

`net`
The *`net`* field must be set to the `port id` that will be used to retrieve any connectionless information that has arrived for that port.

`feature_type`
The *`feature_type`* field should be used to indicate the type of feature. The currently supported types are `FEATURE_NSM_RAS`, `FEATURE_XRS` and `FEATURE_CONNECTIONLESS_FACILITY`.

`enable`
If set to a non-zero value messages of the specified `feature_type` will be delivered to the application. In order to ensure standards conformance when `FEATURE_NSM_RAS` messages are delivered to the application, the application must ensure that it sends `FEATURE_XRS` messages in response to any `FEATURE_NSM_RAS` messages that it does not understand.

Return values

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

4.44 call_maint_port_block()/call_maint_port_unblock() - block or unblock timeslots

The block function is used to block a group of timeslots within a port, preventing these timeslots from being used for subsequent calls. Conversely, unblock is used on a blocked port to unblock a group of timeslots for that port, bringing the timeslots back into service.

Synopsis

```
ACU_ERR call_maint_port_block (PORT_BLOCKING_XPARMS *blockp);
ACU_ERR call_maint_port_unblock (PORT_BLOCKING_XPARMS *unblockp);

typedef struct port_blocking_xparms
{
 ACU_ULONG            size;              /* IN */
 ACU_PORT_ID          net;               /* IN */
 ACU_INT              flags;             /* IN */
 ACU_INT              type;              /* IN */
 ACU_INT              reserved;          /* IN */
 union
 {
   ACU_ULONG          ts_mask;           /* IN */
 }          unique_xparms;
} PORT_BLOCKING_XPARMS;
```

NOTE

These calls currently support QSIG/SS7/ISUP/ETS300/NI2/AT&T T1 and DMS-100 only.

Input parameters

The functions take a pointer, *blockp* or *unblockp*, to a structure, PORT_BLOCKING_XPARMS. The structure must be initialised before invoking the function, (see section 2.2).

net

Identifies the network port id of the affected circuits.

flags(SS7/ISUP only)

If set to ACUC_MAINT_SYNC, allows the command to operate synchronously - i.e. the calling thread will sleep within the driver until the remote SP has acknowledged the request.

If set to ACUC_MAINT_EVENT then a port event of type ACU_CALL_EVT_BLOCKING_STATE_CHANGE will be generated when the request completes.

flags (ETS300 & QSIG)

When set to ACU_MAINT_BLOCK_B_CHAN, this allows a particular timeslot to be configured for B-Channel blocking (when called with the call_maint_port_block() function) or B-Channel unblocking (when called with the call_maint_port_unblock() function). If a timeslot is configured for B-Channel blocking while a call is present, the blocking will take effect after that call is cleared.

flags (ETS300 only)

When set to ACU_MAINT_ETS_D_CHAN, will result in the establishment of the D channel if call_maint_port_unblock() is used, or the disconnection of the D channel if call_maint_port_block() is used. Blocking the D channel will return an error ERR_PORT_BLOCKED should you attempt to generate a protocol message on the D channel.

flags (NI2/AT&T T1/DMS-100 only)
When set to `ACU_MAINT_SERVICE_BLOCKING`, the selection of timeslots defined in the `ts_mask` field will be blocked/unblocked using service messages. After timeslots have been blocked in this manner, no outgoing calls should be made through the API on those timeslots. Incoming calls for the blocked timeslots will be rejected until unblocked

type
May be set to `ISUP_HW_BLOCK` or `ISUP_MAINTENANCE_BLOCK` to identify whether the blocking is because of maintenance operations or because of hardware failure.

NOTE

Type has no effect with the NI2, AT&T T1, or DMS-100 signalling systems.

NOTE

With the NI2, AT&T T1, or DMS-100 signalling systems it is possible that the network may change the status of the port. In that case there is currently no event /notification to the application.

reserved
Currently not used; for future enhancements.

ts_mask
Is a 32 bit mask where bit 0 represents timeslot 0 etc.

Return values

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

4.45 call_maint_port_reset() - reset and clear timeslots

This function is used to reset the status of a group of timeslots on a port. This will clear any calls in progress on the timeslots defined.

Synopsis

```
ACU_ERR call_maint_port_reset (PORT_RESET_XPARMS *resetp);

typedef struct port_reset_xparms
{
  ACU_ULONG           size;              /* IN */
  ACU_PORT_ID         net;               /* IN */
  ACU_INT             flags;             /* IN */
  ACU_INT             reserved;          /* IN */
  union
  {
    ACU_ULONG             ts_mask;          /* IN */
  }     unique_xparms;
} PORT_RESET_XPARMS;
```

NOTE

This call currently supports SS7/ISUP only.

Input parameters

The `call_maint_port_reset()` function takes a pointer *`resetp`*, to a structure,`PORT_RESET_PARMS`. The structure must be initialised before invoking the function, (see section 2.2) .

`net`
identifies the network port id containing the affected circuits.

`flags`
if set to `ACUC_MAINT_SYNC`, allows the command to operate synchronously - i.e. the calling thread will sleep within the driver until the remote SP has acknowledged the request.

If set to `ACUC_MAINT_EVENT` then a port event of type `ACU_CALL_EVT_RESET_STATE_CHANGE` will be generated when the request completes.

`reserved`
is reserved for possible future enhancements.

`ts_mask`
is a 32 bit mask where bit 0 represents timeslot 0 etc.

NOTE

For SS7 multiple reset messages will be sent if the requested circuits are not contiguous.

Return values

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

4.46 call_maint_ts_block()/call_maint_ts_unblock() - block or unblock a timeslot

The block function is used to block (take out of service) a specific timeslot, preventing it from being used to set up a call. Conversely, unblock is used on a blocked timeslot to unblock the timeslot and bring it back into service.

Synopsis

```
ACU_ERR call_maint_ts_block (TS_BLOCKING_XPARMS *blockp);
ACU_ERR call_maint_ts_unblock (TS_BLOCKING_XPARMS *unblockp);

typedef struct ts_blocking_xparms
{
 ACU_ULONG       size;              /* IN */
 ACU_PORT_ID     net;               /* IN */
 ACU_INT         ts;                /* IN */
 ACU_INT         flags;             /* IN */
} TS_BLOCKING_XPARMS;
```

NOTE

These calls currently support SS7/ISUP, ETS300, QSIG, NI-2, AT&T T1 and DMS-100.

Input parameters

The functions takes a pointer, *blockp* or *unblockp*, to a structure, TS_BLOCKING_XPARMS. The structure must be initialised before invoking the function, (see section 2.2) .

net
net field defines the network port id.

ts
Is the timeslot that is to be blocked/un-blocked.

flags *(ISUP)*
If set to ACUC_MAINT_SYNC, allows the command to operate synchronously - i.e. the calling thread will sleep within the driver until the remote SP has acknowledged the request.

If set to ACUC_MAINT_EVENT then a port event of type ACU_CALL_EVT_BLOCKING_STATE_CHANGE will be generated when the request completes.

flags (ETS300 & QSIG)
When set to ACU_MAINT_BLOCK_B_CHAN, this allows a particular timeslot to be configured for B-Channel blocking (when called with the call_maint_ts_block() function) or B-Channel unblocking (when called with the call_maint_ts_unblock() function). If a timeslot is configured for B-Channel blocking while a call is present, the blocking will take effect after that call is cleared.

flags (ETS300 only)
When set to ACU_MAINT_ETS_D_CHAN, will result in the establishment of the D channel if call_maint_port_unblock() is used, or the disconnection of the D channel if call_maint_port_block() is used. Blocking the D channel will return an error ERR_PORT_BLOCKED should you attempt to generate a protocol message on the D channel.

flags (NI-2/AT&T T1/DMS-100)
When set to ACU_MAINT_SERVICE_BLOCKING, a service message will be sent to block or unblock the timeslot indicated in the *ts* field. After a timeslot has been blocked in this manner no outgoing calls should be made through the API on that timeslot. Incoming calls for the particular timeslot will be rejected.

NOTE

With the NI-2, AT&T T1, or DMS-100 signalling systems it is possible that the network may change the status of the port. In that case there is currently no event /notification to the application.

Return values

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

4.47 call_maint_port_status() – obtain per-timeslot status

This function is used to obtain protocol specific per-timeslot status information.

Synopsis

```
ACU_ERR call_maint_port_status(PORT_STATUS_XPARMS *st_parms);

typedef struct port_status_xparms {
    ACU_ULONG     size;         /* IN */
    ACU_PORT_ID   net;          /* IN */
    ACU_INT       rqst;         /* IN */
    ACU_INT       flags;        /* IN */
    ACU_ULONG     ts_mask;      /* IN */
    ACU_UCHAR     ts_info[32]; /* OUT */
} PORT_STATUS_XPARMS;
```

Input parameters

The functions takes a pointer, *st_parms*, to a structure, TS_STATUS_XPARMS. The structure must be initialised before invoking the function, (see section 2.2) .

net
net field defines the network port id.

rqst
Defines the status information required. Protocol specific, see tables below.

flags
None currently defined, set to zero.

ts_mask
A 32bit mask identifying the timeslots from which information is required. May be updated to indicate the timeslots from which information is valid.

Return values

ts_info
One byte per timeslot containing the requested state information.

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

ISUP requests:

rqst	ts_info
ACU_MAINT_PORT_GET_BLOCK_STATE	One or more of: ACU_MAINT_PORT_BLOCKED_LOC_MAINT ACU_MAINT_PORT_BLOCKED_LOC_HW ACU_MAINT_PORT_BLOCKED_REM_MAINT ACU_MAINT_PORT_BLOCKED_REM_HW ACU_MAINT_PORT_BLOCKED_UCIC ACU_MAINT_PORT_BLOCKED_BEFORE_RESET ACU_MAINT_PORT_UNBLOCK_IN_PROGRESS ACU_MAINT_PORT_BLOCK_IN_PROGRESS
ACU_MAINT_PORT_GET_RESET_STATE	One or more of: ACU_MAINT_PORT_RESET_INWARDS ACU_MAINT_PORT_RESET_IN_PROGRESS
ACU_MAINT_PORT_GET_APPLICATION_REF	Per SS7 destination value settable in the ss7.cfg file, defaults to 0 for ITU and 1 for ANSI.
ACU_MAINT_PORT_GET_CIC_0_7 and ACU_MAINY_PORT_GET_CIC_8_15	Low and high bytes of the ISUP CIC for the timeslot.

ACU_MAINT_PORT_GET_LPC_0_7 ACU_MAINT_PORT_GET_LPC_8_15 and ACU_MAINY_PORT_GET_LPC_16_23	SS7 local pointcode for the timeslot.
ACU_MAINT_PORT_GET_RPC_0_7 ACU_MAINT_PORT_GET_RPC_8_15 and ACU_MAINY_PORT_GET_RPC_16_23	SS7 remote pointcode for the timeslot.

ETS300 requests:

rqst	ts_info
ACU_MAINT_PORT_GET_SERVICE_STATE	None, a SERVICE_REQ message is sent for each selected timeslot.

4.48 call_set_handle_event_queue() - associate a handle with an event queue

This function is used to associate a call handle with an event queue. All call events that occur for the specified call handle will be notified via this event queue.

NOTE

This function can be called at any time – any events pending for the handle will be transferred from the old queue to the new queue.

Synopsis

```
ACU_ERR call_set_handle_event_queue(ACU_QUEUE_PARMS* queue_parms);

typedef struct _ACU_QUEUE_PARMS
{
  ACU_ULONG                size;              /* IN */
  ACU_RESOURCE_ID          resource_id;       /* IN */
  ACU_EVENT_QUEUE          queue_id;          /* IN */
} ACU_QUEUE_PARMS;
```

Input Parameters

The `call_set_handle_event_queue()` function takes a pointer, *`queue_parms`*, to a structure, `ACU_QUEUE_PARMS`. The structure must be initialised before invoking the function, (see section 2.2) .

`resource_id`

The *`resource_id`* field must be set to a valid call handle.

`queue_id`

The *`queue_id`* field must be set to a valid queue as returned by `acu_allocate_event_queue()` when creating a queue.

Return Values

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

4.49 call_get_handle_event_wait_object() - get a wait object for a handle

This function is used to get a wait object that is associated with a specific call handle. This wait object will be signalled while there are call events pending for that call handle. The wait object returned by this function can be used with operating system specific wait functions such as `WaitForMultipleObjects()` or `poll()`.

Synopsis

```
ACU_ERR  call_get_handle_event_wait_object(CALL_HANDLE_WAIT_OBJECT_PARMS*
wo_parms);

typedef struct _CALL_HANDLE_WAIT_OBJECT_PARMS
{
  ACU_ULONG                size;              /* IN */
  ACU_CALL_HANDLE          handle;            /* IN */
  ACU_WAIT_OBJECT          wait_object;       /* OUT */
} CALL_HANDLE_WAIT_OBJECT_PARMS;
```

Input Parameters

The `call_get_handle_event_wait_object()` function takes a pointer *`wo_parms`*, to a structure,`CALL_HANDLE_WAIT_OBJECT_PARMS`. The structure must be initialised before invoking the function, (see section 2.2) .

`handle`

`handle` should be set to the appropriate call handle.

Return Values

`wait_object`

`wait_object` will be set to a valid operating system specific wait object associated with the specified call handle.

NOTE

The wait object returned by this function will be valid while the application keeps the associated call handle open. It will become invalid when call_release() is called for that handle.

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

4.50 call_set_handle_app_context_token() - associate data with a handle

This function is used to associate application-defined data with a call handle. This data is returned as the *context* field by `acu_get_event_from_queue()`.

The token assigned using this function can also be retrieved using `call_get_handle_app_ context_token()`.

Synopsis

```
ACU_ERR call_set_handle_app_context_token(ACU_APP_CONTEXT_TOKEN_PARMS*
token_parms);

typedef struct _ACU_APP_CONTEXT_TOKEN_PARMS
{
  ACU_ULONG               size;              /* IN */
  ACU_RESOURCE_ID         resource_id;       /* IN */
  ACU_ACT                 app_context_oken;  /* IN */
} ACU_APP_CONTEXT_TOKEN_PARMS;
```

Input Parameters

The `call_set_handle_app_context_token()` function takes a pointer *`token_parms`*, to a structure,`ACU_APP_CONTEXT_TOKEN_PARMS`. The structure must be initialised before invoking the function, (see section 2.2) .

`resource_id`

The *`resource_id`* field must be initialised to a valid call handle (returned by `call_openout()`, `call_openin()` etc.).

`app_context_token`

The *`token`* field should be set to the data you want to associate with the call.

Return Values

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

4.51 call_get_handle_app_context_token() - receive data for a handle

This function is used to retrieve application defined data that was associated with a call handle by an earlier call to `call_openin()`, `call_openout()`, or `call_set_handle_app_context_ token()`.

Synopsis

```
ACU_ERR call_get_handle_app_context_token(ACU_APP_CONTEXT_TOKEN_PARMS*
token_parms);

typedef struct _ACU_APP_CONTEXT_TOKEN_PARMS
{
  ACU_ULONG                size;              /* IN */
  ACU_RESOURCE_ID          resource_id;       /* IN */
  ACU_ACT                  app_context_token; /* OUT */
} ACU_APP_CONTEXT_TOKEN_PARMS;
```

Input Parameters

The `call_get_handle_app_context_token()` function takes a pointer *`token_parms`*, to a structure,`ACU_APP_CONTEXT_TOKEN_PARMS`. The structure must be initialised before invoking the function, (see section 2.2) .

`resource_id`
The *`resource_id`* field must be initialised to a valid call handle (returned by `call_openout()`, `call_openin()` etc.)

Return Values

`app_context_token`
The *`token`* field will be set to the current application-defined data associated with the call.

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

4.52 call_set_port_default_handle_event_queue() - set the default call event queue for a port

Each port in the system is associated with a default call event queue for calls made on the port. This is initially set to the global call event queue but applications can associate a port with a new default call event queue by calling this function.

NOTE

This function does not change the event queue for calls made on the port prior to this function being called. Calls made before this function is called will be associated with the previous queue associated with the port (by default the global call event queue is associated with every port).

Synopsis

```
ACU_ERR call_set_port_default_handle_event_queue(ACU_QUEUE_PARMS*
queue_parms);

typedef struct _ACU_QUEUE_PARMS
{
  ACU_ULONG                size;              /* IN */
  ACU_RESOURCE_ID          resource_id;       /* IN */
  ACU_EVENT_QUEUE          queue_id;          /* IN */
} ACU_QUEUE_PARMS;
```

Input Parameters

The `call_set_port_default_handle_event_queue()` function takes a pointer *`queue_parms`*, to a structure,`ACU_QUEUE_PARMS`. The structure must be initialised before invoking the function, (see section 2.2) .

`resource_id`

The *`resource_id`* field must be set to a valid port id.

`queue_id`

The *`queue_id`* field must be set to a valid queue as returned by `acu_allocate_event_queue()` when creating a queue.

Return Values

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

4.53 call_set_port_app_context_token() – associate data with a port

This function is used to associate application-defined data with a port. This data is returned as the *context* field by `acu_get_event_from_queue()`.

The token assigned using this function can also be retrieved using `call_get_port_app_context_token()`.

Synopsis

```
ACU_ERR call_set_port_app_context_token(ACU_APP_CONTEXT_TOKEN_PARMS*
token_parms);

typedef struct _ACU_APP_CONTEXT_TOKEN_PARMS
{
  ACU_ULONG                    size;              /* IN */
  ACU_RESOURCE_ID              resource_id;       /* IN */
  ACU_ACT                      app_context_token; /* IN */
} ACU_APP_CONTEXT_TOKEN_PARMS;
```

Input Parameters

The `call_set_port_app_context_token()` function takes a pointer *token_parms*, to a structure,`ACU_APP_CONTEXT_TOKEN_PARMS`. The structure must be initialised before invoking the function, (see section 2.2).

resource_id

The *resource_id* field must be initialised to a valid port id (returned by `call_open_port()`).

app_context_token

The *token* field should be set to the data you want to associate with the port.

Return Values

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

4.54 call_get_port_app_context_token() - retrieve data for a port

This function is used to retrieve application-defined data that is associated with a port. The data can be set using `call_set_port_app_context_token()`.

Synopsis

```
ACU_ERR call_get_port_app_context_token(ACU_APP_CONTEXT_TOKEN_PARMS*
token_parms);

typedef struct tACU_APP_CONTEXT_TOKEN_PARMS
{
  ACU_ULONG                    size;              /* IN */
  ACU_RESOURCE_ID              resource_id;       /* IN */
  ACU_ACT                      app_context_token; /* OUT */
} ACU_APP_CONTEXT_TOKEN_PARMS;
```

Input Parameters

The `call_get_port_app_context_token()` function takes a pointer *`token_parms`*, to a structure,`ACU_APP_CONTEXT_TOKEN_PARMS`. The structure must be initialised before invoking the function, (see section 2.2) .

`resource_id`
The *`resource_id`* field must be initialised to a valid port id (returned by `call_open_port()`).

Return Values

`app_context_token`
The *`token`* field will be set to the associated data.

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

4.55 call_set_port_notification_queue() - set a queue for a port

This function is used to associate a port with a queue. All port notification events for this port will be notified via this event queue.

NOTE

This function can be called at any time – any notification events pending for the port will be transferred from the old queue to the new queue.

Synopsis

```
ACU_ERR call_set_port_notification_queue(ACU_QUEUE_PARMS* queue_parms);

typedef struct _ACU_QUEUE_PARMS
{
  ACU_ULONG                size;              /* IN */
  ACU_RESOURCE_ID          resource_id;       /* IN */
  ACU_EVENT_QUEUE          queue_id;          /* IN */
} ACU_QUEUE_PARMS;
```

Input Parameters

The `call_set_port_notification_queue()` function takes a pointer *`queue_parms`*, to a structure,`ACU_QUEUE_PARMS`. The structure must be initialised before invoking the function, (see section 2.2) .

`resource_id`

The *`resource_id`* field must be set to a valid port id.

`queue_id`

The *`queue_id`* field must be set to a valid queue as returned by `acu_allocate_event_queue()` when creating a queue.

Return Values

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

4.56 call_get_port_notification() - retrieve events for a port

The call driver queues a number of events that are not associated with a particular call. This function retrieves events that are associated with a particular port.

Synopsis

```
ACU_ERR call_get_port_notification(CALL_PORT_NOTIFICATION_PARMS* eventp);

typedef struct call_port_notification_parms
{
  ACU_ULONG               size;              /* IN */
  ACU_PORT_ID             port_id;           /* IN */
  ACU_INT                 event;             /* OUT */
} CALL_PORT_NOTIFICATION_PARMS;
```

Input Parameters

The `call_get_port_notification()` function takes a pointer *`eventp`*, to a structure,`CALL_PORT_NOTIFICATION_PARMS`. The structure must be initialised before invoking the function, (see section 2.2) .

`port_id`

The *`port_id`* field must be set to a valid port id (returned from a call to `call_open_port()`);

Return Values

`event`

The *`event`* field is set to one of the following values:

#define	Description
ACU_CALL_EVT_NO_EVENT	There are no events in the queue
ACU_CALL_EVT_L1_CHANGE	Layer 1 has changed on the specified port – use `call_l1_stats()` to determine what the change is
ACU_CALL_EVT_L2_CHANGE	Layer 2 has changed on the specified port – use `call_l2_state()` to determine what the change is
ACU_CALL_EVT_CONNECTIONLESS	A connectionless message has arrived on the specified port. Use `call_get_connectionless()` to retrieve it.
ACU_CALL_EVT_FIRMWARE_CHANGE	The firmware on a port has changed
ACU_CALL_EVT_HW_CLOCK_STOP	This message is sent in the unlikely event that the firmware has crashed on a port.
ACU_CALL_EVT_NO_CHANNEL_AVAILABLE	The firmware rejected an incoming call because the driver could not process it.
ACU_CALL_EVT_BLOCKING_STATE_CHANGE	The Service state has changed on at least one channel. Use `call_l2_state()` to determine which timeslots have been blocked or unblocked.
ACU_CALL_EVT_PORT_COMMS_LOST	The connection with the firmware on this port has been lost. This usually indicates a network problem but may also be triggered when the firmware has stopped or crashed. Firmware will need to be re-downloaded to the port in order to use the

#define	Description
	port again.
ACU_SIP_EV_RESPONSE	See The Extended SIP API Guide
ACU_SIP_EV_REQUEST	See The Extended SIP API Guide
ACU_SIP_EV_REQUEST_TIMEOUT	See The Extended SIP API Guide
ACU_CALL_EVT_BLOCKING_STATE_CHANGE	Event generated to indicate a request is completed if ACUC_MAINT_EVENT set when making a call to call_maint_port_block/unblock or call_maint_ts_block/unblock.
ACU_CALL_EVT_RESET_STATE_CHANGE	Event generated to indicate a request is completed if ACUC_MAINT_EVENT set when making a call to call_maint_port_reset.

NOTE

These notifications are an indication that something has changed. Upon receipt of one of these events, the application will need to make further API calls to determine what has changed.

NOTE

These notifications are queued. It may be possible that when an application retrieves an event, the state change it is describing has been superseded by another state change. Applications should be designed to cope with this. (i.e. don't assume that because a Layer 2 state change notification has been received, Layer 2 has gone down).

NOTE

ACU_CALL_EVT_L1_CHANGE and ACU_CALL_EVT_L2_CHANGE are not automatically sent when firmware is downloaded. They will be sent if the firmware encounters an error condition immediately after download. The driver will also send this event if the AT&T blocking-state changes, see below

NOTE

Only the AT&T firmware supports the ACU_CALL_EVT_BLOCKING_STATE_CHANGE event. Use the appropriate firmware configuration (-s switch) to enable it. See the AT&T firmware release notes for more details.

To avoid polling this function you can either:

- use call_get_port_notification_wait_object() to obtain a wait object that is signaled when an event is queued for the port; or
- create an event queue (using acu_allocate_event_queue()) and then use call_set_port_notification_queue() to associate a particular port with that queue then wait for events to occur on the queue.

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

Example Usage

```
CALL_PORT_WAIT_OBJECT_PARMS wo_parms;
ACU_PORT_ID; /* the id of a port obtained previously */
CALL_PORT_NOTIFICATION_PARMS event_parms;
ACU_INT error;

INIT_ACU_STRUCT(&wo_parms);

wo_parms.port_id = port_id;
error = call_get_port_notification_wait_object(&wo_parms);
if (error != 0)
{
   printf("Failed getting wait event with error %d\n", error);
   exit(-1);
}
error = WaitForSingleObject(wo_parms.wait_object, INFINITE);
if (error == WAIT_OBJECT_0)
{
   INIT_ACU_STRUCT(&event_parms);
   event_parms.port_id = port_id;
   error = call_get_port_notification(&event_parms);
   if (error == 0)
   {
        /* handle event here */
   }
}
```

4.57 call_get_port_notification_wait_object() – get the wait object for a port

This function is used to get the wait object that is associated with a given port's notification event queue. The event returned by this function can be used with operating system specific wait functions such as `WaitForMultipleObjects()` or `poll()`.

Synopsis

```
ACU_ERR call_get_port_notification_wait_object(CALL_PORT_WAIT_OBJECT_PARMS*
wo_parms);

typedef struct _CALL_PORT_WAIT_OBJECT_PARMS
{
  ACU_ULONG               size;             /* IN */
  ACU_PORT_ID             port_id;          /* IN */
  ACU_WAIT_OBJECT         wait_object;      /* OUT */
} CALL_PORT_WAIT_OBJECT_PARMS;
```

Input Parameters

The `call_get_port_notification_wait_object()` function takes a pointer *`wo_parms`*, to a structure,`CALL_PORT_WAIT_OBJECT_PARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`port_id`

The *`port_id`* parameter should be a valid port id (returned from an earlier call to `call_open_port()`).

Return Values

`wait_object`

`wait_object` will be set to a valid operating system specific event associated with the specified port.

NOTE

The wait object associated with a port will remain signaled while there are notification events queued for that port.

Example Usage

See the example for `call_get_port_notification()` shown above.

4.58 call_get_global_event_wait_object() - get the global wait object for call events

This function is used to get the wait object that is signalled whenever there is a pending call event on any call handle that the application owns that is associated with the global call event queue (see `call_event()`).

The *`wait object`* returned by this function can be used with operating system specific wait functions such as `WaitForMultipleObjects()` or `poll()`.

Synopsis

```
ACU_INT call_get_global_event_wait_object(CALL_GLOBAL_WAIT_OBJECT_PARMS*
wo_parms);

typedef struct _CALL_GLOBAL_WAIT_OBJECT_PARMS
{
  ACU_ULONG                size;              /* IN */
  ACU_WAIT_OBJECT          wait_object;       /* OUT */
} CALL_GLOBAL_WAIT_OBJECT_PARMS;
```

NOTE

Events will not be reported for calls that have been associated with application-created event queues.

Input parameters

The `call_ get_global_event_wait_object()` function takes a pointer *`wo_parms`*, to a structure,`CALL_GLOBAL_WAIT_OBJECT_PARMS`. The structure must be initialised before invoking the function, (see section 2.2) .

Return Values

`wait_object`
Will be set to a valid operating system specific wait object associated with the global call event queue.

NOTE

The wait object returned will remain signaled while there are any events queued.

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

4.59 call_set_global_notification_queue() – set the queue for global notification events

Set the event queue for global notification events. Future global notification events will be notified via this queue.

Synopsis

```
ACU_ERR call_set_global_notification_queue(CALL_SET_GLOBAL_WAIT_QUEUE_PARMS*
queue_parms);

typedef struct tCALL_SET_GLOBAL_WAIT_QUEUE_PARMS
{
  ACU_UINT                size;              /* IN */
  ACU_EVENT_QUEUE         queue_id;          /* IN */
} CALL_SET_GLOBAL_WAIT_QUEUE_PARMS;
```

Input parameters

The `call_ set_global_notification_queue()` function takes a pointer *`queue_parms`*, to a structure,`CALL_ SET_GLOBAL_WAIT_QUEUE_PARMS`. The structure must be initialised before invoking the function, (see section 2.2) .

`queue_id:`
The *`queue_id`* field must be set to a valid queue as returned by `acu_allocate_event_queue` when creating a queue.

Return values

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

4.60 call_get_global_notification() - retreive the next global notification event

This function retrieves the next pending event from the global notification event queue.

NOTE

Currently the only use of this queue is to determine the outcome of Proxy/Gatekeeper registrations (see ipt_add_alias() in the V6 IP Telephony Guide).

Synopsis

```
ACU_ERR call_get_global_notification(CALL_GET_GLOBAL_NOTIFICATION_PARMS*
notify_parms);

typedef struct tCALL_GET_GLOBAL_NOTIFICATION_PARMS
{
   ACU_UINT      size;              /* IN */
   ACU_UINT      event;             /* OUT */
   void*         context;           /* OUT */
} CALL_GET_GLOBAL_NOTIFICATION_PARMS;
```

Input parameters

The `call_get_global_notification()` function takes a pointer *`notify_parms`*, to a structure,`CALL_ GET_GLOBAL_NOTIFICATION_PARMS`. The structure must be initialised before invoking the function, (see section 2.2) .

Return values

`event`

Upon successful execution, the event field will hold a value that uniquely identifies the event. O indicates that no event has been returned.

See the IP telephony API Guide, Registration Event Notification, for a list of the event codes associated with Proxy/Gatekeeper registration.

`context`

The context field holds a context that can help identify the operation that the event refers to (for instance, in Proxy/Gatekeeper registration, the context is the registration handle).
On successful completion, a value of zero is returned; otherwise, a negative value will be returned indicating the type of error.

4.61 call_get_global_notification_wait_object() – get the wait object for global notification events

Returns the wait object which will be signalled when a global notification event is pending.

Synopsis

```
ACU_ERR call_get_global_notification_wait_object(
        CALL_GET_GLOBAL_NOTIFICATION_WAIT_OBJECT_PARMS* wo_parms);

typedef struct tCALL_GET_GLOBAL_NOTIFICATION_WAIT_QUEUE_PARMS
{
  ACU_UINT            size;             /* IN */
  ACU_WAIT_OBJECT     wait_object;      /* OUT */
} CALL_GET_GLOBAL_NOTIFICATION_WAIT_OBJECT_PARMS;
```

Input parameters

The `call_get_global_notification_wait_object()` function takes a pointer `wo_parms`, to a structure, `CALL_GET_GLOBAL_NOTIFICATION_WAIT_OBJECT_PARMS`. The structure must be initialised before invoking the function, (see section 2.2) .

Return

`wait_object`

`wait_object` will be set to a valid operating system specific wait object associated with the global notification event queue.

NOTE

The wait object returned will remain signaled while there are any events queued.

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

4.62 call_open_iptel_port – Open a port for TiNG Media Configuration

There are times when an application may wish to have a greater level of control over the media resources on an IP Telephony call than the basic API offers – for example when dealing with IP to IP calls it is not necessary to switch the calls to TDM. In conjunction with Prosody X and Prosody S there is the option to open a special system port that allows the application to allocate and deallocate the media resources on the board, and thereby to control how they are used. The protocol will perform negotiation as usual, and will configure the users resources accordingly, but will not allocate additional resources e.g. TDM stream and timeslot.

Once an iptel port has been opened using this function, calls on this port can then be passed media by supplying a vmprxid and vmptxid in `call_openout()` for outgoing calls, and `call_accept()`, `call_progress()` or `xcall_incoming_ringing()` for incoming calls. These identifiers should be acquired via the TiNG API using the functions defined there. The allocation and deallocation of these resources is the responsibility of the application, the protocol will only perform basic configuration (codec type, packet length, VAD settings, remote RTP address etc.)

For more information on TiNG Media Configuration see Appendix K.

NOTE

The application is responsible for specifying the valid codecs for the call. The application must check which codecs are supported by the board.

Synopsis

```
ACU_ERR call_open_iptel_port(CALL_OPEN_IPTEL_PORT_PARMS* port_parms);
typedef struct tCALL_OPEN_IPTEL_PORT_PARMS
{
  ACU_ULONG           size;                 /* IN */
  ACU_INT             protocol_type;        /* IN */
  ACU_PORT_ID         port_id;              /* OUT */
} CALL_OPEN_IPTEL_PORT_PARMS;
```

Input Parameters

This `call_open_iptel_port()` function takes a pointer, *`port_parms`*, to a structure,`CALL_OPEN_IPTEL_PORT_PARMS`. The structure must be initialised before invoking the function.

`protocol_type`
The *`protocol_type`* field must be set to the id of the protocol that owns the port to open – either `S_SIP` or `S_H323`.

Return Values

`port_id`
The *`port_id`* 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.

4.63 call_set_dtmf_handling()

This function can be used to control DTMF User Input Indication event notification and relay.

Synopsis

```
ACU_INT call__set_dtmf_handling(CALL_DTMF_HANDLING_XPARMS *dtmf_handlingp);

typedef struct tcall__set_dtmf_handling_xparms
{
  ACU_ULONG           size;
  ACU_PORT_ID         unused;                     /* IN */
  ACU_CALL_HANDLE     handle;                     /* IN */
  ACU_UINT            enable_event_notification;  /* IN */
  ACU_UINT            relay_disabled;             /* IN */
} CALL_DTMF_HANDLING_XPARMS;
```

Input parameters

`call_set_dtmf_handling()` takes a pointer, *`dtmf_handlingp`*, to a structure, `CALL_DTMF_HANDLING_XPARMS`. The structure must be initialised before invoking the function.

`unused`
This field is ignored.

`handle`
The handle field is used to identify the call to which the DTMF handling options are to apply.

`enable_event_notification`
If set to 1, any User Input Indications that are received from the network will be identified at the API level through event notification. Valid values are :

`0` – disable event notification
`1` – enable event notification

`relay_disabled` (Not applicable to TiNG media configurations)

By default any User Input Indications that are received from the network will be relayed to the TDM side. This can be disabled by setting `relay_disabled` to 1.

`0` – relay enabled
1 – relay disable

Return values

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

4.64 h323_call_details() - Get h323 call details

This function is used to gather the details of a current h323 call, either incoming or outgoing, connected through the device driver. It returns a full collection of aliases unlike call_details, which contains only a subset.

Synopsis

```
ACU_ERR h323_call_details(H323_DETAIL_XPARMS* detailsp);

typedef struct h323_detail_xparms
{
   ACU_ULONG             size;                                  /*IN*/
   ACU_CALL_HANDLE       handle;                                /*IN*/
   ACU_LONG              timeout;                               /*OUT*/
   ACU_INT               valid;                                 /*OUT*/
   ACU_INT               stream;                                /*OUT*/
   ACU_INT               ts;                                    /*OUT*/
   ACU_INT               calltype;                              /*OUT*/
   ACU_CHAR              destination_addr[MAXADDR];             /*OUT*/
   ACU_CHAR              originating_addr[MAXADDR];             /*OUT*/
   ACU_CHAR              connected_addr[MAXADDR];               /*OUT*/
   ACU_ACT               app_context_token;                     /*OUT*/
   ACU_ULONG             feature_information;                   /*OUT*/
   ACU_CHAR              destination_display_name[MAXDISPLAY];  /*OUT*/
   ACU_CHAR              originating_display_name[MAXDISPLAY];  /*OUT*/
   ACU_CODEC             codecs[MAXCODECS];                     /*OUT*/
   MEDIA_SETTINGS        media_settings;                        /*OUT*/
   ACU_POINTER           vmprxid;                               /*OUT*/
   ACU_POINTER           vmptxid;                               /*OUT*/
   ACU_CHAR              media_call_type[MAXMEDIACALLTYPE];     /*OUT*/
   ACU_CHAR              destination_alias[MAXADDR];            /*OUT*/
   ACU_CHAR              originating_alias[MAXADDR];            /*OUT*/
   ACU_INT               h245_tunneling;                        /*OUT*/
   ACU_INT               faststart;                             /*OUT*/
   ACU_INT               early_h245;                            /*OUT*/
   ACU_CHAR              dtmf[MAXNUM];                          /*OUT*/
   ACU_INT               progress_location;                     /*OUT*/
   ACU_INT               progress_description;                  /*OUT*/
   ACU_ALIAS_LIST        originating_aliases;                   /*OUT*/
   ACU_ALIAS_LIST        destination_aliases;                   /*OUT*/
} ACU_PACK_DIRECTIVE H323_DETAIL_XPARMS;

    typedef struct acu_alias_list
{
   ACU_INT               no_of_aliases;
   ACU_CHAR              aliases[MAXALIASES][MAXADDR];
} ACU_PACK_DIRECTIVE ACU_ALIAS_LIST;
```

Input parameters

The `h323_call_details()` function takes a pointer, *`detailsp`*, to a structure, `H323_DETAIL_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

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

Return values

`timeout`
Not used.

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

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

NOTE

When the valid field indicates that there is some information in the structure, this does not mean that all information has been received for this call.

stream
Will contain the network stream number on which the call was received. For a full description of the stream numbering on Aculab cards consult the Switch Driver API Guide.

ts
Will contain the timeslot associated with the call.

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 calling line identity (CLI) and direct dial in (DDI) digits respectively if received by the signalling system.

connected_addr
Will contain the actual number of the party connected to a call. This may differ from the *`destination_addr`* field due to services such as call redirection.

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

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

feature_information
Contains an indication of whether any supplementary service information has been received and if so what type of information has been received. A separate API call, `call_feature_ details()`, must be used to retrieve this information.

This field can have any combination of the following values

```
FEATURE_FACILITY
FEATURE_USER_USER
FEATURE_DIVERSION
FEATURE_HOLD_RECONNECT
FEATURE_TRANSFER
FEATURE_RAW_DATA
FEATURE_RAW_MSG
```

destination_display_name, originating_display_name, codecs, media_settings, vmprxid, vmptxid, media_call_type, destination_alias, originating_alias, h245_tunneling, faststart, early_h245, dtmf, progress_location, progress_description

As per the values defined in Unique_xparms for Iptel

originating_aliases
Contains the originating endpoints aliases in an array of strings, where each string is an alias represented in URI notation. The structure includes an element

`no_of_aliases` which defines how many aliases have been set in the element `aliases`

`destination_aliases`
Contains the destination endpoints aliases in an array of strings, where each string is an alias represented in URI notation. The structure includes an element `no_of_aliases` which defines how many aliases have been set in the element `aliases`

NOTE

Be aware that the details returned by this function are not historical. In an application that is slow to process its event queue a number of EV_DETAILS events may be queued. The first call to h323_call_details() will return all of the updated details signaled by the remaining EV_DETAILS events in the event queue.

4.65 h323_gateway_mode() - Set H.323 transfer gateway mode

This function is used to set H.323 call transfer gateway mode and is required to facilitate H.323 call transfer across a network gateway.

When enabled, the H.323 service will not respond to incoming call transfer messages as defined by H.450.2 but will instead raise an `EV_EXT_TRANSFER_INFORMATION` event when an Invoke or ReturnResult message is received or an `EV_TRANSFER_REJECT` when a ReturnError message is received. Furthermore, several H.323 Call Transfer messages can only be sent when gateway mode is enabled.

Synopsis

```
ACU_ERR h323_gateway_mode(H323_GATEWAY_MODE_PARMS* modep);
typedef struct h323_gateway_mode_parms
{
   ACU_ULONG             size;                    /*IN*/
   ACU_PORT_ID           port_id;                 /*IN*/
   ACU_INT               mode;                    /*IN*/
} ACU_PACK_DIRECTIVE H323_GATEWAY_MODE_PARMS;
```

Input parameters

The `h323_gateway_mode()` function takes a pointer, `modep`, to a structure, `H323_GATEWAY_MODE_PARMS`. The structure must be initialised before invoking the function (see section 2.2).

`port_id`
The *`port_id`* field must be set to a valid port id (returned from a call to `call_open_port()`);

`mode`
The *mode* field is used to set the H.323 gateway mode. When set to `IPT_ENABLED`, H.323 gateway mode is enabled. When set to `IPT_DISABLED`, H.323 gateway mode is disabled.

Call Transfer

The four functions; `call_hold()`, `call_enquiry()`, `call_reconnect()` and `call_transfer()` are used to implement call transfer and associated features. This section describes the order in which these calls must be made and how different signalling systems handle the call transfer procedure.

Call transfer has been implemented on a single port basis (enquiry and held call must be on the same network port) for DPNSS, ETS300, NI2, DMS-100, and QSIG etc in accordance with the following documents.

EuroISDN	ETS300 196 Generic Functional Protocol for the Support of Supplementary Services.
	ETS300 369 Explicit Call Transfer (ECT).
NI2	GR-2865-CORE Generic Requirements for ISDN PRI Two B-Channel Transfer.
	Or Nortel NIS-A211-1 ISDN Primary Rate User-Network Interface Specification (When –cNA1 switch applied)
QSIG	Call Transfer Supplementary Service.
	(ISO/IEC 13869/13874, ECMA176/178, ETS300 259/261).
DPNSS	BTNR 188 - Sections 12 & 13 (Hold and Three Party Working. New Path only).
H.323	H.450.2 Call transfer supplementary service for H.323.

SIP	Session Initiated Protocol service examples – see http://www.ietf.org/internet-drafts/draft-ietf-sipping-service-examples-09.txt
DMS-100	Nortel NIS-A211-1 ISDN Primary Rate User-Network Interface Specification

The normal process for making a call transfer is as follows:

1. The call that is to be transferred is first put on hold by using `call_hold()`.
2. An enquiry call is then made via a call to `call_enquiry()`.
3. Once the call is connected a transfer may be initiated by a call to `call_transfer()`.

A call in the held state may be reconnected via a call to `call_reconnect()`.

For further details on call transfer with SIP, please see the ***Aculab SIP programmers guide.***

NOTE

For protocols that do not support putting a call in a hold state, a call to call_hold will not result in anything being transmitted on the line with the request acting as a 'dummy' hold, and no error will be indicated to the application. This is deliberately designed so as to allow an application to be coded in exactly the same way as for a protocol that supports call hold, maintaining the generic nature of the call control API.

For ETS300, NI-2, DMS-100, & QSIG the call transfer could also be requested when the enquiry call reaches the ringing (alerting) stage as follows:

1. The call that is to be transferred is first put on hold by using `call_hold()`.
2. An enquiry call is then made via a call to `call_enquiry()`.
3. Once the call is alerting (`EV_WAIT_FOR_ACCEPT`) a transfer may be initiated by a call to `call_transfer()`.

   (An enquiry call can be in an alerting state).

NOTE

For EuroISDN, call transfer in the alert state is an optional part of the protocol and therefore may not be supported by all networks.

NOTE

Call hold before a call transfer is optional for EuroISDN, DPNSS, NI-2, and DMS-100. A transfer can be done without the call first being put on hold. For EuroISDN, call transfer without hold is an optional part of the protocol and therefore may not be supported by all networks

NOTE

For QSIG, only call transfer without hold is supported.

NOTE

For EuroISDN, call transfer without hold may not be supported by all

networks.

NOTE

For H.323 the enquiry call may be virtual. In this case no enquiry call is made on the network. call_enquiry() is called with the cnf flag set to CNF_TSVIRTUAL. Additionally, for H.323 it is not necessary to place the call on hold first.

NOTE

For protocols that do not support putting a call in a hold state, a call to call_hold will not result in anything being transmitted on the wire, with no error indication returned to the calling application. This allows an application to be coded in the same way for both protocols that do, and protocols that do not, require a call to be put on hold. Maintaining the protocol independency of the call control API.

Once a call has been made to `call_transfer()` then the clear down procedure of the link will differ according to protocol and switch manufacturer. Protocols ETS300, NI2 and DMS-100 will clear down the link on receipt of the transfer (if the switch permits the operation). Protocols DPNSS and QSIG will enter a transit state in which all messages will be passed transparently between the two parties involved. Once the switch has established that this link is redundant it will establish a direct link and tear down the redundant link, using either Route Optimisation (DPNSS) or Path Replacement (QSIG).

4.66 call_hold() - Put a call on hold

An incoming or outgoing call can be put on hold by use of the `call_hold()` function. To allow enhancements to the driver API this call will in turn call `xcall_hold()` that may be called directly.

A successful hold will result in the state `EV_HOLD`. It is possible for a switch to reject a hold message sent by the Aculab software. If this happens the call will move to the state `EV_HOLD_REJECT`. For protocols that do not support hold then this function will automatically move the call to the state `EV_HOLD`.

Synopsis

```
ACU_ERR call_hold(int handle);

ACU_ERR xcall_hold(HOLD_XPARMS *holdp);

typedef struct hold_xparms
{
  ACU_ULONG      size;              /* IN */
  ACU_CALL_HANDLE     handle;       /* IN */
  union
  {
    struct
    {
      ACU_INT    hold_type;         /* IN */
    } sig_h323;
  } ACU_PACK_DIRECTIVE  unique_xparms;
} HOLD_XPARMS;
```

Input parameters

`call_hold()`

The input parameter *`handle`* identifies the call that is to be put on hold.

`xcall_hold()`

The `xcall_hold()` function takes a pointer, *`holdp`*, to a structure,`HOLD_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`handle`

The *`handle`* field identifies the call that is to be put on hold.

H.323 Specific Parameters

`hold_type`

The *`hold_type`* field identifies the method of call hold to be used. Options are `ACU_HOLD_NEAR_END` (initiates a near end hold as defined in H.450.4), `ACU_HOLD_REMOTE` (initiates a remote hold as defined in H.450.4) or `ACU_HOLD_MEDIA` (initates a third party hold by sending an empty capability set).

NOTE

When an EV_HOLD event is detected a call to call_feature_details can determine the type of hold.

Return values

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

4.67 call_reconnect() - Reconnect a call on hold

A call that is in the held state can be reconnected by use of the `call_reconnect()` function. To allow enhancements to the driver API this call will in turn call `xcall_reconnect()` which may be called directly. This call will cause the call to return to the connected state. It is possible for a switch to reject a reconnect message sent by the application. If this happens the call will move to the state `CS_RECONNECT_REJECT`.

NOTE

For protocol reasons, an H.323 call may be in CS_HOLD state due to a remote pause, rather than because call_hold() has been called. In this case, call_reconnect() cannot reconnect the call, and CS_RECONNECT_REJECT will be generated. Such calls must be resumed remotely.

Synopsis

```
ACU_ERR call_reconnect(int handle);

ACU_ERR xcall_reconnect(HOLD_XPARMS *holdp);

typedef struct hold_xparms
{
  ACU_ULONG                size;          /* IN */
  ACU_CALL_HANDLE          handle;        /* IN */
  union
  {
    struct
    {
      ACU_INT         hold_type;          /* IN */
    } sig_h323;
  }ACU_PACK_DIRECTIVE  unique_xparms;
} HOLD_XPARMS;
```

Input parameters

call_reconnect()

The input parameter *handle* identifies the call that is to be reconnected.

xcall_reconnect()

The `xcall_reconnect()` function takes a pointer, *holdp*, to a structure,`HOLD_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

handle

The *handle* field identifies the call that is to be reconnected.

H.323 Specific Parameters

hold_type

The *hold_type* field identifies the method of call hold to be reconnected. Options are `ACU_HOLD_NEAR_END` (initiates a near end reconnect as defined in H.450.4), `ACU_HOLD_REMOTE` (initiates a remote reconnect as defined in H.450.4 or `ACU_HOLD_MEDIA` (initates a third party reconnect by sending a new capability set).

Return values

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

4.68 call_enquiry() - Make an enquiry call

During the process of call transfer, this function allows an application to make an enquiry call, that is, an outgoing call to a third party.

The function is essentially the same as `call_openout()`, having all of the same call states and events. The function registers the enquiry call requirement with the device driver. If the driver is satisfied with the calling parameters, it will return a unique call handle. The call handle must be used in all successive call control related operations for this call.

Synopsis

```
ACU_ERR call_enquiry (struct out_xparms *enquiryp)
```

NOTE

For a full description of out_xparms, see the call_openout().

Input parameters

The `call_enquiry()` function takes a pointer, *`enquiryp`*, to a structure,`out_xparms`. The structure must be initialised as specified for `call_openout()` before invoking the function.

NOTE

When choosing the time slot it is important to be aware of the type of transfer available for the protocol. For ETS300 it possible to use the timeslot of the held call and thus use fewer timeslots. To initiate such an enquiry the timeslot of the held party must be specified in the timeslot field of out_xparms. For NI2, QSIG and DPNSS the call must go out on a different timeslot.

Return values

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

4.69 call_transfer() - Call transfer

After a successful enquiry call, the calls may be transferred by use of the `call_transfer()` function call. The enquiry call is deemed successful if the state of the enquiry call has reached `CS_CONNECTED` or `CS_OUTGOING_RINGING` (for 'blind' transfer).

Synopsis

```
ACU_ERR call_transfer(TRANSFER_XPARMS *transferp);

typedef struct transfer_xparms
{
  ACU_ULONG      size;          /* IN */
  ACU_INT        handlea;       /* IN */
  ACU_INT        handlec;       /* IN */
} TRANSFER_XPARMS;
```

Input parameters

The `call_transfer()` function takes a pointer, *transferp*, to a structure,`TRANSFER_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

handlea

The *handlea* field is the handle of the 'A Party', that is the handle of the call that is on hold.

handlec

The *handlec* field is the handle of the 'C Party' call, that is the handle for the enquiry call (returned by `call_enquiry()`).

Example:

```
INIT_ACU_CL_STRUCT (&transfer);
transfer.handlea = hold_handle;
transfer.handlec = enquiry_handle;
call_transfer(&transfer);
```

Return values

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

Some protocols have the ability to reject transfer messages. If the switch rejects the transfer then an `EV_TRANSFER_REJECT` event will be received on the enquiry call. For ETS300 or QSIG this event can be received on either of the two calls involved in the transfer. If no rejection message is received then the held and enquiry calls will stay in the same state until the switch clears down the calls. For ETS300, and NI2 this should be almost instantaneous. With QSIG and DPNSS this process takes longer as the switch must optimise the link and use an alternative path.

Resilience framework

This feature provides a framework in which application level resilience may be built into call control networks. Multiple redundant call control applications are executed in such a way that a "spare" application may recover calls originally setup by a redundant application.

API support for redundant applications

By default Aculab telephony software exhibits a simple client-server based behaviour regarding the conservation of call control resources. When an application abnormally terminates the Aculab telephony software hosting any calls left open are configured to "clear down" those calls and return any associated resource. Such behaviour is detrimental to a system wishing to provide application-level resilience, which would

prefer that a spare application recover calls left by a now defunct application.

The following APIs permit the original application to enquire of a system unique identifier for call, and in doing so instruct the server software not to clear this call in the event of application demise. The identifier acquired by `call_get_failover_id()` may be saved to a storage mechanism of choice. The ownership of a call modified by the previous functionality may then be transferred to a different application through `call_reopen()`.

The resilience APIs are intended to provide an initial foundation for the use of application writers. The application writer must supply additional resources to “glue” the overall system together. An essential component for such a system is a storage medium, for example a network file system or shared memory, in order that failover identifiers and call information may be saved and retrieved. Additionally, a scheme for detecting application failures must be devised to determine when to activate backup applications.

The requisite APIs are detailed below.

NOTE

These resilience functions are not supported by all signalling protocols: please refer to the specific signalling protocol firmware release notes for further clarification.

4.70 call_get_failover_id() - Get unique identifier for a call

The routine `call_get_failover_id()` provides the facility of retrieving a unique identifier for a call. The Aculab Call handle is not sufficient for this purpose as it is only unique on a particular chassis and not unique to the issuing software/firmware in distributed scenarios. After the failover identifier has been collected, the Aculab telephony software relinquishes the requirement to clear down that particular call on application termination. The calling application may save this identifier to a storage medium for future reference. All aspects of call control are still permitted using the current call handle by the original application after executing `call_get_failover_id()`.

The failover identifier may be retrieved by another application and used to "re-open" the original call. It is the application developer's responsibility to remove the identifier from storage when the "last owned" call handle has been released.

A failover identifier may only be collected from a call handle when that call is in an appropriate state:

- For SIP, all states except `EV_WAIT_FOR_INCOMING` are valid
- For ISDN, the call must have valid call details

`ERR_COMMAND` will be returned if `call_get_failover_id()` is called for a handle in an invalid state.

Synopsis

```
ACU_ERR call_get_failover_id(CALL_GET_FAILOVER_ID_PARMS* parms);

typedef struct tCALL_GET_FAILOVER_ID_PARMS
{
   ACU_ULONG          size;          /* IN */
   ACU_CALL_HANDLE    handle;        /* IN */
   ACU_FAILOVER_ID    failover_id;   /* OUT */
} CALL_GET_FAILOVER_ID_PARMS;
```

The function `call_get_failover_id ()` takes a pointer to a structure `CALL_GET_FAILOVER_ID_PARMS`. The structure must be initialised before invoking the function.

Input Parameters

`handle`
The handle field contains the Aculab call handle relating the call whose failover identifier we wish to retrieve.

Return Values

`failover_id`
On successful execution a 64 bit failover identifier value for the call will be written to this field.

On successful completion, a value of zero is returned. The following codes may be returned to report errors:

`ERR_HANDLE` Supplied handle was invalid

`ERR_COMMAND` Call was in inappropriate state for the command to complete

`ERR_CFAIL` Unspecific error occurred

4.71 call_reopen() - Recover a handle to an earlier call

This routine acquires a handle to a call which was previously opened and in which a failover identifier has been retrieved. When the recovering application successfully re-opens a call, the "ownership" of that call is transferred from the previous application to this recovering application. Call control operations will be invalid on the handle in the previous application, any subsequent function calls receiving an `ERR_HANDLE` return. Call control events will now be raised to the re-opening program. It is possible for a call to re-opened several times once a failover id has been acquired.

When a call has been successfully re-opened, the last call control event is presented to the new application to assist the synchronisation of the call control state machine.

Synopsis

```
ACU_ERR call_reopen(CALL_REOPEN_PARMS* parms);

typedef struct tCALL_REOPEN_PARMS
{
   ACU_ULONG          size;              /* IN */
   ACU_PORT_ID        net;               /* IN */
   ACU_FAILOVER_ID    failover_id;       /* IN */
   ACU_EVENT_QUEUE    queue_id;          /* IN */
   ACU_ACT            app_context_token; /* IN */
   ACU_CALL_HANDLE    handle;            /* OUT */
} CALL_REOPEN_PARMS;
```

Input parameters

net
The port id in which this call is to be opened on.

failover_id
Unique identifier for the call as collected by call_get_failover_id.

queue_id
The event queue to which events for this call should be sent. This must be either a unique event queue identity as returned by `acu_allocate_event_queue()` or zero to denote the default call event queue for the port that this call is made on.

app_context_token
An application specific token which may be returned with call events for this call.

Return values

handle
An application specific handle to the recovered call.

On successful completion, a value of zero is returned. The following codes may be returned to report errors:

`ERR_HANDLE` Supplied handle was invalid

`ERR_CFAIL` Unspecific error occurred

4.72 call_reattach_fmw() – Reattach the call driver to running signaling firmware.

This routine allows a new call driver to attach to a port's firmware whose original driver is now defunct. Reattachment is necessary as it allows a call driver to synchronize with a running firmware and gain knowledge of any existing calls. After reattachment, applications may gain access to existing calls using `call_reopen()`.

Subsequent to invoking this routine, call control by the original call driver is no longer possible.

Synopsis

```
ACU_ERR call_reattach_fmw(CALL_REATTACH_PARMS* parms);

typedef struct call_reattach_parms
{
  ACU_ULONG      size;                              /*IN*/
  ACU_PORT_ID    net;                               /*IN*/
  ACU_CHAR       config_string[CL_MAX_FMW_PARMS]; /*IN*/
} ACU_PACK_DIRECTIVE CALL_REATTACH_PARMS;
```

Input parameters

net

The port id where running signalling firmware resides.

config_string

A null terminated string containing the protocol configuration command line arguments that were implemented at firmware download (for example, -c and -s switches).

Return values

`ERR_NOT_IMPLEMENTED` Function not implemented for hardware.

`ERR_COMMAND` Function called when firmware is in an inappropriate state.

`ERR_CFAIL` Unspecific error occurred

4.73 call_release_lost_failover_ids() – Release any unattached call handles.

Under certain circumstances the application may no longer have any record of the failover identifiers currently associated with calls on the system. If this happens then the resources associated with these calls can be released by using this routine. Any calls in progress will be cleared down.

Synopsis

```
ACU_ERR call_release_lost_failover_ids(CALL_RELEASE_LOST_FAILOVER_IDS_PARMS*
parms);
typedef struct tCALL_RELEASE_LOST_FAILOVER_IDS_PARMS
{
   ACU_ULONG      size;     /* IN */
   ACU_PORT_ID    net;      /* IN */
} ACU_PACK_DIRECTIVE CALL_RELEASE_LOST_FAILOVER_IDS_PARMS;
```

Input parameters

net

The port id in which this call is to be opened on.

Return values

On successful completion, a value of zero is returned. The following codes may be returned to report errors:

`ERR_CFAIL` Unspecific error occurred

4.74 call_change_media() – Change the media for an existing call

An existing (audio) call can be renegotiated to by use of the `call_change_media()` function, for example to change the call to use T38 Fax.

A successful negotiation will result in the extended state `EV_EXT_MEDIA_CHANGE_ACCEPT`, followed by `EV_EXT_MEDIA_CHANGE_COMPLETED`. If the negotiation fails `EV_EXT_MEDIA_CHANGE_REJECT` or `EV_EXT_MEDIA_CHANGE_TIMEOUT` (i.e. no response) will be raised.

Synopsis

```
ACU_ERR call_change_media(CALL_CHANGE_MEDIA_PARMS* mediap);

typedef struct call_change_media_parms
{
  ACU_ULONG             size;
  ACU_CALL_HANDLE       handle; /*IN*/
  union
  {
    union
    {
      struct
      {
        ACU_CHAR local_rtp_address[ACU_MAX_IP_ADDRESS]; /*IN*/
        ACU_UINT local_rtp_port;          /*IN*/
        ACU_CHAR local_rtcp_address[ACU_MAX_IP_ADDRESS];/*IN*/
        ACU_UINT local_rtcp_port;          /*IN*/
        ACU_CHAR remote_rtp_address[ACU_MAX_IP_ADDRESS];
        ACU_UINT remote_rtp_port;
        ACU_CHAR remote_rtcp_address[ACU_MAX_IP_ADDRESS];
        ACU_UINT remote_rtcp_port;
        union
        {
          struct
          {
            struct
            {
              ACU_UINT    transport_type; /*IN*/
              ACU_UINT    bit_rate; /*IN*/
              ACU_UINT    fill_bit_removal; /*IN*/
              ACU_UINT    transcoding_jbig; /*IN*/
              ACU_UINT    transcoding_mmr; /*IN*/
              ACU_UINT    version; /*IN*/
              ACU_UINT    fax_rate_management; /*IN*/
              ACU_UINT    udp_max_buffer; /*IN*/
              ACU_UINT    udp_max_datagram; /*IN*/
              ACU_UINT    udp_error_correction;/*IN*/
              ACU_UINT    tcp_bidirectional; /*IN*/
            } t38Fax;
          } data;
        } media;
        ACU_UINT reject_reason;
      } sig_h323;
    } sig_iptel;
  } unique_xparms;
} ACU_PACK_DIRECTIVE CALL_CHANGE_MEDIA_PARMS;
```

Input parameters

`call_change_media()`

The `call_change_media()` function takes a pointer, *`mediap`*, to a structure,`CALL_CHANGE_MEDIA_PARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`handle`

The *`handle`* field identifies the call that is to have its media changed.

H.323 Specific Parameters

local_rtp_address
The local_rtp_address field contains the IP address that the remote endpoint should use to send rtp data to.

local_rtp_port
The local_rtp_port field contains the IP port that the remote endpoint should use to send rtp data to.

local_rtcp_address
The local_rtcp_address field contains the IP address that the remote endpoint should use to send rtcp data to.

local_rtcp_port
The local_rtcp_port field contains the IP port that the remote endpoint should use to send rtcp data to.

H.323 T38 Fax Specific Parameters

The media.data.t38Fax structure contains a number of fields that can be used to negotiate a T38 fax call. Not all settings may be appropriate and will depend on the capabilities of the T38 software.

transport_type
The transport_type field indicates whether UDP or TCP should be used. For UDP this field should be set to T38_FAX_TRANSPORT_UDP. For TCP this field should be set to T38_FAX_TRANSPORT_TCP.

bit_rate
The bit_rate field contains the requested bit rate in units of 100 bit/s.

fill_bit_rate
The fill_bit_rate field if set to 1 indicates that it has the ability to remove and insert fill bits.

transcoding_jbig
The transcoding_jbig field if set to 1 indicates that the gateway has the ability to transcode in real time between the line compression and JBIG for transfer over the IP network.

transcoding_mmr
The transcoding_mmr field if set to 1 indicates that the gateway has the ability to transcode in real time between the line compression and MMR for transfer over the IP network.

version
The version field indicates the version of T38 used. It should be set to one of the following values T38_FAX_VERSION_1, T38_FAX_VERSION_2 or T38_FAX_VERSION_3.

fax_rate_management
The fax_rate_management field indicates whether local generation of TCF is required (for TCP) or if transfer of TCF is required (UDP). It should be set to one of the following values T38_FAX_RATE_MANAGEMENT_LOCAL_TCF or T38_FAX_RATE_MANAGEMENT_TRANSFERRED_TCF.

udp_max_buffer
The udp_max_buffer field, if set to a non-zero value, indicates the maximum number of octets that can be stored before an overflow condition occurs. The default is 0, indicating that there is no maximum value.

udp_max_datagram
The udp_max_datagram field, if set to a non-zero value, indicates the maximum size of a packet that can be received. The default is 0, indicating that there is no maximum size.

udp_error_correction

The udp_error_correction field indicates whether FEC (RFC 2733) or redundancy (RFC 2198) error correction is to be used. It should be set to one of the following values T38_FAX_ERROR_CORRECTION_FEC or T38_FAX_ERROR_CORRECTION_REDUNDANCY.

tcp_bidirectional

The tcp_bidirectional field if set to 1 indicates support for single bidirectional channels.

NOTE

For H323 Any media successfully negotiated using this function will no longer be managed by the H323 service and will instead be under the control of the application.

Return values

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

4.75 call_change_media_accept()

The event `EV_EXT_MEDIA_CHANGE_REQUEST` indicates a change to the media associated with a call. To accept this change the `call_change_media_accept()` function must be used.

When the negotiation is completed the extended event `EV_EXT_MEDIA_CHANGE_COMPLETED` is raised.

Synopsis

```
ACU_ERR call_change_media_accept(CALL_CHANGE_MEDIA_ACCEPT_PARMS* acceptp);

typedef struct call_change_media_accept_parms
{
   ACU_ULONG            size;
   ACU_CALL_HANDLE      handle; /*IN*/
   union
   {
        union
        {
             struct
             {
                  ACU_CHAR rtp_address[ACU_MAX_IP_ADDRESS]; /*IN*/
                  ACU_UINT rtp_port; /*IN*/
                  ACU_CHAR rtcp_address[ACU_MAX_IP_ADDRESS]; /*IN*/
                  ACU_UINT rtcp_port; /*IN*/
             } sig_h323;
        } sig_iptel;
   } unique_xparms;
} ACU_PACK_DIRECTIVE CALL_CHANGE_MEDIA_ACCEPT_PARMS;
```

Input parameters

`call_change_media_accept()`

The `call_change_media_accept()` function takes a pointer, *`acceptp`*, to a structure,`CALL_CHANGE_MEDIA_ACCEPT_PARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`handle`

The *`handle`* field identifies the call that is to have its media changed.

H.323 Specific Parameters

`rtp_address`

The `rtp_address` field contains the IP address that the remote endpoint should use to send rtp data to.

`rtp_port`

The `rtp_port` field contains the IP port that the remote endpoint should use to send rtp data to.

`rtcp_address`

The `rtcp_address` field contains the IP address that the remote endpoint should use to send rtcp data to.

`rtcp_port`

The `rtcp_port` field contains the IP port that the remote endpoint should use to send rtcp data to.

NOTE

For H323 Any media successfully negotiated using this function will no longer be managed by the H323 service and will instead be under the control of the application.

Return values

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

4.76 call_change_media_reject()

The event EV_EXT_MEDIA_CHANGE_REQUEST indicates a change to the media associated with a call. To reject this change the call_change_media_reject() function must be used.

Synopsis

```
ACU_ERR call_change_media_reject(CALL_CHANGE_MEDIA_REJECT_PARMS* rejectp);

typedef struct call_change_media_reject_parms
{
   ACU_ULONG             size;
   ACU_CALL_HANDLE       handle; /*IN*/
   ACU_UINT              reason; /*IN*/
} ACU_PACK_DIRECTIVE CALL_CHANGE_MEDIA_REJECT_PARMS;
```

Input parameters

call_change_media_reject()

The call_change_media_reject() function takes a pointer, *rejectp*, to a structure,CALL_CHANGE_MEDIA_REJECT_PARMS. The structure must be initialised before invoking the function, (see section 2.2).

handle

The *handle* field identifies the call that is to have its media changed.

reason

The reason field must be set ot a value that indicates the reason for rejecting the proposed media change. For H323 possible values are H323_MEDIA_REJECT_MODE_UNAVAILABLE, H323_MEDIA_REJECT_MULTIPOINT_CONSTRAINT and H323_MEDIA_REJECT_REQUEST_DENIED.

Return values

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

4.77 call_media_details() – Get details of a change of media for an existing call

When one of the following events `EV_EXT_MEDIA_CHANGE_REQUEST`, `EV_EXT_MEDIA_CHANGE_REJECT` or `EV_EXT_MEDIA_CHANGE_COMPLETED` is raised there will be information available for the application to use. To collect this information `call_media_details()` should be called.

Synopsis

```
ACU_ERR call_change_media(CALL_CHANGE_MEDIA_PARMS* detailsp);
```

```
typedef struct call_change_media_parms
{
  ACU_ULONG             size;
  ACU_CALL_HANDLE       handle; /*IN*/
  union
  {
    union
    {
      struct
      {
        ACU_CHAR local_rtp_address[ACU_MAX_IP_ADDRESS]; /* OUT */
        ACU_UINT local_rtp_port;          /* OUT */
        ACU_CHAR local_rtcp_address[ACU_MAX_IP_ADDRESS]; /* OUT */
        ACU_UINT local_rtcp_port;          /* OUT */
        ACU_CHAR remote_rtp_address[ACU_MAX_IP_ADDRESS]; /* OUT */
        ACU_UINT remote_rtp_port;          /* OUT */
        ACU_CHAR remote_rtcp_address[ACU_MAX_IP_ADDRESS]; /* OUT */
        ACU_UINT remote_rtcp_port; /* OUT */
        union
        {
          struct
          {
            struct
            {
              ACU_UINT    transport_type; /* OUT */
              ACU_UINT    bit_rate; /* OUT */
              ACU_UINT    fill_bit_removal; /* OUT */
              ACU_UINT    transcoding_jbig; /* OUT */
              ACU_UINT    transcoding_mmr; /* OUT */
              ACU_UINT    version; /* OUT */
              ACU_UINT    fax_rate_management; /* OUT */
              ACU_UINT    udp_max_buffer; /* OUT */
              ACU_UINT    udp_max_datagram; /* OUT */
              ACU_UINT    udp_error_correction; /* OUT */
              ACU_UINT    tcp_bidirectional; /* OUT */
            } t38Fax;
          } data;
        } media;
        ACU_UINT reject_reason; /* OUT */
      } sig_h323;
    } sig_iptel;
  } unique_xparms;
} ACU_PACK_DIRECTIVE CALL_CHANGE_MEDIA_PARMS;
```

Input parameters

`call_media_details()`

The `call_media_details()` function takes a pointer, *`detailsp`*, to a structure,`CALL_CHANGE_MEDIA_PARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`handle`

The *`handle`* field identifies the call for which details should be collected.

H.323 Specific Parameters

`local_rtp_address`

The `local_rtp_address` field contains the IP address that the remote endpoint will use

to send rtp data to.

local_rtp_port
The local_rtp_port field contains the IP port that the remote endpoint will use to send rtp data to.

local_rtcp_address
The local_rtcp_address field contains the IP address that the remote endpoint should use to send rtcp data to.

local_rtcp_port
The local_rtcp_port field contains the IP port that the remote endpoint should use to send rtcp data to.

remote_rtp_address
The remote_rtp_address field contains the IP address that the local endpoint will use to send rtp data to.

remote_rtp_port
The remote_rtp_port field contains the IP port that the local endpoint will use to send rtp data to.

remote_rtcp_address
The remote_rtcp_address field contains the IP address that the local endpoint should use to send rtcp data to.

remote_rtcp_port
The remote_rtcp_port field contains the IP port that the local endpoint should use to send rtcp data to.

reject_reason
The reject_reason field contains the reason a proposed media change was rejected by the remote end. Possible reasons are H323_MEDIA_REJECT_MODE_UNAVAILABLE, H323_MEDIA_REJECT_MULTIPOINT_CONSTRAINT and H323_MEDIA_REJECT_REQUEST_DENIED. This field will only be set after a EV_EXT_MEDIA_CHANGE_REJECTED event.

The remote_rtp_address, remote_rtp_port, remote_rtcp_address and remote_rtcp_port fields will only be ready to read after receiving the EV_EXT_MEDIA_CHANGE_COMPLETED event.

H.323 T38 Fax Specific Parameters

transport_type
The transport_type field indicates whether UDP or TCP should be used. For UDP this field will be set to T38_FAX_TRANSPORT_UDP. For TCP this field will be set to T38_FAX_TRANSPORT_TCP.

bit_rate
The bit_rate field contains the requested bit rate in units of 100 bit/s.

fill_bit_rate
The fill_bit_rate field if set to 1 indicates the ability to remove and insert fill bits.

transcoding_jbig
The transcoding_jbig field if set to 1 indicates the ability to transcode in real time between the line compression and JBIG for transfer over the IP network.

transcoding_mmr
The transcoding_mmr field if set to 1 indicates the ability to transcode in real time between the line compression and MMR for transfer over the IP network.

version
The version field indicates the version of T38 used. It will be set to one of the following values T38_FAX_VERSION_1, T38_FAX_VERSION_2 or T38_FAX_VERSION_3.

fax_rate_management
The fax_rate_management field indicates whether local generation of TCF is required

(for TCP) or if transfer of TCF is required (UDP). It will be set to one of the following values `T38_FAX_RATE_MANAGEMENT_LOCAL_TCF` or `T38_FAX_RATE_MANAGEMENT_TRANSFERRED_TCF`.

`udp_max_buffer`
The `udp_max_buffer` field, if set to a non-zero value, indicates the maximum number of octets that can be stored before an overflow condition occurs. The default is 0, indicating that there is no maximum value.

`udp_max_datagram`
The `udp_max_datagram` field, if set to a non-zero value, indicates the maximum size of a packet that can be received. The default is 0, indicating that there is no maximum size.

`udp_error_correction`
The `udp_error_correction` field indicates whether FEC (RFC 2733) or redundancy (RFC 2198) error correction is to be used. It will be set to one of the following values `T38_FAX_ERROR_CORRECTION_FEC` or `T38_FAX_ERROR_CORRECTION_REDUNDANCY`.

`tcp_bidirectional.`
The `tcp_bidirectional` field if set to 1 indicates support for single bidirectional channels.

The fields in the `t38Fax` struct will be available to read after receiving a `EV_EXT_MEDIA_CHANGE_REQUEST` event.

Return values

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

5 Miscellaneous functions

The multiple device driver API supports a mixture of Aculab cards in the same system. The following functions are provided so that an application can obtain:

the direction of a call

the port number for a particular handle

to convert a handle to logical channel number

NOTE

These functions do not make calls to the device driver but are resolved by the library.

API	Description
`call_port_2_swdrvr()`	Used to obtain the switch id relating to a particular port id. This switch id may then be used to access a particular switch in a multiple switch driver environment. (See Switch Control API Guide).
`call_handle_2_io()`	Used to convert a given call reference value or handle to an indication of call direction.
`call_handle_2_port()`	Used to convert a given handle to its equivalent port id number.
`call_handle_2_chan()`	Used to convert a given handle to a logical channel number, which may be useful as an index into a control structure array. This removes the requirement for scanning control structure arrays for any given handle value.
`call_get_port_dsp_stream()`	Used to retrieve a trunk DSP stream that was allocated to a network port at protocol firmware download time
`idle_net_ts()`	Used to write the IDLE pattern on to a given network stream and timeslot once the call has been terminated.
`call_api_version()`	Returns the revision number of the API with any additional note as applicable.

NOTE

It is now possible to associate data with a call, so the above techniques are included for backwards compatibility.

5.1 call_port_2_swdrvr() - Determine port's switch

This function may be used to obtain the switch id relating to a particular network port. This switch id may then be used to access a particular switch in a multiple switch driver environment. (See Switch Control API Guide).

NOTE

For information on downloading firmware to an IP telephony card refer to the IP Telephony Card V6 API Guide.

Synopsis

```
ACU_ERR call_port_2_swdrvr(ACU_PORT_ID portnum);
```

Input parameters

The input parameter *portnum* identifies the network port number for which the switch driver number is required and will have a valid port id as returned from `call_open_port()`.

Return values

On successful completion the function will either return the switch id number relating to the port id provided or a negative value will be returned indicating the type of error (see Error Codes).

5.2 call_handle_2_io() - Convert handle to call direction

This function can be used to determine call direction the given handle.

Synopsis

```
ACU_ERR call_handle_2_io(ACU_CALL_HANDLE handle);
```

Input parameters

The input parameter *handle* must contain a valid call handle.

Return values

On completion, the function will return:

0 - Incoming call

1 - Outgoing call

2 – Enquiry call

The return value may be useful as an index into an array of control structures selecting either incoming outgoing calls.

NOTE

In V6 it is prefereable to associate data with a call handle using the app_context_token field of call_openin() or call_openout().

5.3 call_handle_2_port() - Determine port_id of a given handle

This function can be used to determine the port_id given.

Synopsis

```
ACU_ERR call_handle_2_port(ACU_CALL_HANDLE handle);
```

Input parameters

The input parameter *handle* must contain a valid call handle.

Return values

On successful completion the function will return valid port id as returned from `call_open_port()` with the given `handle`. Otherwise a negative value will be returned indicating the type of error.

5.4 call_handle_2_chan() - Convert handle to logical channel number

This function can be used to convert a given handle to a logical channel number, which may be useful as an index into a control structure array. This removes the requirement for scanning control structure arrays for any given handle value.

NOTE

call_handle_2_chan() is deprecated. to associate data with a call, use the app_context_token field of call_openin() or call_openout()

NOTE

Traditionally the maximum number of channels achievable in a particular direction on a particular port was limited to NCHAN (255). When using IP Telephony, especially when an application is not using any media processing resources for a call, it is possible to allocate more than NCHAN calls on a single port. When this happens, call_handle_2_chan() will return a channel larger than NCHAN. NCHAN has not been changed for backwards compatibility reasons.

The logical channel has no relationship with the timeslot value. The timeslot value can only be obtained using the `call_details()` API call.

Synopsis

```
ACU_ERR call_handle_2_chan(ACU_CALL_HANDLE handle);
```

Input parameters

The input parameter *`handle`* must contain a valid call handle.

Return values

On successful completion the function will return the value of the logical channel associated with this handle and will have the range `0 - 255`. Otherwise a negative value will be returned indicating the type of error.

NOTE

The channel number is only unique for the port the call is on.

NOTE

The channel number returned is not the timeslot number but is the number of the call control block in the device driver responsible for this call. When the call is in progress the timeslot will be dynamically assigned to a control block.

There is no function in the Aculab API that can convert a call handle to a timeslot. If you need to ascertain the timeslot then use the handle in conjunction with the `call_details()` function to obtain this information.

5.5 call_get_port_dsp_stream() – Retrieve a trunk DSP stream allocated to a network port.

This function is used to retrieve a trunk DSP stream that was allocated to a network port at protocol firmware download time.

NOTE

Currently only supported by PMX hardware.

Synopsis

```
ACU_ERR call_get_port_dsp_stream (CALL_DSP_STREAM_PARMS* parms);

typedef struct tCALL_DSP_STREAM_PARMS
{
ACU_ULONG   size;                          /*IN*/
ACU_PORT_ID port_id;                       /*IN*/
ACU_UINT    stream;                        /*OUT*/
} ACU_PACK_DIRECTIVE CALL_DSP_STREAM_PARMS;
```

Input parameters

Port_id

The port id where running signalling firmware resides.

Return values

Stream

The associated Trunk DSP stream.

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

5.6 idle_net_ts() - Write the IDLE pattern to the network timeslot

This function may be used to write the IDLE pattern on to a given network stream and timeslot once the call has been terminated.

Synopsis

```
ACU_ERR idle_net_ts(ACU_PORT_ID portnum, int timeslot)
```

NOTE

This function makes use of the switch API and, as such, will only function correctly if the application has already opened the card for switch API use using the acu_open_switch() function.

Input parameters

The input parameter *portnum* identifies the network port id for which the stream number is required and will have a valid port id as returned from `call_open_port()`.

The input parameter *timeslot* must contain a valid timeslot number on the network stream specified by portnum to which the `IDLE` pattern is to be written.

NOTE

It is important that this function should be used especially when using CAS tone signaling. Calling the function at the end of the incoming or outgoing call will ensure that the tone detect logic is reconnected to the network port ready to process the next call.

5.7 call_api_version()

Returns the revision number of the API with any additional note as applicable.

Synopsis

```
ACU_ERR call_api_version(CALL_API_VERSION_PARMS* version);

typedef struct tCALL_API_VERSION_PARMS
{
  ACU_UINT       size;                  // IN
  ACU_UINT       major;                 // OUT
  ACU_UINT       minor;                 // OUT
  ACU_UINT       rev;                   // OUT
  ACU_CHAR       desc[MAX_VER_DESC];    // OUT
} CALL_API_VERSION_PARMS;
```

Returns

major

The first numerical value of the revison.

minor

The second numerical value of the revision.

rev

The third numerical value of the revision.

desc

A null terminated string containing human readable text describing the version, for example:

6.1.3 Beta

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

6 Downloading and Configuring Firmware

The section of the guide describes the functions available in the library to allow the signalling system firmware to be downloaded onto Aculab cards via the device driver or allow the signalling system to be reset and restarted.

API	Description
`call_restart_fmw()`	Used to restart or change the configuration of the signalling system firmware on a port on an Aculab card without having to reboot the machine. The function will cause the selected port processor on the card to be reset and the firmware to be downloaded.
`call_reconfig_fmw ()`	Used to reconfigure the –s switches for the signalling system firmware without restarting the firmware.
`call_stop_fmw()`	This function is used to stop the firmware on a port.
`call_is_download()`	Used to ascertain whether signalling system firmware should be downloaded to a port.

6.1 call_restart_fmw() - Restart signalling system firmware

This function is used to restart or change the configuration of the signalling system firmware on a port on an Aculab card without having to reboot the machine. The function will cause the selected port processor on the card to be reset and the firmware to be downloaded. The `call_restart_fmw()` function may be used to effect the initial firmware download.

Synopsis

```
ACU_ERR call_restart_fmw(RESTART_XPARMS *restartp);

typedef struct restart_xparms
{
  ACU_ULONG           size;              /* IN */
  ACU_PORT_ID         net;               /* IN */
  ACU_CHAR            *filenamep;        /* IN */
  ACU_CHAR            *config_stringp;   /* IN */
  ACU_CHAR            *dspa_filenamep;   /* Not used */
  ACU_CHAR            *dspb_filenamep;   /* Not used */
} RESTART_XPARMS;
```

Input parameters

The `call_restart_fmw()` function takes a pointer, *restartp*, to a structure,`RESTART_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

net

The *net* field must contain the network *port_id* on the card on which the signalling system firmware is to be restarted, as returned from `call_open_port()`.

filenamep

The *filenamep* field must contain a pointer to a null terminated ASCII string containing the name of the firmware file to be downloaded including any path or drive symbol information required to access that file.

config_stringp

The *config_stringp* field must contain a pointer to a null terminated ASCII string of configuration information for the signalling system. This information is described in the Call, Switch and Speech Installation Guide.

For example:

```
struct restart_xparms rxp;

rxp.config_stringp = "-cNE -cD4";
```

NOTE

If no configuration information is provided, then config_stringp must contain a pointer to an empty string. Only Call driver configuration options may be used in this field. Any hardware configuration options must be already configured in the appropriate way for the system.

Return values

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

The `call_restart_fmw()` function will reset the processor and hardware on the specified port only, none of the other ports will be affected. The configuration switches will be read prior to starting the firmware. The file presented for downloading will then be read by the library, verified that it is a firmware file, transferred to the Aculab card and executed by the on board monitor. The driver will check the firmware

to ensure that it can be supported and the device driver will then register it for use.

NOTE

Before restarting the firmware ensure that:

- there are no calls in progress
- there are no open call handles
- there are no processes blocked in the device driver

NOTE

After restarting:

- the network port must be re-initialised
- if the network was the clock reference for the system, the clock reference must be re-established either from the network or the MVIP|\SC-Bus\H100 bus.

It is recommended that the restart procedure be used to change modes of operation of the signalling system and not as a method of restarting a ‘runaway’ application.

6.2 call_reconfig_fmw () – reconfigure the signalling system firmware

This function is used to reconfigure the –s switches for the signalling system firmware without restarting the firmware. This function is only supported for these signalling systems – EuroISDN, QSIG, NI-2, AT&T T1, CAS (T1RB) and ISUP/SS7.

```
ACU_ERR call_reconfig_fmw (RECONFIG_XPARMS *reconfig)

typedef struct reconfig_xparms
{
   ACU_ULONG      size;
   ACU_PORT_ID    net;
   ACU_CHAR*      switch_stringp;
} RECONFIG_XPARMS;
```

Description

This function call can be used to reconfigure firmware parameters (-s switches). Except for ISUP only the the –s99,n switch is supported.

Input Parameters

net

This field must contain the network *port_id* on the card for which the firmware is to be reconfigured.

switch_stringp

A null terminated string holding the firmware download switches to be reconfigured.

NOTE

Except for ISUP/SS7 only the –s99,xx switch is supported and the string returned by call_get_fmw_dl_parms() is not updated (See the firmware release notes for a description of this switch).

NOTE

ISUP/SS7 allows ISUP circuits and MTP2 signalling links be added or removed, call_get_fmw_dl_params() will return an updated string.
Layer 1 parameters cannot be changed.
Refer to the SS7 Installation and Admin guide for further information.

Example:

```
RECONFIG_XPARMS reconfig;
char switch_string[15];
ACU_ERR result;

strcpy(switch_string, "-s99,224");  /* -s99,224 to turn trace on / -s99,0 to
turn trace off */
INIT_ACU_CL_STRUCT (&reconfig);
reconfig.net = port_id;
reconfig.switch_stringp = switch_string;
result = call_reconfig_fmw(&reconfig);
```

6.3 call_stop_fmw()

This function is used to stop the firmware on a port.

For cards that can be controlled by multiple hosts, this function will relinquish control of the port so that another host can use it.

NOTE

It is up to the developer to synchronise access to different resources between applications running on different hosts.

Synopsis

```
ACU_ERR call_stop_fmw(CALL_STOP_FMW_PARMS* stop_parms);

typedef struct tCALL_STOP_FMW_PARMS
{
  ACU_ULONG           size;
  ACU_PORT_ID         port_id;
} CALL_STOP_FMW_PARMS;
```

Input parameters

The `call_stop_fmw()` function takes a pointer, *`stop_parms`*, to a structure,`CALL_STOP_FMW_ PARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`size`
This is the size of the structure. Use `INIT_ACU_CL_STRUCT()` to initialise this field.

`port_id`
This is the port_id of the port to stop.

Return values

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

6.4 call_is_download() - Check if network port requires firmware download

This function may be used to ascertain whether signalling system firmware should be downloaded to a port.

NOTE

Not supported for IP Telephony. For more information on the status of board firmware on an IP telephony card refer to the IP Telephony Card V6 API Guide.

Synopsis

```
ACU_ERR call_is_download(ACU_PORT_ID portnum);
```

Input parameters

The input parameter *portnum* must contain the number of the network port on the card on for which the enquiry is being made and will have a valid port id as returned from `call_open_port()`.

Return values

`0`	The value zero will be returned if NO firmware download is required.
`1`	The value 1 will be returned if signalling system firmware must be downloaded to the specified network port.
`-ve`	A negative value will be returned if an error has occurred.

7 Diagnostic Functions

This section describes functions that may be used for diagnostic purposes. The functions provide an overview of how the system is behaving so that any problems that may occur can be readily diagnosed.

API	Description
`call_dcba()`	Used to read the CAS ABCD signalling bits on the network interface. The signalling bits present on the network port are not affected by the use of this function.
`call_protocol_trace()`	All signalling systems pass protocol information to the device driver. This function allows the application to recover the protocol information form the Aculab card for display or diagnostic purposes.
`call_l1_stats()`	Used to obtain the current condition of Layer 1 of the signalling system and allows an application to inspect and clear line condition flags and counters.
`call_l2_state()`	Used to obtain the current condition of Layer 2 of the signalling system. This may be useful when installing an application on site. Used in conjunction with the layer 1 statistics, this function will give indication of whether the Aculab card is correctly configured to the exchange.
`call_start_trace()`	Starts the trace collection on a port identified by a port_id supplied by the user.
`call_stop_trace()`	Stops the collection of trace for a given `port_id`.
`call_set_trace_mode()`	Allows the user to change the tracing parameter whilst trace is being collected. This works on a per port basis.

7.1 call_dcba() - Reading the CAS ABCD bits

This function may be used to read the CAS ABCD signalling bits on the network interface. The signalling bits present on the network port are not affected by the use of this function.

Synopsis

```
ACU_ERR call_dcba(DCBA_XPARMS *dcbap);

typedef struct dcba_xparms
{
  ACU_ULONG      size;          /* IN */
  ACU_PORT_ID    net;           /* IN */
  ACU_UCHAR      tdcba[16];     /* OUT */
  ACU_UCHAR      rdcba[16];     /* OUT */
} DCBA_XPARMS;
```

Input parameters

The `call_dcba()` function takes a pointer, *`dcbap`*, to a structure,`DCBA_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`net`

The *`net`* field must contain the network `port id` on the card from which the abcd bits are to be obtained, as returned from `call_open_port()`.

Return values

`tdcba`

The character array *`tdcba`* will contain the ABCD bits being transmitted by the card at the time of the function call.

`rdcba`

The character array *`rdcba`* will contain the ABCD bits being received by the card at the time of the function call.

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

NOTE

The information returned by call_dcba() is the current state of the ABCD bits. It is therefore possible for the application to miss rapid transitions of any of the signaling bits.

Use of `call_dcba()` with a ISDN protocol is allowed but will not result in any meaningful information being returned.

NOTE

For PMX and PCIe based hardware, call_dcba() support is enabled using a firmware configuration switch. For performance reasons, it is recommended that call_dcba() is only enabled for debug purposes. Please refer to the firmware release notes for further clarification.

Format of the tdcba and rdcba arrays

Each element of the transmit and receive arrays (tdcba and rdcba respectively) contains information about two timeslots. . The way each element is formatted depends on a network port's hardware and line type.

The format of the transmit and receive arrays are as follows:

tdcba array for all hardware, with any line type.

Element	Bit Position and Timeslot (TS)	
	7654 TS	3210 TS
tdcba[0]	dcba 16	dcba 0
tdcba[1]	dcba 17	dcba 1
tdcba[2]	dcba 18	dcba 2
tdcba[3]	dcba 19	dcba 3
•	•	•
•	•	•
tdcba[12]	dcba 28	dcba 12
tdcba[13]	dcba 29	dcba 13
tdcba[14]	dcba 30	dcba 14
tdcba[15]	dcba 31	dcba 15

rdcba array for all remaining hardware, with any line type.

Element	Bit Position and Timeslot (TS)	
	7654 TS	3210 TS
rdcba[0]	abcd 0	abcd 16
rdcba[1]	abcd 1	abcd 17
rdcba[2]	abcd 2	abcd 18
rdcba[3]	abcd 3	abcd 19
•	•	•
•	•	•
rdcba[12]	abcd 12	abcd 28
rdcba[13]	abcd 13	abcd 29
rdcba[14]	abcd 14	abcd 30
rdcba[15]	abcd 15	abcd 31

7.2 call_protocol_trace() - Obtaining protocol information

All signalling systems pass protocol information to the device driver. This function allows the application to recover the protocol information form the Aculab card for display or diagnostic purposes.

NOTE

This function is not supported for IP Telephony.

Synopsis

```
ACU_ERR call_protocol_trace(LOG_XPARMS *logp);

typedef struct log_xparms
{
  ACU_ULONG      size;          /* IN */
  ACU_PORT_ID    net;           /* IN */
  struct log     log;           /* OUT */
} LOG_XPARMS;

typedef struct log
{
  ACU_LONG            TimeStamp;          /* OUT */
  ACU_UCHAR           RxTx;               /* OUT */
  ACU_UCHAR           Data_Packet[75];    /* OUT */
} LOG;
```

Input parameters

The `call_protocol_trace()` function takes a pointer, *`logp`*, to a structure,`LOG_XPARMS`. The structure must be initialised before invoking the function, (see section 2.2).

`net`

The *`net`* field must contain the network *`port_id`* on the card from which the protocol information is to be obtained, as returned from `call_open_port()`.

Return values

`log`

The *`log`* field is a structure containing the protocol information and contains the following:

`TimeStamp`

The return value *`TimeStamp`* contains a time value in 5ms increments relative to when the protocol was started on the Aculab card. With PMX based hardware the increment is 1ms instead of 5ms.

`RxTx`

The field *`RxTx`* contains a Boolean value that indicates the direction of the protocol information, that is whether the message was received or transmitted.

`RxTx = 0`	Receive
`RxTx = 1`	Transmit

`Data_Packet`

The return value *`Data_Packet`* is an array containing the protocol message.

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

Format of Data_Packet

For ISDN protocols `Data_Packet` contains the actual message received or transmitted from Layer 2 of the protocol stack in hexadecimal format, where:

`Data_Packet[0]` = number of bytes following
`Data_Packet[1..74]` = protocol information

If there is no protocol trace available then:

`Data_Packet[0] == 0` and `TimeStamp` and `RxTx` are not valid.

For the CAS protocols `Data_Packet` contains null terminated ASCII text fabricated by the CAS protocol running on the Aculab card. This text contains the `abcd` bits and state transitions that have occurred in the CAS signalling system and will only be available if the **–s99** switch has been set in the device driver (see the release notes for CAS variants).

If there is no data available then:

`Data_Packet[0] == 0` and `TimeStamp` and `RxTx` are not valid.

NOTE

The actual contents of Data_Packet is not documented, is entirely dependent upon the protocol being examined at the time and interpretation should be left to the Aculab Support Department.

7.3 call_l1_stats() - Layer 1, statistics

This function may be used to obtain the current condition of Layer 1 of the signalling system and allows an application to inspect and clear line condition flags and counters.

NOTE

This function is not supported for IP Telephony.

Synopsis

```
ACU_ERR call_l1_stats(L1_XSTATS *l1_statsp);
typedef struct l1_xstats
{
  ACU_ULONG      size;
  ACU_PORT_ID    net;
  struct getset  getset;
  struct get     get;
} L1_XSTATS;

typedef struct getset
{
  ACU_INT             linestat;
  ACU_INT             bipvios;
  ACU_INT             faserrs;
  ACU_INT             sliperrs;
} GETSET;

typedef struct get
{
  ACU_UCHAR      nos;
  ACU_UCHAR      ais;
  ACU_UCHAR      los;
  ACU_UCHAR      rra;
  ACU_UCHAR      tra;
  ACU_UCHAR      rma;
  ACU_UCHAR      tma;
  ACU_UCHAR      usr;
  ACU_UCHAR      majorrev;
  ACU_UCHAR      minorrev;
  ACU_ULONG      clock;
  char           buildstr[16];
  char           manstr[16];
  char           sigstr[16];
} GET;
```

Input parameters

The `call_l1_stats()` function takes a pointer, *`l1_statsp`*, to a structure,`L1_XSTATS`. The structure must be initialised before invoking the function, (see section 2.2).

`net`

The *`net`* field must contain the network *`port_id`* on the Aculab card on which the layer 1 statistics are to be obtained, as returned from `call_open_port()`.

`l1_xstats`

The *`l1_xstats`* structure contains two further structures, the *`getset`* structure and the *`get`* structure.

Return value

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

Memset the whole structure to zero to return the `getset` and `get` structure parameters.

NOTE

Setting the getset values to 0xFFFF resets the layer 1 stats.

The getset structure

This structure contains layer 1 information that may be read and modified by the application. So information bits and counter values may be obtained and reset once they have been read.

For any element in the *getset* structure the driver performs the following operations. The layer 1 value will be ANDed with the bitwise inverse of the value supplied in the *getset* structure, then the value will be returned.

```
inestat &= ~l1_stats.getset.linestat;
l1_stats.getset.linestat = linestat;
```

Therefore, to read a layer 1 value without modifying its contents, the element of the getset structure should be set to zero prior to invoking the function.

```
l1_stats.getset.linestat = 0;
```

To reset a layer 1 value the element of the getset structure should be set to all 1 bits prior to invoking the function.

```
l1_stats.getset.linestat = ~0;
```

linestat

This location provides several line status indication flags as follows:

Name	Description
LSTAT_NOS	Lost Signal Flag
LSTAT_LOS	Lost Synchronisation Flag
LSTAT_AIS	Alarm in Service
LSTAT_FEC	Error Rate > 1 in 1000 Flag
LSTAT_RRA	Alarm from Remote End
LSTAT_SLP	Frame Slip in Receive Buffer Flag
LSTAT_CVC	Bipolar Code Violation Flag
LSTAT_CRC	CRC error
LSTAT_FFA	False Frame Alignment
LSTAT_CML	CAS Multiframe Lost

Each of these status bits are set and latched when detected by the card and are updated as and when they occur.

Each bit may be individually reset. For example, to reset the LSTAT_NOS bit:

```
l1_stats.getset.linestat = LSTAT_NOS;
```

Only the LSTAT_NOS bit will be reset and all other bits will be returned unaffected.

bipvios- Bipolar Violations

bipvios counts the occurrences of bipolar violations on the receive line from the network. Frequent occurrences of bipolar violations are usually an indication of poor line quality, and will often indicate a degree of corruption of B-channel data. This location may be reset at any time to restart the count.

To GET the bipvios count:

```
l1_stats.getset.bipvios = 0;
```

To RESET the bipvios count:

```
l1_stats.getset.bipvios = ~0;
```

faserrs - Frame Alignment Signal Errors

faserrs provides a count of the number frames that include at least 1 bit in error in the Frame Alignment Signal (FAS). This location may be reset at any time to restart the count.

To GET the faserrs count:

```
l1_stats.getset.faserrs = 0;
```

To RESET the faserrs count:

```
l1_stats.getset.faserrs = ~0;
```

sliperrs - Frame Slip Errors

sliperrs provides a count of the number of received frame slips in the incoming G703 signal.

NOTE

If frame slips are detected, it usually indicates that there are problems with the configuration. These must be corrected.

If a network connected port is correctly deriving it's clocking from the network frame slips should not occur. Frame slips can occur when the clock control setup is incorrect or inappropriate.

For example, if clocking is set to be derived from another card on the MVIP, SC Bus or H100 bus, and that card is disconnected, out of service, or not correctly initialised, then frame slips are likely to occur.

To GET the sliperrs count:

```
l1_stats.getset.sliperrs = 0;
```

To RESET the sliperrs count:

```
l1_stats.getset.sliperrs = ~0;
```

The get structure

This structure contains layer 1 information that may be read by the application. The elements in the 'get' structure may assume any value prior to invoking the function.

nos
Semaphore indicating the occurrence of the Lost Signal condition. The semaphore will be set to 0xff on condition and will persist as long as the condition.

ais
Semaphore indicating the occurrence of the Incoming Alarm Indication condition. The semaphore will be set to 0xff on condition and will persist as long as the condition.

los
Semaphore indicating the occurrence of the Lost Synchronisation condition. The semaphore will be set to 0xff on condition and will persist as long as the condition.

rra
Semaphore indicating the occurrence of the Receive Remote Alarm condition. The semaphore will be set to 0xff on condition and will persist as long as the condition.

tra
Semaphore indicating the occurrence of the Transmit Remote Alarm condition. The semaphore will be set to `0xff` on condition and will persist as long as the condition. It should be noted that the `tra` semaphore might be set by either Layer 1 of the protocol or because of the `call_send_alarm()` function.

rma
Semaphore to indicate the receipt of the Receive Multi-Frame alarm. The semaphore will be set to `0xff` on condition and will persist as long as the condition persists. This is currently only supported in the CAS protocols.

tma
Semaphore to indicate the occurrence of the Transmit Multi-Frame alarm. The semaphore will be set to `0xff` on condition and will persist as long as the condition persists.

It should be noted that the `tma` semaphore might be set by either Layer 1 of the protocol or because of the `call_send_alarm()` function. This is currently only supported in the CAS protocols.

usr
Provides fault conditions documented under CCITT I431.

Condition `0xf1` - operational – primary rate

Condition `0xf3` - network side – basic rate

Condition `0xf7` - user side – basic rate

majorrev, minorrev
Major and Minor Revision These locations will receive the value of the signalling software major and minor revision numbers once the port signalling software has initialised.

buildstr, manstr, sigstr
These locations provide extra information regarding the version of the firmware, once the port signalling software has been initialised.

7.4 call_l2_state() - Layer 2 State

This function may be used to obtain the current condition of Layer 2 of the signalling system. This may be useful when installing an application on site. Used in conjunction with the layer 1 statistics, this function will give indication of whether the Aculab card is correctly configured to the exchange.

NOTE

This function is not supported for SIP.

Synopsis

```
ACU_ERR call_l2_state(L2_XSTATE *l2_statep);

typedef struct l2_xstate
{
  ACU_ULONG      size;          /* IN */
  ACU_PORT_ID    net;           /* IN */
  ACU_LONG       state;         /* OUT */
} L2_XSTATE;
```

Input parameters

The `call_l2_state()` function takes a pointer, `l2_statep`, to a structure,`L2_XSTATE`. The structure must be initialised before invoking the function, (see section 2.2).

`net`

The *`net`* field must contain the network `port id` on the card for which the layer 2 state is to be obtained, as returned from `call_open_port()`.

Return values

`state`

The *`state`* field is a bit field, each bit indicating the state of the timeslots at layer 2. Bit 0 of `state` indicates the condition of layer 2 for timeslot 0 while bit 31 of state indicates the condition of layer 2 for timeslot 31. A bit set to '1' indicates that Layer 2 for that timeslot is operational, while a bit cleared to '0' indicates the non-operation of that timeslot.

NOTE

For CAS signaling systems the state bit field indicates the condition of the 'far end' of the signaling system. A bit reset to 0 indicates that the 'far end' is sending the BUSY pattern so will not receive incoming calls. A bit set to 1 indicates that the 'far end' will accept incoming calls.

NOTE

For H.323 signalling systems the returned bit field will either be all '0's to indicate that the system is not registered with a gatekeeper, or all '1's to indicate that it is currently registered.

NOTE

When using the AT&T protocol the state bitmap will reveal changes in the Service state. The bitmap will now show which channels have been blocked or unblocked. Receipt of an ACU_CALL_EVT_BLOCKING_STATE_CHANGE event will prompt the application to check the current state. This behaviour must be enabled with the appropriate AT&T firmware configuration (-s switch); see the AT&T release notes for more details. If it is enabled then the bitmap will set bits to 1 when the timeslot is in-service and when L2 is also active. Otherwise, the function will only report the current L2 state.

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

7.5 call_start_trace()

Starts the trace collection on a port identified by a port_id supplied by the user. The flags parameter sets the type of tracing that is required. This may be altered mid trace using `call_set_trace_mode();`

The filename parameter is optional. If specified, the trace for the given port will go to a file named filename. Otherwise the trace will be written to `STDOUT`.

Synopsis

```
ACU_ERR call_start_ trace( START_TRACE_PARMS* start_trace_info );

typedef struct _START_TRACE_PARMS
{
  ACU_ULONG      size                    /* IN */
  ACU_PORT_ID    port_id;                /* IN */
  ACU_UINT       flags;                  /* IN */
  ACU_UINT       extra_flags             /* IN */
  ACU_UINT       delay;                  /* IN */
  ACU_CHAR       filename[MAX_LENGTH];   /* IN */
} START_TRACE_PARMS;
```

Input parameters

`port_id`

This is the `port_id` returned by `call_open_port()`. It sets the port that is to be traced. Multiple ports can be traced by calling this function multiple times.

`flags`

Trace flags define the type of trace to be collected. These are defined in the header file under `TRACE FLAGS`.

`extra_flags`

Protocol specific flags for trace.

`delay`

This can only be set by the first port set to collect trace. It controls the length between trace collections. If this is left set to 0, the default will be 100 ms.

`filename`

The name of the output file. This will divert the trace from stdout to a file. A file will be created in the application's working directory. If the file doesn't exist it will be created, otherwise, new trace will be appended to the the existing file.

Returns

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

7.6 call_stop_trace()

Stops the collection of trace for a given `port_id`. The port is indicated by `port_id` and is a valid port ID as returned by a call to `call_open_port()`.

Stopping trace collection on an IP port (SIP & H.323) results in the trace mode being set to zero. This means that the circular buffer used to collect trace is emptied. If trace is started again, only trace from the point at which trace collection was restarted is collected.

Stopping trace collection on a non-IP port will retain the trace buffer. When trace is started again, the new trace will include any previous trace that has been retained in the buffer.

Synopsis

```
ACU_ERR call_stop_trace( ACU_PORT_ID);
```

Input parameters

`port_id`
This is a `port_id` currently being traced, which must be a valid port as returned from a call to `call_open_port()`.

Returns

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

7.7 call_set_trace_mode()

Allows the user to change the tracing parameter whilst trace is being collected. This works on a per port basis.

Synopsis

```
ACU_ERR call_set_trace_mode(TRACE_MODE_PARMS *trace_mode_info);

typedef struct _TRACE_MODE_PARMS
{
  ACU_ULONG      size              /* IN */
  ACU_PORT_ID    port_id;          /* IN */
  ACU_UNIT       flags;            /* IN */
  ACU_UINT       extra_flags       /* IN */
} TRACE_MODE_PARMS;
```

Input parameters

port_id

this is the port_id return from a call to call_open_port(). It sets the port that is to be traced. Multiple ports can be traced by calling this function multiple times.

flags

Trace flags define the type of trace to be collected. No flags are set default.

TRACE_API_CALL	: Trace API
TRACE_PROTOCOL	: Trace protocol
TRACE_EVENT	: Trace internal event generation information
TRACE_HW	: Trace hardware specific information
TRACE_RESOURCE	: Trace resource allocation
TRACE_GENERIC	: Trace generic driver internals
TRACE_LEVEL_LOW	: Trace only essential information
TRACE_LEVEL_MEDIUM	: Trace more detailed information
TRACE_LEVEL_HIGH	: Trace very detailed information
TRACE_TLS	: Trace the thin layer

extra_flags

Protocol specific flags for trace.

Returns

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

8 unique_xparms

The parameter *unique_xparms* is a union that provides extensions required by specific signalling systems. The union will vary depending upon the signalling system supported by the device driver and cards.

```
typedef union uniquex
{
  UNIQUEX_DASS      sig_dass;
  UNIQUEX_DPNSS     sig_dpnss;
  UNIQUEX_CAS       sig_cas;
  UNIQUEX_Q931      sig_q931;
  UNIQUEX_ISUP      sig_isup;
  UNIQUEX_IPTEL     sig_iptel; /* #include iptel_lib.h and iptel_lib.lib */
} UNIQUEXU;
```

8.1 unique_xparms for Q931

```
typedef struct uniquex_q931
{
  ACU_UCHAR      service_octet;
  ACU_UCHAR      add_info_octet;
  ACU_UCHAR      dest_numbering_type;
  ACU_UCHAR      dest_numbering_plan;
  struct
  {
    ACU_UCHAR         ie[MAXBEARER];
    ACU_UCHAR         last_msg;
  } bearer;
  ACU_UCHAR      orig_numbering_type;
  ACU_UCHAR      orig_numbering_plan;
  ACU_UCHAR      orig_numbering_presentation;
  ACU_UCHAR      orig_numbering_screening;
  ACU_UCHAR      conn_numbering_type;
  ACU_UCHAR      conn_numbering_plan;
  ACU_UCHAR      conn_numbering_presentation;
  ACU_UCHAR      conn_numbering_screening;
  ACU_UCHAR      dest_subaddr[MAXNUM];
  ACU_UCHAR      orig_subaddr[MAXNUM];
  struct
  {
    ACU_UCHAR         ie[MAXHILAYER];
    ACU_UCHAR         last_msg;
  } hilayer;
  struct
  {
    ACU_UCHAR         ie[MAXLOLAYER];
    ACU_UCHAR         last_msg;
  } lolayer;
  struct
  {
    ACU_UCHAR         ie[MAXPROGRESS];
    ACU_UCHAR         last_msg;
  } progress_indicator;
  struct
  {
    ACU_UCHAR         ie[MAXNOTIFY];
    ACU_UCHAR         last_msg;
  } notify_indicator;
  struct
  {
    ACU_UCHAR         ie[MAXNUM];
    ACU_UCHAR         last_msg;
  } keypad;
  struct
  {
    ACU_UCHAR         ie[MAXDISPLAY];
    ACU_UCHAR         last_msg;
```

```
  } display;
  ACU_LONG       slotmap;
  EP             endpoint_id;
  ACU_UCHAR      oli_ani_2;
  struct
  {
    ACU_UCHAR ie[MAXCAUSE];
    ACU_UCHAR last_msg;
  } cause;
  ACU_UCHAR      additional_orig_addr[MAXADDR];
  ACU_UCHAR      add_orig_numbering_type;
  ACU_UCHAR      add_orig_numbering_plan;
  ACU_UCHAR      add_orig_numbering_presentation;
  ACU_UCHAR      add_orig_numbering_screening;
  ACU_UCHAR      omit_calling_party_ie;
  ACU_USHORT     call_ref_value;
  ACU_UCHAR      conn_subaddr[MAXNUM];
} UNIQUEX_Q931;
```

Q931 Specific Information

The Q931 structure, `sig_q931`, must be initialised in the following way before invoking the function.

service_octet

Must contain a valid service indicator for the call and must be appropriate for the signalling system, for example,

```
UNKNOWN_SERVICE
TELEPHONY
ABSERVICE
X21SERVICE
FAXGP4
VIDEO64K
DATA64K
X25SERVICE
TELETEXT64
MIXEDMODE
TELEACTION
GRAPHIC
VIDEOTEXT
VIDEOPHONE
```

`add_info_octet`

Must contain additional information about the service required by the application and must be related to the *`service_octet`* and appropriate for the signalling system, for example,

For `TELEPHONY`

```
ISDN_3K1
ANALOGUE
ISDN_7K
```

For `ABSERVICE`

```
FAXGP2
FAXGP3
```

For `X21SERVICE`

```
UC4
UC5
UC6
UC19
```

For `X25SERVICE`

```
UC8
UC9
UC10
```

```
UC11
UC13
UC19K2
```

For VIDEOPHONE

```
SOUND_3K1
SOUND_7K
IMAGE
```

If the *service_octet* and *add_info_octet* both contain zero, the device driver will assume default values of:

```
service_octet = TELEPHONY
add_info_octet = ANALOGUE
```

dest_numbering_type

Can contain a valid numbering type for the call and must be appropriate for the signalling system. Allowable values are:

```
NT_UNKNOWN
NT_INTERNATIONAL
NT_NATIONAL
NT_NETWORK_SPECIFIC
NT_SUBSCRIBER_NUMBER
NT_ABBREVIATED_NUMBER
```

NOTE

Some signaling systems may not have support for all these values.

The default value is NT_UNKNOWN.

dest_numbering_Plan

Can contain a valid numbering plan for the call and must be appropriate for the signalling system. For example, allowable values may include:

```
NP_UNKNOWN
NP_ISDN
NP_DATA
NP_TELEX
NP_NATIONAL_STANDARD
NP_PRIVATE
```

NOTE

Some signaling systems may not have support for all these values.

The default value is NT_UNKNOWN.

bearer

The *bearer* field is used to provide 'bearer code' information to the protocol. The format of the bearer code is dependent upon the signalling system and reference should be made to the appropriate specification for the protocol. (MAXBEARER = 16)

For those signalling systems that support bearer codes, the service octet and bearer code convey the same information in different forms and the application may use either method. The device driver makes the following assumption when presented with a bearer code and service octet.

```
if bearer.ie[0] == 0 use Service Octet

if bearer.ie[0] != 0 then use bearer code.
```

If you do not wish to use the bearer code then ensure that the bearer field is empty.

```
bearer.ie[0] = 0
```

orig_numbering_type
Values as detailed in *dest_numbering_type* above.

orig_numbering_plan
Values as detailed in *dest_numbering_plan* above.

orig_numbering_presentation
The *orig_numbering_presentation* field indicates the intention of the calling party for the presentation of the calling number (*originating_address*) to the called party.

The four values are:

```
PR_ALLOWED (Presentation Allowed)
PR_RESTRICTED (Presentation Restricted)
PR_NOTAVAILABLE (Number not available due to interworking)
PR_RESERVED
```

orig_numbering_screening
The possible values for this field are:

SC_NOTSCREENED (User provided, not screened)
SC_VERIFYPASS* (User provided, verified and passed)
SC_VERIFYFAIL (User provided, verified and failed)
SC_NETWORKPROVIDED* (network provided)

conn_numbering_type (not applicable for an outgoing call)
This field must contain a valid numbering type for the call and be appropriate for the signalling system. Allowable values are:

```
NT_UNKNOWN
NT_INTERNATIONAL
NT_NATIONAL
NT_NETWORK_SPECIFIC
NT_SUBSCRIBER_NUMBER
NT_ABBREVIATED_NUMBER
```

Some signalling systems may not have support for all these values.

The default value is NT_UNKNOWN.

conn_numbering_plan (not applicable for an outgoing call)
This field must contain a valid numbering plan for the call and be appropriate for the signalling system. For example, allowable values may include:

```
NP_UNKNOWN
NP_ISDN
NP_DATA
NP_TELEX
NP_NATIONAL_STANDARD
NP_PRIVATE
```

Some signalling systems may not have support for all these values.

The default value is NP_UNKNOWN for Q.931 and NP_ISDN for ISUP.

conn_numbering_presentation (not applicable for an outgoing call)
This input parameter indicates the intention of the calling party for the presentation of the calling number (*connected_address*) to the called party.

The four values are:

```
PR_ALLOWED (Presentation Allowed)
PR_RESTRICTED (Presentation Restricted)
PR_NOTAVAILABLE (Number not available due to interworking)
PR_RESERVED
```

conn_numbering_screening
Is not applicable for an outgoing call. The possible values for this field are:

SC_NOTSCREENED (User provided, not screened)
SC_VERIFYPASS* (User provided, verified and passed)
SC_VERIFYFAIL (User provided, verified and failed)
SC_NETWORKPROVIDED* (network provided)

dest_subaddr
Maximum length MAXNUM using digits 0 to 9. See Appendix F: for further details.

orig_subaddr
Maximum length MAXNUM using digits 0 to 9. See Appendix F: for further details.

hilayer & lolayer
By using the *hilayer* and *lolayer* fields in association with *bearer*, the call type may be specified to the remote user. The format of hilayer and lolayer is dependent upon the signalling system and reference should be made to the appropriate specification for the protocol.

The *ie* field of the *hilayer* structure should contain the high layer compatibility information that is sent transparently to the other end. The *last_msg* field of the *hilayer* structure should be set to zero. (MAXHILAYER = 16)

The *ie* field of the *lolayer* structure should contain the low layer compatibility information that is sent transparently to the other end. The *last_msg* field of the *lolayer* structure should be set to zero.(MAXLOLAYER = 16)

progress_indicator
The *progress_indicator* field can be used to indicate events pertaining to the call regarding interworking or in-band information. The *ie* field of the *progress_indicator* structure should contain the progress information that will be transmitted unmodified by the drivers. The *last_msg* field should be set to zero. This information is dependent on the signalling system and reference should be made to the appropriate specification for the protocol. (MAXPROGRESS = 4)

notify_indicator (not applicable for an outgoing call)

(MAXNOTIFY = 4)

The *notify_indicator* field will provide valid notify information to the application. The ie field of the *notify_indicator* structure will contain the notify information. This information is dependent on the signalling specification and reference should be made to the appropriate specification for the protocol. The *last_msg* field in the *notify_indicator* structure will contain the value of the message that delivered the last progress information element.

keypad
The *keypad* field can be used to transmit IA5 characters (0 to9) that may be used to control supplementary services. (MAXNUM = 32)

display
The *display* field can be used to transmit information that may be displayed by the user. (MAXDISPLAY = 34)

slotmap

A 32 bit string used to indicate which timeslots to use for an outgoing call. The following example will make a call utilising timeslots 12 and 19.

```
31                                  0
0000 0000 0000 1000 0001 0000 0000 0000
```

NOTE

Avoid using signalling channels, for example timeslot 16 for E1.

endpoint_id
A gatekeeper assigned token supplied in the RCF message identifying the endpoint and should be returned in all future communication with the gatekeeper while the system remains registered.

Cause **ie**
holds the raw coding as supplied by the signalling system with the first element as the length field.

last_msg
shows the last message type the cause received, which can be one of :

```
Q931_PROGRESS
Q931_DISCONNECT
Q931_RELEASE
Q931_RELEASE_CMPL
```

NOTE

The last_msg field should not be used when sending information

additional_orig_addr
The input character buffer *additional_orig_addr* can be supplied with a null terminated string of IA5 digits. This string represents an additional originating subscriber number. The string can be passed to the signalling system when the outgoing call is made, as required, on a per call basis.

add_orig_numbering_type;
Can contain a valid numbering type for the call and must be appropriate for the signalling system. Allowable values are:

```
NT_UNKNOWN
NT_INTERNATIONAL
NT_NATIONAL
NT_NETWORK_SPECIFIC
NT_SUBSCRIBER_NUMBER
NT_ABBREVIATED_NUMBER
```

NOTE

Some signaling systems may not have support for all these values.

The default value is `NT_UNKNOWN`.

add_orig_numbering_plan;
Can contain a valid numbering plan for the call and must be appropriate for the signalling system. For example, allowable values may include:

```
NP_UNKNOWN
NP_ISDN
NP_DATA
NP_TELEX
NP_NATIONAL_STANDARD
NP_PRIVATE
```

The default value is `NT_UNKNOWN`.

add_orig_numbering_presentation;
The *add_orig_numbering_presentation* field indicates the intention of the calling party for the presentation of the additional calling number (*additional_orig_addr*) to the called party.

The four values are:

```
PR_ALLOWED (Presentation Allowed)
PR_RESTRICTED (Presentation Restricted)
PR_NOTAVAILABLE (Number not available due to interworking)
PR_RESERVED
```

add_orig_numbering_screening;
The possible values for this field are:

SC_NOTSCREENED (User provided, not screened)
SC_VERIFYPASS* (User provided, verified and passed)
SC_VERIFYFAIL (User provided, verified and failed)
SC_NETWORKPROVIDED* (network provided)

omit_calling_party_ie;
Reserved for future use.

call_ref_value;
A unique value assigned by the originating side of a call, which may be the same value as the call handle.

The purpose of the call reference is to be able to identify a call message, for example, a call facility registration or cancellation request message.

conn_subaddr
Maximum length MAXNUM using digits 0 to 9. See Appendix F: for further details.

8.2 unique_xparms for DASS2

```
typedef struct uniquex_dass
{
  ACU_UCHAR      sic1;
  ACU_UCHAR      sic2;
} UNIQUEX_DASS;
```

DASS structure

The DASS structure, *sig_dass*, must be initialised in the following way before invoking the function.

The *sic1* and *sic2* fields must contain information about the service required by the application and must be appropriate for the signalling system. If both *sic1* and *sic2* contain zero, then the driver will allocate a default value of speech.

sic1

Digital Access Signalling System N0 2 (DASS2) service indictor codes are octets, sic1 and sic2. The sic1 octet is structured as follows:

Bits 1 to 4

These bits specify either the speech characteristics, for example, A-Law 64Kbit/s, or the data rate.

Bits 5-7

These bits specify the type of information and include:

```
Speech
Data
Teletex
Videotex
Facsimile
SSTV
```

Bit 8

Indicates if there is a second (further) octet;

```
0 = no further octet
1 = further octet
```

sic2

Digital Access Signalling System N0 2 (DASS2) servcie indictor codes are octets, sic1 and sic2. The sic2 octet is structured as follows:

Bits 1 to 3

These bits specify either the synchronous or asynchronous modes of operation.

Bit 4

Duplex Mode:

```
0 = Full duplex
1 = Half duplex
```

Bits 5-7

These bits specify the data format, clock and flow control parameters

Bit 8

Indicates there are no further octets, always set to 0.

8.3 unique_xparms for DPNSS

```
typedef struct uniquex_dpnss
{
  ACU_UCHAR      sic1;
  ACU_UCHAR      sic2;
  ACU_UCHAR      clc[MAXCLC];
} UNIQUEX_DPNSS;
```

DPNSS structure

The DPNSS structure, *sig_dpnss*, must be initialised in the following way before invoking the function.

The *sic1* and *sic2* fields must contain information about the service required by the application and must be appropriate for the signalling system. If both *sic1* and *sic2* contain zero, then the driver will allocate a default value of speech.

sic1

Digital Access Signalling System N0 2 (DASS2) service indictor codes are octets, sic1 and sic2. The sic1 octet is structured as follows:

Bits 1 to 4

These bits specify either the speech characteristics, for example, A-Law 64Kbit/s, or the data rate.

Bits 5-7

These bits specify the type of information and include:

```
Speech
Data
```

Bit 8

Indicates if there is a second (further) octet;

```
0 = no further octet
1 = further octet
```

sic2

Digital Access Signalling System N0 2 (DASS2) servcie indictor codes are octets, sic1 and sic2. The sic2 octet is structured as follows:

Bits 1 to 3

These bits specify either the synchronous or asynchronous modes of operation.

Bit 4

Duplex Mode:

```
0 = Full duplex
1 = Half duplex
```

Bits 5-7

These bits specify the data format, clock and flow control parameters

Bit 8

Indicates there are no further octets, always set to 0.

clc

clc `(MAXCLC = 10) ca`n contain additional information about the calling line category for DPNSS. For example:

```
struct out_xparms outxp;
strcpy(outxp.unique_xparms.sig_dpnss.clc,"*2");
```

If you do not wish to use clc then set

```
clc[0] = 0.
```

In this case the driver will assume the default calling line category of `'*1'`.

NOTE

To allow for future upgrades and enhancements none of the elements of the unique_xparms will be value checked by the driver and will be passed directly to the signaling system.

The following examples apply:

1 Ordinary
2 Decadic
3 ISDN
4 PSTN
5 SSMF5
6 Operator
7 Network
8 Conference (no longer a valid option).

8.4 unique_xparms for CAS

```
typedef struct uniquex_cas
{
  ACU_UCHAR      category;
} UNIQUEX_CAS;
```

category (R2T1 only)

Used for the presentation & restriction of CLI

NOTE

The "-s47,n" R2T1 firmware configuration switch must be implemented for this option.

When making an outgoing going call:

If *category* is set to 0, CLI is indicated as 'Not Restricted'.

If *category* is set to 1, CLI is indicated as 'Restricted'.

When receiving an incoming call:

If *category* is set to 0 after calling call_details, CLI is 'Not Restricted'.

If *category* is set to 1 after calling call_details, CLI is 'Restricted'.

If *category* is set to 2 after calling call_details, CLI is 'Restricted' and no CLI was received.

Contact Aculab for details details of the CAS protocols currently supported.

8.5 unique_xparms for ISUP/SS7

Not all calls that use unique_xparms use all parameters listed, in these cases the parameters used are indicated in the section for the parameter.

```
typedef struct uniquex_isup
{
  ACU_UCHAR      service_octet;
  ACU_UCHAR      add_info_octet;
  ACU_UCHAR      dest_natureof_addr;
  ACU_UCHAR      dest_numbering_plan;

  struct
  {
    ACU_UCHAR    ie[MAXBEARER];
    ACU_UCHAR    last_msg;
  } bearer;

  ACU_UCHAR      orig_natureof_addr;
  ACU_UCHAR      orig_numbering_plan;
  ACU_UCHAR      orig_numbering_presentation;
  ACU_UCHAR      orig_numbering_screening;
  ACU_UCHAR      conn_natureof_addr;
  ACU_UCHAR      conn_numbering_plan;
  ACU_UCHAR      conn_numbering_presentation;
  ACU_UCHAR      conn_numbering_screening;
  ACU_UCHAR      conn_number_req;        /* only used in call_openout */
  ACU_UCHAR      orig_category;
  ACU_UCHAR      orig_number_incomplete;
  ACU_UCHAR      dest_subaddr[MAXNUM];
  ACU_UCHAR      orig_subaddr[MAXNUM];

  struct
  {
    ACU_UCHAR    ie[MAXHILAYER];
    ACU_UCHAR    last_msg;
  } hilayer;

  struct
  {
    ACU_UCHAR    ie[MAXLOLAYER];
    ACU_UCHAR    last_msg;
  } lolayer;

  struct
  {
    ACU_UCHAR    ie[MAXPROGRESS];
    ACU_UCHAR    last_msg;
  } progress_indicator;

  ACU_UCHAR      in_band;
  ACU_UCHAR      nat_inter_call_ind;     /* not used in call_openout */
  ACU_UCHAR      interworking_ind;
  ACU_UCHAR      isdn_userpart_ind;
  ACU_UCHAR      isdn_userpart_pref_ind;
  ACU_UCHAR      isdn_access_ind;
  ACU_UCHAR      dest_int_nw_ind;
  ACU_UCHAR      continuity_check_ind;
  ACU_UCHAR      satellite_ind;
  ACU_UCHAR      charge_ind;
  ACU_UCHAR      dest_category;     /* not used in call_openout */
  ACU_UCHAR      add_calling_num_qualifier_ind;
  ACU_UCHAR      add_calling_num_natureof_addr;
  ACU_UCHAR      add_calling_num_plan;
  ACU_UCHAR      add_calling_num_presentation;
  ACU_UCHAR      add_calling_num_screening;
  ACU_UCHAR      add_calling_num_incomplete;
  ACU_UCHAR      add_calling_num[MAXNUM];
```

```
  ACU_UCHAR      exchange_type;
  ACU_UCHAR      collect_call_request_ind;
  ACU_UCHAR      raw_fci[2]
  ACU_UCHAR      raw_nci
  ACU_UCHAR      raw_bci[2]
} UNIQUEX_ISUP;
```

As for Q931 Specific Information but with the following additional parameters.

dest_natureof_addr and conn_natureof_addr

Must contain a valid nature of address for the call and be appropriate for the signaling system. Allowable values include:

```
NOA_SUBSCRIBER_NUMBER
NOA_NATIONAL_RESERVED
NOA_NATIONAL
NOA_INTERNATIONAL
```

These are the values defined by Q.767 (International ISUP), although the driver will pass other values transparently. The default value is `NOA_INTERNATIONAL`.

conn_number_req

When set to a non-zero value, indicates that the caller has requested that the connected number is supplied when available.

```
0  supply connected number not allowed
1  supply connected number allowed
```

orig_category

The calling party's category is used to specify a type of call, for example:

```
CPC_FRENCH_OPERATOR
CPC_ENGLISH_OPERATOR
CPC_GERMAN_OPERATOR
CPC_RUSSIAN_OPERATOR
CPC_SPANISH_OPERATOR
CPC_ORDINARY_SUBSCRIBER
CPC_PRIORITY_SUBSCRIBER
CPC_DATA
CPC_TEST
CPC_PAYPHONE
```

These are the some of the values defined by Q.763; the driver will pass other values transparently. The default value is `CPC_ORDINARY_SUBSCRIBER`

orig_number_incomplete

This is the calling party incomplete (NI) parameter value as defined in Q763. It can be in one of two states:

```
0 complete
1 incomplete
```

in_band

Used to indicate if optional backward call information is available on the call.

```
0  no indication - In band information is not available
1  In band information or an appropriate pattern is now available
```

nat_inter_call_ind

When set to a non-zero value, indicates that the call is to be treated as an international call.

NOTE

The fields - isdn_access_ind, isdn_userpart_ind and interworking_ind need to be set in a slightly unusual way. This is to provide source-level compatibility with applications that were written to earlier versions of the API that did not support these fields. If the 'valid' field is set to a non-zero value, then the parameter value will be taken from the 'value' field. If the 'valid' field is zero, isdn_access_in will default to 1, and isdn_userpart_ind and interworking_ind will echo the same status as the corresponding field sent by the originator of the call.

interworking_ind

The *interworking_ind* field, when set to a non-zero value, indicates that interwoking has occurred, i.e. that SS7 has not been used in all parts of the networking connection. See note below regarding usage.

isdn_userpart_ind

This field indicates whether ISUP signaling has been used by all preceeding parts of the network connection (0 = Not ISUP all the way, 1 = ISUP).

isdn_userpart_pref_ind

The ISUP preference indicator will be encoded in the forward call indicators from the originating switch to enable correct routing at intermediate switches by indicate whether ISUP is preferred/required/not required all the way. Allowable values include:

```
PI_ISUP_PREFERRED
PI_ISUP_NOT_REQUIRED
PI_ISUP_REQUIRED
```

isdn_access_ind

This field indicates whether the called party is ISDN (1 = true, 0 = false).

dest_int_nw_ind

The *dest_int_nw_ind* field, when set to a non-zero value, indicates that routing to an internal network number is not allowed.

continuity_check_ind

The *continuity_check_ind* field, when set to a non-zero value, indicates that a continuity check is being performed either on this or a previous circuit. The user must wait for this to complete before progressing with the call. When the continuity check completes the bit will clear. Values currently assigned to this parameter include:

CCI_NOT_REQUIRED

CCI_REQUIRED - a continuity check is being performed on the circuit.

CCI_PREVIOUS - a continuity check is being performed on a previous cct.

NOTE

When a value of CCI_REQUIRED is encountered, the application must loop back transmit/receive data paths for the appropriate timeslot and then wait for the continuity check to complete.

satellite_ind

One of the nature of connection indicators, this may be set to either 1 or 2 if it is known that the call already includes 1 or 2 satellite circuits respectively.

```
0  no satellite circuits in the connection
1  one satellite circuit in the connection
2  two satellite circuits in the connection
```

charge_ind

The *charge_ind* field is applicable to outgoing calls only, and indicates whether the call is chargeable. The value may change at any state transition or event during call setup. Values assigned to this field include:

```
CHARGE_IND_NO_INDICATION
CHARGE_IND_NO_CHARGE
CHARGE_IND_CHARGE
```

dest_category

This field provides further information about the called party. Values assigned by ITU are as follows:

```
0 = No indication
1 = Ordinary subscriber
2 = Payphone
3 = spare
```

exchange_type

The exchange_type field specifies the ISUP Exchange Type for the call, as described by Q.764 paragraph 2.9.5.2. Values assigned to this field include:

```
EXCHANGE_TYPE_A
EXCHANGE_TYPE_B
```

collect_call_request_ind

The collect_call_request_ind field allows an application to query or specify the value of a Collect Call Request parameter in an IAM message. The format of the Collect Call Request parameter is described by Q.763 section 3.81.

Add calling number

These fields allow an additional calling party number to be sent during `call_openout()`. The additional calling party number will appear in an oubound IAM as a Generic Number parameter.

add_calling_num_qualifier_ind

The value for the number qualifier field as specified in Q.763 section 3.26

add_calling_num_natureof_addr

Must contain a valid nature of address. Allowable values include:

```
NOA_SUBSCRIBER_NUMBER
NOA_NATIONAL
NOA_INTERNATIONAL
```

The default value is `NOA_INTERNATIONAL`

add_calling_num_plan

```
Can contain a valid numbering plan for the call and must be appropriate for
the signalling system. Allowable values are as specified in Q.763 section
3.26. Some signalling systems may not support all of these values.
```

```
NP_ISDN
NP_DATA
NP_TELE
```

add_calling_num_presentation

This field indicates the intention of the calling party for the presentation of the calling number (*originating_address*) to the called party.

The possible values are:

```
PR_ALLOWED (Presentation Allowed)
PR_RESTRICTED (Presentation Restricted)
PR_NOTAVAILABLE (Number not available due to interworking)
```

add_calling_num_screening

The possible values for this field are:

`SC_NOTSCREENED` (User provided, not screened)

SC_VERIFYPASS (User provided, verified and passed)
SC_VERIFYFAIL (User provided, verified and failed)
SC_NETWORKPROVIDED (network provided)

add_calling_num_incomplete
This is the number incomplete indicator as defined in Q763 section 3.26 d). It can be in one of two states:

```
0 = complete
1 = incomplete
```

add_calling_num
The digits to be sent should be placed in this field.

8.6 unique_xparms for IP telephony (iptel)

The parameters that are common for all IP Telephony protocols will form the basis of a structure called, *uniquex_iptel*. This structure will also contain a union of structures called *protocol_specific*. Parameters specific to a particular IP Telephony protocol will be defined here, one structure per protocol.

To use *uniquex_iptel*, you must include the `iptel_lib.lib` library and `#include iptel_lib.h`

NOTE

The IP telephony unique structure parameters used by xcall_getcause and xcall_disconnect vary slightly from the uniquex_iptel structure parameters used by all other generic API calls.

```
typedef struct uniquex_iptel
{
  ACU_CHAR                destination_display_name[MAXDISPLAY];
  ACU_CHAR                originating_display_name[MAXDISPLAY];
  ACU_CODEC               codecs[MAXCODECS];
  MEDIA_SETTINGS          media_settings;
  ACU_POINTER             vmprxid;
  ACU_POINTER             vmptxid;
  ACU_CHAR                media_call_type[MAXMEDIACALLTYPE];
  union protocol_union    protocol_specific;
  ACU_UINT                ipv6_media;
} UNIQUEX_IPTEL;

typedef struct acu_codec
{
  ACU_INT                 codec_type;
  ACU_INT                 vad;
  ACU_INT                 fpp;
  ACU_ULONG               options;
} ACU_CODEC;

typedef struct media_settings
{
  ACU_INT                 tdm_encoding;
  ACU_INT                 encode_gain;
  ACU_INT                 decode_gain;
  ACU_INT                 echo_cancellation;
  ACU_INT                 echo_suppression;
  ACU_INT                 echo_span;
  ACU_UINT                rtp_tos;
  ACU_UINT                rtcp_tos;
  ACU_UINT                dtmf_detector;
} MEDIA_SETTINGS;

union protocol_union
{
  struct
  {
    ACU_CHAR              destination_alias[MAXADDR];
    ACU_CHAR              originating_alias[MAXADDR];
    ACU_INT               h245_tunneling;
    ACU_INT               faststart;
    ACU_INT               early_h245;
    ACU_CHAR              dtmf[MAXNUM];
    ACU_INT               progress_location;
    ACU_INT               progress_description;
  } sig_h323;

  struct
```

```
  {
    ACU_CHAR              contact_address[MAXADDR];
    ACU_INT               zero_connection_address_hold;
    ACU_INT               disable_reliable_provisional_response;
    ACU_INT               disable_early_media;
  } sig_sip;
};
```

destination_display_name
Used to transmit destination display information.

originating_display_name
Used to transmit originating display information.

codecs
Codec assignment array that encapsulates the settings available on a per codec basis. It contains a list of the permitted codecs that may be negotiated with the remote endpoint during call establishment. The first entry is treated as highest priority and the last entry is treated as the lowest.

codec_type
Permitted codec values are:

```
G711_ALAW
G711_ULAW
G723
G729
G729A
```

Valid permutations for the codec structure are as follows :

```
G711_ALAW             VAD on
G711_ALAW             VAD off
G711_ULAW             VAD on
G711_ULAW             VAD off
G723                  VAD on
G723                  VAD off
G729                  VAD off
G729                  VAD on
G729A                 VAD off
G729A                 VAD on
```

With `fpp` set in the range 1-3.

If a codec is specified that is not supported by the board then the API will ignore it. This allows the same codec list to be used over boards with different capabilities.

If no valid codecs are specified then the system codec list will be used. If this also contains no valid codecs then calls will be failed with a parameter error.

vad
Allows the **voice activity detector** to be turned on for this call. Turning on the voice activity detector allows the IP Telephony card to perform silence suppression for that call. Permitted values are :

```
VAD_ON      1
VAD_OFF     0
```

fpp
Can be used in API calls, prior to call connection, to specify the actual number of **frames per packet**. The minimum value is 1 and the maximum value is 3, with the default being 2 frames per packet.

options
Required for future development.

media_settings
Parameter information that is required by the Media Gateway API.

tdm_encoding

The *tdm_encoding* parameter allows μ-law or a-law encoding to be selected for the telephony interface on a per call basis and has no effect on the selected IP Telephony codec. If no value is specified for the call, the system will default to the encoding configured for the firmware, this is currently set to μ-law. The permitted values are:

```
TDM_ULAW  1
TDM_ALAW  2
```

encode_gain/decode_gain

The *encode_gain* parameter allows adjustment of the input signal from the telephony interface to the IP Telephony encoder, while the decode_gain parameter allows adjustment of the output signal from the IP Telephony decoder to the telephony interface. Permitted values for these two parameters are:

Allow the IP Telephony card to choose a sensible default gain level. This is currently equivalent to setting a gain of 0x2000

0x0001 – 0xFFFF specify a gain level manually

NOTE

encode_gain and decode_gain are not supported on Prosody X cards, and will be ignored.

echo_cancellation

The possible values are :

EC_OFF – disables echo canceller (invalidates *echo_span* option)
EC_ON – enables G.168 echo canceller for Prosody X cards.
EC_ON_NLP – enablesG.168 echo canceller with non-linear processing for Prosody X cards.

echo_suppression

The possible values are :

ES_OFF - echo suppression option is disabled
ES_12DB - echo suppression option is enabled

NOTE

The echo canceler and suppressor are independent subsytems of the echo software and as such can be controlled independently.

echo_span

This is the length, in milliseconds, of the echo canceller tail. It may be 4, 6, 8, 10, 12, 14, 16 or 32ms tail length.

NOTE

A 32ms tail length cannot be used with G.723.1

The defaults for *echo_cancellation*, *echo_suppression* and *echo_span* are : EC_G165, ES_OFF, 16.

NOTE

echo_suppression and echo_span are not supported on Prosody X cards, and will be ignored.

rtp_tos
The byte field *rtp_tos* specifies the type of service field that will be used in the IP headers of RTP packets sent by the board on a per call basis for `call_openout()` and `xcall_accept()` functions. To set the ToS to zero a value of 0x100 should be used.

rtcp_tos
The byte field rtcp_tos specifies the type of service field that will be used in the IP headers of RTCP packets sent by the board on a per call basis for `call_openout()` and `xcall_accept`() functions. To set the ToS to zero a value of 0x100 should be used.

dtmf_detector
The IP Telephony card can detect DTMF in the audio stream switched to it and treat it differently to normal audio by blocking DTMF in the outgoing audio stream and sending RFC 2833 frames instead.

If *dtmf_detector* is set to `IPT_ENABLED` then this processing will be performed.
if *dtmf_detector* is set to `IPT_DISABLED` then DTMF will not be detected, and will be treated as normal audio.

vmprxid
Used for TiNG media configuration, see Appendix K for details.

vmptxid
Used for TiNG media configuration, see Appendix K for details.

media_call_type
This is a call type defined via the TiNG Resource Manager. If specified for a call on a card, that uses the TiNG Resource Manager, then resources of that call type will be reserved for this call. It must be specified in `call_openout()` or `call_opening()`, otherwise a default call type will be used. Supported call types for IP Telephony are:

```
acu+g711+td+echo
acu+g711+td
acu+g711+echo
acu+g711.
```

If you have G.729 support on your card, the following call types may also be supported:

```
acu+g711+g729+td+echo
acu+g711+g729+td
acu+g711+g729+echo
acu+g711+g729.
```

See the TiNG Resource manager API for more information on call types. Any `media_call_type` setting will only be used for cards which support the TiNG Resource Manager, such as Prosody X.

protocol_union (protocol_specific)
A union of structures representing various IP Telephony signalling protocols.

ipv6_media
Present in version 6.6.0 and later. When set to a non-zero value, the connection address for RTP will use IPv6. Otherwise it will default to IPv4.

Unique parameters for H.323

destination_alias
Only applies when using call_openout() or call_details(). For H.323 this will take the form of an additional alias for the destination target. It will be in URI address format.

originating_alias
Only applies when using call_openout() or call_details(). For H.323 this will take the form of an additional alias for the originating source. It will be in URI address format.

h245_tunneling
Allows H.245 Tunneling to be enabled for this call. This is the process of sending H.245 PDUs through the Q.931 channel (encapsulating the H.245 messages within H.225/Q.931 messages). The same TCP/IP socket that is already in use for the Call Signalling Channel, is also used by the H.245 Control Channel.

If set to IPT_ENABLED, tunnelling is enabled.
If set to IPT_DISABLED, tunnelling is disabled.

NOTE

H.245 Tunneling is enabled by default. This default can be overridden using ipt_set_protocol_defaults. Refer to IP Telephony card V6 API Guide for further information.

faststart
Allows Fast Start, also known as Fast Connect, to be enabled for this call. This procedure reduces the time required to set up a call to one round-trip delay following the H.225 TCP connection. If set to IPT_ENABLED, Fast Start is enabled. If set to IPT_DISABLED, Fast Start is disabled.

NOTE

Fast Start is enabled by default. This default can be overridden using ipt_set_protocol_defaults. Refer to IP Telephony card V6 API Guide for further information.

early_h245
Allows early H.245 to be enabled by default for this call. This involves opening the H.245 channel before H.225 has completed. By starting H.245 early, two endpoints can establish media quicker. If set to IPT_ENABLED, early H.245 is enabled. If set to IPT_DISABLED, early H.245 is disabled.

NOTE

early H.245 is enabled by default. This default can be overridden using ipt_set_protocol_defaults. Refer to IP Telephony card V6 API Guide for further information.

dtmf
Only applies when using call_details() as a result of EV_DETAILS to retrieve details of the current call, either incoming or outgoing. It holds the User Input Indication information that has just been received on the network for the call.

progress_location
Only applies when using call_details() as a result of EV_PROGRESS to retrieve details of the current call. It holds the progress location information that has just been

received on the network for the call.

progress_description
Only applies when using `call_details()` as a result of `EV_PROGRESS` to retrieve details of the current call. It holds the progress description information that has just been received on the network for the call.

Unique parameters for SIP

contact_address
Used to build a non-default contact header. For chassis containing only one NIC card this field should be left blank. It will be in URI address format.

URIs are a well defined international standard for supporting multiple addressing types. They take the form:

```
<scheme>:<scheme-specific-part>
```

Examples of URIs could be:

```
sip:joe.bloggs@aculab.com
h323:joe.bloggs@aculab.com
mailto:joe.bloggs@aculab.com
http://www.aculab.com
tel:+441315610104
```

NOTE

The SIP service currently only supports the sip: uri.

IANA's (Internet Assigned Numbers Authority) current register of registered schemes can be found at http://www.iana.org/assignments/uri-schemes.

If the address supplied does not conform exactly to the URI format, for example, `<scheme>: section missing`, the IP protocol will try to determine what has been entered.

zero_connection_address_hold
When set the service assumes that remote party implements call hold to the earlier Internet specification, that is c=0.0.0.0 in the SDP body. By default the service assumes that the latest specification (`a=sendonly/recvonly`). It may be set in `call_openout` or `xcall_accept`.

disable_reliable_provisional_response
For an incoming call, if the `INVITE` message from the caller suggests that the caller does not support reliable provisional response (absence of "Supported: 100rel"), this flag will be set in `call_details`. It is not used in order that parties may choose to switch off reliable provisional responses.

disable_early_media
For an outgoing call, when this is set the calling party refuses to participate in an early media session, even if one is offered by the called party. By default the calling party will participate in such sessions, if offered by called party.

Example usage:

To set the TDM encoding, say for incomming ringing, you can use the following:

```
struct incoming_ringing_xparms ringing_xparms;
ringing_parms.unique_xparms.sig_iptel.media_settings.tdm_encoding=TDM_ALAW;
```

9 disconnect_xparms

Used by:

```
ACC_ERR xcall_getcause(DISCONNECT_XPARMS *causep);
ACC_ERR xcall_disconnect(DISCONNECT_XPARMS *causep);
ACC_ERR xcall_release(DISCONNECT_XPARMS *causep);
```

and takes the form:

```
typedef struct disconnect_xparms
{
  ACU_ULONG           size;
  ACU_CALL_HANDLE     handle;
  ACU_INT             cause;
  union uniqueu       unique_xparms;
} DISCONNECT_XPARMS;

union uniqueu
{
  /* see the following protocol specific structures */
} unique_xparms;
```

9.1 Q931

```
    struct
    {
      ACU_INT         raw;
      struct
      {
        ACU_UCHAR          ie[MAXBEARER];
        ACU_UCHAR          last_msg;
      } progress_indicator;
      struct
      {
        ACU_UCHAR          ie[MAXDISPLAY];
        ACU_UCHAR          last_msg;
      } display;
      struct
      {
        ACU_UCHAR          ie[MAXBEARER];
        ACU_UCHAR          last_msg;
      } notify_indicator;
      ACU_INT         location;
      ACU_INT          coding_standard;
    } sig_q931;
```

9.2 ISUP/SS7

```
    struct
    {
      ACU_INT         raw;
      struct
      {
        ACU_UCHAR          ie[MAXPROGRESS];
        ACU_UCHAR          last_msg;
      } progress_indicator;
        ACU_INT        location;
        ACU_INT        reattempt;
    } sig_isup;
```

location

The user can specify the location value to be used, or use a value of -1 to default to the location value specified in the ss7 configuration file.

9.3 DPNSS

```
struct
{
  ACU_INT             raw;
} sig_dpnss;
```

9.4 DASS

```
struct
{
  ACU_INT             raw;
} sig_dass;
```

9.5 CAS

```
struct
{
  ACU_INT             raw;
} sig_cas;
```

9.6 IP telephony (iptel)

```
Struct
{
  union
  {
    struct
    {
      ACU_INT          raw;
      ACU_INT          raw_type;
    } sig_h323;
    struct
    {
      ACU_INT          raw;
      ACU_CHAR         warning[MAXSIPWARNING];
      ACU_CHAR         response_explanation[MAXSIPRESPONSEEXPLANATION];
    } sig_sip;
  } protocol_specfic;
} sig_iptel;
```

The *disconnect_xparms* structure can be used when more protocol specific information is made available relating to the clearing of the incoming or outgoing call.

handle
Is used to identify the call.

cause
May be used to provide the device driver with the generic clearing cause for the call. The cause must one from the standard set of generic LC_xxxx clearing causes as detailed in Appendix E:.

unique_xparms

raw

Protocol specific clearing cause. Must contain a value that is appropriate for the protocol in use.

raw_type (H.323)

Used to identify the H.323 specific clearing cause type. Valid values are :

H225_RCR	- H.225 release complete reason
Q931_CAUSE	- Q.931 clearing cause

progress_indicator (Q931)
May be used to indicate events pertaining to the call regarding in-band information. The ie field of the *progress_indicator* structure should contain the progress

information which is sent transparently to the other end. The `last_msg` field should be left blank. This information is dependent on the signalling system and reference should be made to the appropriate specification for the protocol.

`progress_indicator (ISUP)`
Is currently unused for ISUP and is ignored by the driver.

`notify_indicator (Q931)`
Can be used to indicate the information detailing the call related event pertaining to the call. The `ie` field of the *`notify_indicator`* structure should contain the notify information that is sent transparently to the other end. The *`last_msg`* field should be set to zero. This information is dependent on the signalling system and reference should be made to the appropriate specification for the protocol.

`display (Q931)`
Can be used to transmit information that may be displayed by the user.

`location (ISUP)`
Provides the device driver with the value that will be used in the cause indicators parameter if an ISUP `RELEASE` message is sent.

`reattempt (ISUP)`
If set to a non-zero value, this indicates that an outgoing call has failed owing to a transient condition (such as call collision) and is reattempt able.

`Warning (SIP)`
This field contains the contents of the SIP Warning header if one is present in the terminating SIP response.

`response_explanation(SIP)`
This field may contain extra information suggesting what action may be taken in order that future call attempt should succeed. The nature of this information is dependent on the raw cause (ie. the SIP response code). If raw cause is 420 (Bad Extension) and the response has an unsupported header then this string contains the options not supported by the remote end.

10 Feature xparms

There are a number of data structures used in the extended feature section of the call API. These are:

```
uui_xparms
facility_xparms
diversion_xparms
feature_hold_xparms
feature_transfer_xparms
raw_data_struct
mlpp_xparms
non_standard_data_xparms
raw_msg_xparms
call_waiting_xparms
restart_channel_xparms
addressed_non_standard_data_xparms
feature_activation_xparms
information_request_xparms
name_presentation_xparms
```

10.1 uui_xparms- user to user information

User-to-user (UU) signalling is passed transparently across a network to the receiving user. UU services are currently supported on Aculab EuroISDN (primary rate), SS7, NI2 and QSIG signalling systems.

Types of service include:

`Service 1` – used to send data during the call setup and clearing phases

`Service 2` – used to send data during the call alerting stage.

`Service 3` – used to send data in the call-connected state.

Normally it is necessary to request a service and receive an acceptance before a service can be used to send data. Calling the user makes the service requests, but as a network option it may be possible for service 3 to be requested by the called user when in the connected state. Not all networks will support all three types of service.

An exception to the above applies to service 1 where data can be sent in a call setup message without sending a request (implicit preferred request).

If a service is requested as required but the called party or the network rejects the service, then the call is normally cleared by either the called party or the network.

Data can be sent at the same time as a request for service.

The UU services on Aculab EuroISDN and QSIG are based on the services described in ETSI EN 286-1. The SS7 implementation is based on Q737.1

To `include` UU information the `feature_information` field should be supplied with the value `FEATURE_USER_USER`.

```
typedef struct uui_xparms
{
  ACU_INT         command;
  ACU_UINT        request;
  ACU_INT         tx_response;
  ACU_INT         rx_response;
  char            control;
  char            flow_control;
  ACU_UCHAR       protocol;
  char            more;
  ACU_UCHAR       length;
  ACU_UCHAR       data[MAXUUI_INFO];
} UUI_XPARMS;
```

The `uui_xparms` structure must be used in the following way to transmit information in `call_feature_openout()`, `call_feature_send()` or `call_feature_details()`.

`command`

For *`uui_xparms`*, the command field may be used to define what is transmitted. The possible values are:

`UU_SERVICE_CMD` – When sending a UU service request or response.

`UU_DATA_CMD` – When sending UU data

`UU_GET_PENDING_DATA_CMD` – Only applicable to `call_feature_details()`.

NOTE

For ETS300, the `UU_DATA_CMD` command may be used to send UU data and UU service requests/responses in the same protocol message.

request

If a UU service has been requested of the application then following a call to `call_feature_details()` the request field will contain the value of the requested service. If no service has been requested then the value will be zero.

If a service request needs to be made using `call_feature_send()` or `call_feature_openout()` then the request field should be set to the requested service. `call_feature_openout()` uses UU service 1-3 but call_feature_send() only uses UU service 3. Requests for service 1 and/or 2 should only be made using `call_feature_openout()`.

Valid values for the request field are:

```
UUS_1_IMPLICITLY_PREFERRED
UUS_1_REQUIRED
UUS_1_PREFERRED
UUS_2_REQUIRED
UUS_2_PREFERRED
UUS_3_REQUIRED
UUS_3_PREFERRED
```

tx_response

`tx_response` is used when sending a response to a service request using `call_feature_send()`. It indicates whether or not the request had been accepted. Valid values are:

```
UUS_1_ACCEPTED
UUS_1_REFUSED_BY_PEER
UUS_1_REFUSED_BY_NET
UUS_2_ACCEPTED
UUS_2_REFUSED
UUS_3_ACCEPTED
UUS_3_REFUSED
```

NOTE

For ETS300, use of the `UUS_1_REFUSED_BY_PEER` and `UUS_1_REFUSED_BY_NET` values when setting `uui_xparms.tx_response` requires v6.5.46 call drivers or newer.

NOTE

For ETS300, when setting `uui_xparms.tx_response` to either `UUS_2_REFUSED` or `UUS_3_REFUSED`, the call driver will send an appropriate rejection end type (by peer or by network) in the protocol message based on the call firmware running.

NOTE

For ETS300, the call driver may automatically respond to UU service requests by either accepting them (if the `-cFU` switch is set) or refusing them (if the `-cFU` switch is omitted). The automatic UU service response occurs when the application fails to supply the call driver with a UU service response (using `call_feature_send()`) before calling `call_incoming_ringing()` or `call_accept()`. The automatic UU responses maybe disabled using the `-cDAUUS` configuration switch. Requires version v6.5.46 call drivers or newer.

rx_response

If the application has requested a User to User service then following a call to

call_feature_details() the rx_response field may contain the response of the other party to this request. A response can take one of the following values:

```
UUS_1_ACCEPTED
UUS_1_REFUSED_BY_PEER
UUS_1_REFUSED_BY_NET
UUS_2_ACCEPTED
UUS_2_REFUSED_BY_PEER
UUS_2_REFUSED_BY_NET
UUS_3_ACCEPTED
UUS_3_REFUSED_BY_PEER
UUS_3_REFUSED_BY_NET
```

Control

The control field can take one of two values that will affect the way the facility information is transmitted. These values are

```
CONTROL_DEFAULT       - The information will be transmitted immediately in a
Q.931 FACILITY message by the driver

CONTROL_NEXT_CC_MESSAGE    - The information will be stored and send in the
next appropriate call control message.
```

flow_control

If an application sends UU service 3 data and the driver wishes to indicate that this will be the last transmission until some future time, then the UUI_FC_STOP_SENDING bit in the flow_control field will be set. If capacity remains to send additional messages, this bit is cleared.

If an application continues to send UU service 3 data before sending is once more possible, then the UUI_FC_DATA_DISCARDED bit in the *flow_control* field will be set and the data will not be queued for transmission.

NOTE

Flow control is not applicable to ISUP/SS7 and should be set to zero.

protocol

The protocol field should be supplied with one of the following values

```
UUI_PROTOCOL_USER_SPECIFIC
UUI_PROTOCOL_OSI_HIGHER_LAYER
UUI_PROTOCOL_CCITT_X244
UUI_PROTOCOL_SYSMNG_CONV
UUI_PROTOCOL_IA5
UUI_PROTOCOL_CCITT_V120
UUI_PROTOCOL_CCITT_Q931
```

more

If the *more* field contains a non-zero value then this is an indication to the receiver that more data will follow belonging to the same block.

length

This is the *length* in bytes of the data being supplied in the data field. The max value is subject to the current data function, for example, for uui_xparms the value would be as currently defined in the header file for MAXUUI_INFO.

In most instances the current maximum is 128 bytes.

data

The data field should be supplied with or contain information structured according to the protocol parameter described previously. The length of this information should be supplied in the length field.

NOTE

To send a protocol discriminator octet under ISUP/SS7 when length is zero, set the first byte of data to the special value `UUI_NO_DATA_IN_MESSAGE`.

Using User to User Information with NI2

It is possible to send User-to-User Information (UUI) with the following protocol messages in NI2:

SETUP

Sent using `call_feature_openout`, with the `uui_xparms` request field controlling call behaviour if the far end does not support UUI, as below

```
uui_xparms.command      = UU_DATA_CMD;
uui_xparms.request      = UUS_1_PREFERRED; //OR UUS_1_REQUIRED
uui_xparms.control      = CONTROL_DEFAULT;
uui_xparms.protocol     = UUI_PROTOCOL_CCITT_Q931;
```

With `UUS_1_REQUIRED` if the far end responds with a STATUS message showing it does not support UUI the driver will release the call.

With `UUS_1_PREFERRED` if the far end responds with a STATUS message showing it does not support UUI the driver will not drop the call, but will not convey further UUI messages which are attempted to be sent to the far end

CONNECT, DISCONNECT, RELEASE and PROGRESS

Sent using `call_feature_send`, populated as below, prior to the call control API call used to send the next call control message

```
feature_detail_xparms.feature_type         = FEATURE_USER_USER;
feature_detail_xparms.message_control      = CONTROL_NEXT_CC_MESSAGE;
feature_detail_xparms.feature.uui.request  = UUS_1_IMPLICITLY_PREFERRED;
```

USER INFORMATION

Sent using `call_feature_send`, populated as below,

```
feature_detail_xparms.feature_type         = FEATURE_USER_USER;
feature_detail_xparms.message_control      = CONTROL_DEFAULT;
feature_detail_xparms.feature.uui.request  = UUS_1_IMPLICITLY_PREFERRED;
```

NOTE

UUI in NI2 uses UUI in a SETUP message to request UUI for the rest of the call. If UUI is not sent/received in the SETUP message but attempts are made to send UUI in subsequent call control messages an error will be returned to the application.

NOTE

To enable UUI with NI2 the `-cFUN` switch must be added to the firmware download parameters.

NOTE

To enable USER INFORMATION messages with NI2, the `-s80,n` must be added to the download parameters. Please see the firmware release notes for more details. Requires v6.5.26 call drivers (or newer) and v1.6.23 NI2 firmware (or newer).

Using User to User Information service with AT&T

Firmware must be downloaded with -cFU set. Unlike with the Q.931 signalling types, where the services are split up into services 1, 2 and 3, with AT&T they are split into MA-UUI, CA-TSC and NCA-TSC.

Message Associated UUI (MA-UUI)

User to user data is transferred in Q.931 call control messages either at call setup or call clearing phases of a call. This can be in SETUP, ALERT, CONNECT or DISCONNECT messages.

To send data in a SETUP message...

Use `call_feature_openout()` with *command* in the uui_xparms structure set to `UUS_ATT_MA_UUI` and *data* populated with the data to send and *length* field set.

To receive data from a SETUP message.

After `EV_INCOMING_CALL_DET`, call details will show user to user feature available. Then call `call_feature_details()` with *feature_type* set to `FEATURE_USER_USER`. The *data* field in the uui_xparms structure would be populated with the data received, and *length* set to the length of the data.

To send data in other call control messages

After `EV_INCOMING_CALL_DET`, send `call_feature_send()` with *feature_type* set to `FEATURE_USER_USER`, and *command* in the uui_xparms structure set to `UUS_ATT_MA_UUI`. *data* would be populated with the data to send and *length* field set. For example do this before `call_incoming_ringing()` to send in ALERT message.

To receive data from other call control messages (ALERT/CONNECT/DISCONNECT)...

For other call control messages a call to `call_feature_details()` would retrieve the data received in the same way with *feature_type* set to `FEATURE_USER_USER`.

Call Associated Temporary Signalling Connections (CA-TSC)

CA-TSC is like a combination of services 2 and 3 on ETSI ETS300/QSIG. However one difference is that CA-TSC allows user to user data to be exchanged before the alerting message is received by the calling user. This can be done if the called user responds to accept the service before the alerting message is sent.

Requesting CA-TSC at call setup

Request using `call_feature_openout()` and feature_information set to `FEATURE_USER_USER`. *request* in the uui_xparms structure is set to `UUS_ATT_CA_TSC`.

Requesting CA-TSC after call setup

Use `call_feature_send()` with feature_type set to `FEATURE_USER_USER`. In the uui_xparms structure *request* should be set to `UUS_ATT_CA_TSC`.

In either case a response would be received in an extended event `EV_EXTENDED`, `EV_EXT_UUS_SERVICE_REQUEST`

If accepted then *rx_response* would be set to `ATNT_UUS_CA_TSC_ACCEPTED`

If rejected then *rx_response* would be set to ATNT_UUS_CA_TSC_REJECTED

CA-TSC Data Transfer

After CA-TSC has been accepted, User to User data can be transferred.

The maximum length of data than can be transmitted or received with AT&T is MAXUUI_INFO_ATT, which is different from the length allowed with ETS300 or QSIG (MAXUUI_INFO).

To send CA-TSC data for either an incoming or outgoing call after the service has been set up, call call_feature_send() with *feature_type* set to FEATURE_USER_USER and *command* in the uui_xparms structure set to UUS_ATT_CA_TSC_DATA.

To receive CA-TSC data for either an incoming or outgoing call after the service has been set up, an event will be received EV_EXTENDED, EV_EXT_UUI_PENDING. A call to call_feature_details() with *feature_type* set to FEATURE_USER_USER will result in the *data* field of the uui_xparms structure populated with the received data and *length* to the length of the data.

CA-TSC Congestion Messages (for flow control)

When using the CA-TSC service it is possible for either end to send a 'Receive not ready' indication. The end that sends the 'receive not ready' is the one that can clear it with a 'receive ready'. When a receive not ready is received, no more data should be sent until the condition is cleared. The network can discard any messages received after a 'receive not ready' has been sent.

If a congestion message with receive not ready is received, a EV_EXT_UUI_CONGESTED extended event will be received.

When the condition is cleared a EV_EXT_UUI_UNCONGESTED extended event will be received.

To send a receive not ready message use call_feature_send() with *feature_type* set to FEATURE_USER_USER and *command* in the uui_xparms structure set to UUS_ATT_CONGESTION_RNR.

To send a receive ready message use call_feature_send() with *feature_type* set to FEATURE_USER_USER and *command* in the uui_xparms structure set to UUS_ATT_CONGESTION_RR.

Non-Call Associated Temporary Signalling Connections (NCA-TSC)

This allows a call to be set up with no bearer. The call is set up and cleared in a similar way to a normal call. User to user data can be exchanged after the call is connected in USER INFORMATION messages.

To set up a call with NCA-TSC

To set up the NCA-TSC call, use call_feature_openout() with *feature_information* set to FEATURE_USER_USER and in the uui_xparms structure have *request* set to UUS_ATT_NCA_TSC.

If it is necessary to select between SDN and ACCUNET Switched Digital then these values can be used - UUS_ATT_NCA_TSC_SDN and UUS_ATT_NCA_TSC_ACCUNET. The default with UUS_ATT_NCA_TSC is to have the request set to SDN.

To accept an incoming call with NCA-TSC

An incoming call will be detected with EV_INCOMING_CALL_DET. The result from call_details() will include f*eature_information* set to FEATURE_USER_USER. The result from call_feature_details() with *feature_type* set to FEATURE_USER_USER will have *request* set to UUS_ATT_NCA_TSC.

To accept an incoming request for NCA-TSC use `call_feature_send()` with *feature_type* set to `FEATURE_USER_USER` and *tx_response* in the uui_xparms structure set to `ATNT_UUS_NCA_TSC_ACCEPTED`. The call will go to connected state.

To send NCA-TSC data for either an incoming or outgoing call after the service has been set up, `call call_feature_send()` with *feature_type* set to `FEATURE_USER_USER` and *command* in the uui_xparms structure set to `UUS_ATT_NCA_TSC_DATA`.

To receive NCA-TSC data for either an incoming or outgoing call after the service has been set up, an event will be received `EV_EXTENDED`, `EV_EXT_UUI_PENDING`. A call to `call_feature_details()` with *feature_type* set to `FEATURE_USER_USER` will result in the *data* field of the uui_xparms structure populated with the received data and *length* to the length of the data.

10.2 facility_xparms - facility information

To include facility information, the feature_information field should be supplied with the value FEATURE_FACILITY.

```
typedef struct facility_xparms
{
  ACU_INT             command;
  ACU_UCHAR           control;
  ACU_UCHAR           length;
  ACU_UCHAR           data[MAXFACILITY_INFO];
  char                destination_addr[MAXNUM];
  char                originating_addr[MAXNUM];
  ACU_UCHAR           dest_subaddr[MAXNUM];
  ACU_UCHAR           dest_numbering_type;
  ACU_UCHAR           dest_numbering_plan;
  ACU_UCHAR           orig_numbering_type;
  ACU_UCHAR           orig_numbering_plan;
  ACU_UCHAR           orig_numbering_presentation;
  ACU_UCHAR           orig_numbering_screening;
  ACU_UCHAR           more;
} FACILITY_XPARMS;
```

Example call_feature_openout()

The *facility*_xparms structure must be used in the following way to transmit information in call_feature_openout():

data

The *data* field should be supplied with protocol dependant information.

length

The length of this information should be supplied in the *length* field.

All other fields are not applicable for an outgoing call.

Example Call_feature_send()

control

The *control* field can take one of two values that will affect the way the facility information is transmitted. These values are

CONTROL_DEFAULTS /* Immediate */

If the immediate option is taken then the information will be transmitted immediately in a Q.931 FACILITY message by the driver.

CONTROL_NEXT_CC_MESSAGE /* Next call control message */

If the Next Call Control Message option is chosen then the information will be stored and sent in the next appropriate call control message.

Example:

If call_incoming_ringing() was subsequently called then the information would be included in a Q.931 ALERTING message).

The data field should be supplied with protocol dependant information.

The length of this information should be supplied in the length field.

All other fields are not applicable to call_feature_send()

Facility_xparm parameter definitions

Command

For facilities_xparms, EuroISDN allows the possibility of two types of connectionless transmission. The command field selects the mode for this call. If using the connectionless data facility, this field must contain one of the values:

FAC_CLESS_DL_DATA_CMD ETS300 196 Section 8.3.2.2

FAC_CLESS_DL_UNIT_DATA_CMD ETS300 196 Section 8.3.2.4

This field only needs to be filled when sending data using connectionless data. If the data is being sent as a part of a call then it should not be filled in.

This value is ignored when using QSIG.

Control
The control field can take one of two values that will affect the way the facility information is transmitted. These values are

CONTROL_DEFAULT

Immediate - the information will be transmitted immediately in a Q.931 FACILITY message by the driver.

CONTROL_NEXT_CC_MESSAGE

Next Call Control Message (the information will be stored and sent in the next appropriate call control message).

Length
This is the length in bytes of the data being supplied in the data field. The max value is subject to the current data function, for example, for uui_xparms the value would be as currently defined in the header file for MAXUUI_INFO.

In most instances the current maximum is 128 bytes.

Data
The data parameter is not to be confused with data types.

For data types the device driver makes the following assumptions about the data types char, int and long:

```
char:   8 bit, signed
int:    32 bit, signed
long:   32 bit, signed
```

destination_addr
The input character buffer contains a null terminated string of IA5 digits (0-9). This field can be:

- The whole of the number to be dialled (en bloc).
- Part of the number to be dialled (overlap sending).
- Empty, indicating no digits provided (overlap sending).

The *destination_addr* contains the IP address of the call destination. It must be an unsigned long in network byte order, as returned by the inet_addr socket function.

The digits supplied are copied, (***not*** concatenated) to destination_addr. If, when initiating the outgoing call, the sending_complete parameter was set to 1, then the destination_addr field may contain a '!' to indicate that the number is complete.

originating_addr
The input character buffer originating_addr can be supplied with a null terminated string of IA5 digits. This string represents the originating subscriber number. This string will be passed to the signalling system when the outgoing call is made. This provides for originating_addr to be specified on a per call basis.

NOTE

In the DASS signaling system, originating_addr may contain a null terminated ASCII string of extension number digits.

dest_subaddr
Maximum length MAXNUM using digits 0 to 9 – see Appendix H Using Subaddress

Information for further details.

dest_numbering_type

Can contain a valid numbering type for the call and must be appropriate for the signalling system. Allowable values are:

```
NT_UNKNOWN
NT_INTERNATIONAL
NT_NATIONAL
NT_NETWORK_SPECIFIC
NT_SUBSCRIBER_NUMBER
NT_ABBREVIATED_NUMBER
```

NOTE

Some signaling systems may not have support for all these values.

The default value is `NT_UNKNOWN`.

NOTE

Not applcable for an outgoing call

dest_numbering_Plan

Can contain a valid numbering plan for the call and must be appropriate for the signalling system. For example, allowable values may include:

```
NT_UNKNOWN
NT_ISDN
NT_DATA
NT_TELEX
NT_NATIONAL_STANDARD
NT_PRIVATE
```

NOTE

Some signaling systems may not have support for all these values.

The default value is `NT_UNKNOWN`.

orig_numbering_type

Can contain a valid numbering type for the call and must be appropriate for the signalling system. Allowable values are:

```
NT_UNKNOWN
NT_INTERNATIONAL
NT_NATIONAL
NT_NETWORK_SPECIFIC
NT_SUBSCRIBER_NUMBER
NT_ABBREVIATED_NUMBER
```

NOTE

Some signaling systems may not have support for all these values.

The default value is `NT_UNKNOWN`.

orig_numbering_plan

Can contain a valid numbering plan for the call and must be appropriate for the signalling system. For example, allowable values may include:

```
NT_UNKNOWN
```

```
NT_ISDN
NT_DATA
NT_TELEX
NT_NATIONAL_STANDARD
NT_PRIVATE
```

NOTE

Some signaling systems may not have support for all these values.

The default value is `NT_UNKNOWN`.

`orig_numbering_presentation`

The `orig_numbering_presentation` field indicates the intention of the calling party for the presentation of the calling number (`originating_address`) to the called party.

The four values are:

`PR_ALLOWED` (Presentation Allowed)

`PR_RESTRICTED` (Presentation Restricted)

`PR_NOTAVAILABLE` (Number not available due to interworking)

`PR_RESERVED`

`orig_numbering_screening`

The possible values for the *`orig_numbering_screening`* field are:

`SC_NOTSCREENED` (User provided, not screened)

`SC_VERIFYPASS`* (User provided, verified and passed)

`SC_VERIFYFAIL` (User provided, verified and failed)

`SC_NETWORKPROVIDED`* (network provided)

`more`

If the *`more`* field contains a non-zero value then this is an indication to the receiver that more facility data is available.

10.3 diversion_xparms - Diversion/redirect supplementary service

It is possible to include call diversion information to say that a new call has been diverted from another number. This makes it possible to include information on what number the call was diverted from and how many times diversion has occurred. This feature is supported on the EuroISDN, QSIG, NI-2, AT&T, ISUP and H.323 signaling systems.

To include *diversion* information, the *feature_information* field should be supplied with the value FEATURE_DIVERSION.

```
typedef struct diversion_xparms
{
  ACU_UCHAR      diverting_reason;
  ACU_UCHAR      diverting_counter;
  char           diverting_to_addr[MAXNUM];
  char           diverting_from_addr[MAXNUM];
  char           original_called_addr[MAXNUM];
  ACU_UCHAR      diverting_from_type;
  ACU_UCHAR      diverting_from_plan;
  ACU_UCHAR      diverting_from_presentation;
  ACU_UCHAR      diverting_from_screening;
  ACU_UCHAR      diverting_indicator;              /* ISUP only */
  ACU_UCHAR      original_diverting_reason;        /* ISUP/qsig/NI-2 only */
  ACU_UCHAR      diverting_to_type;                /* ISUP only */
  ACU_UCHAR      diverting_to_plan;                /* ISUP only */
  ACU_UCHAR      diverting_to_int_nw_indicator;    /* ISUP only */
  ACU_UCHAR      original_called_type;             /* ISUP/NI-2 only */
  ACU_UCHAR      original_called_plan;             /* ISUP/NI-2 only */
  ACU_UCHAR      original_called_presentation;     /* ISUP/qsig/NI-2 only */
  ACU_INT        operation;                        /* qsig/H.323 only */
  ACU_INT        operation_type;                   /* qsig/H.323 only */
  ACU_INT        error;                            /* qsig/H.323 only */
  ACU_UCHAR      original_called_screening;        /* NI-2 only */
  ACU_UCHAR      diverting_to_presentation;        /* ISUP drivers only */
  ACU_UCHAR      notification_options;             /* ISUP drivers only */
  char           redirection_addr[MAXNUM];         /* H.323 only */
  DIVERTINGNR         diverting_number;
  ORIGINALCALLEDNR    original_called_number;
} DIVERSION_XPARMS;
```

Parameters

The *diversion_xparms* structure must be used in the following way:

diverting_reason

For ISUP only, the *diverting_reason* field should be supplied with the raw protocol value for the Redirecting Reason parameter, which may vary between regions. Values assigned by ITU include:

0 Unknown/Not available

1 User Busy

2 No Reply

3 Unconditional

For AT&T only, the *diverting_reason* field should be supplied with one of the following values:

DIVERTING_BSY

DIVERTING_RNR

DIVERTING_UNKNOWN

DIVERTING_NETWORK_BSY

For other protocols including QSIG and H.323, the *diverting_reason* field should be supplied with one of the following values:

DIVERTING_BSY

DIVERTING_RNR

DIVERTING_UNKNOWN

DIVERTING_UNCONDITIONAL

For other protocols excluding QSIG, the following parameters may also be valid:

DIVERTING_CD

DEFLECTION_RINGING

DEFLECTION_IMM

diverting_counter
The *diverting_counter* field should be supplied with a value of 1 if this is the first diversion or 2 if this is the second diversion and so on.

NOTE

There may be a protocol dependant upper limit on this parameter.

If multiple diversions have occurred, the *original_called_addr* should be supplied with the number of the original called party details (if available).

diverting_to_addr
Is the address towards which the call is being diverted.

diverting_from_addr
The diverting_from_addr should be supplied with the number of the party diverting the call.

original_called_addr
If multiple diversions have occurred, original_called_addr should contain the number of the original called party (if available).

NOTE

Not currently supported for H.323.

diverting_from_plan
Contains a valid numbering plan for the call and must be appropriate for the signalling system. For example, allowable values may include:

NP_UNKNOWN

NP_ISDN

NP_DATA

NP_TELEX

NP_NATIONAL_STANDARD

NP_PRIVATE

NOTE

Some signaling systems may not have support for all these values, for example, AT&T only supports NT_UNKNOWN and NT_ISDN

The default value is NT_UNKNOWN.

diverting_from_type
Contains a valid numbering type for the call and must be appropriate for the signalling system. For example, allowable values may include:

NT_UNKNOWN

NT_INTERNATIONAL

NT_NATIONAL

NT_NETWORK_SPECIFIC

NT_SUBSCRIBER_NUMBER

NT_ABBREVIATED_NUMBER

NOTE

Some signaling systems may not have support for all values, for example, AT&T only supports NT_INTERNATIONAL and NT_NATIONAL.

The default value is NT_UNKNOWN.

diverting_from_presentation

The *presentation* field indicates the intention of the calling party for the presentation of the calling number (*originating_address*) to the called party. The values are:

PR_ALLOWED (Presentation Allowed)

PR_RESTRICTED (Presentation Restricted)

PR_NOTAVAILABLE (Number not available due to interworking)

PR_RESERVED

NOTE

Some signaling systems may not have support for all values, for example, AT&T only supports PR_ALLOWED and PR_RESTRICTED.

diverting_from_screening

The possible values for the *screening* field are:

SC_NOTSCREENED (User provided, not screened)

SC_VERIFYPASS* (User provided, verified and passed)

SC_VERIFYFAIL (User provided, verified and failed)

SC_NETWORKPROVIDED* (network provided)

diverting_indicator

The diverting_indicator (redirection indicator) is used for ISUP to indicate a call was either re-routed or re-directed.

0 no redirection (national use)

1 call rerouted (national use)

2 call rerouted, all redirection information presentation restricted (national use)

3 call diverted

4 call diverted, all redirection information presentation restricted

5 call rerouted, redirection number presentation restricted (national use)

6 call diversion, redirection number presentation restricted (national use)

original_diverting_reason

The original diversion (redirect) reason, when used for ISUP, indicates the original reason a call was either re-routed or re-directed, the values are:

0 unknown/not available

1 user busy (national use)

2 no reply (national use)

3 unconditional (national use)

For other protocols including QSIG, the following values apply:

DIVERTING_BSY

DIVERTING_RNR

DIVERTING_UNKNOWN

DIVERTING_UNCONDITIONAL

Some other protocols, excluding QSIG, may also use:

DIVERTING_CD

DEFLECTION_RINGING

DEFLECTION_IMM

diverting_to_type

The diverting_to_type for ISUP is equivalent to Q 763 nature of address:

Allowable values include:

NOA_SUBSCRIBER_NUMBER

NOA_NATIONAL_RESERVED

NOA_NATIONAL

NOA_INTERNATIONAL

These are the values defined by Q.767 (International ISUP), although the driver will pass other values transparently.

diverting_to_plan

The diverting_to_plan for ISUP is equivalent to Q 763 numbering plan:

Allowable values include:

NP_ISDN

NP_DATA

NP_TELEX

If other values are encountered, the drivers will pass the value transparently.

diverting_to_int_nw_indicator

The diverting_to_int_nw_indicator for ISUP when set to a non-zero value, indicates that routing to an internal network number is not allowed.

original_called_type

The original_called_type for ISUP is equivalent to Q 763 nature of address:

1 subscriber number (national use)

2 unknown (national use)

3 national (significant) number

4 international number

5 network-specific number (national use)

original_called_plan

The original_called_plan for ISUP is equivalent to Q 763 numbering plan:

1 ISDN (Telephony) numbering plan (Recommendation E.164)

3 Data numbering plan (Recommendation X.121) (national use)

4 Telex numbering plan (Recommendation F.69) (national use)

original_called_presentation

The original_called_presentation for ISUP is equivalent to Q 763 presentation restricted indicator. Available values are:

PR_ALLOWED (Presentation Allowed)

PR_RESTRICTED (Presentation Restricted)

PR_NOTAVAILABLE (Number not available due to interworking)

PR_RESERVED

Operation

The operation field indicates which divert message is being transmitted.

OP_ACTIVATE_DIVERSION

OP_DEACTIVATE_DIVERSION

OP_INTERROGATE_DIVERSION

OP_CALL_REROUTE_REQ - network originating call reroute request

OP_DIVERTING_LEG_INFO1 - call was diverted at gatekeeper

OP_DIVERTING_LEG_INFO2 - call was diverted from another number

OP_DIVERTING_LEG_INFO3 - call was diverted to another number

operation_type

The operation_type field indicates the type of the operation and can take one of the following values:

INVOKE - A request for a service

RETURN_RESULT - An acknowledgement of an INVOKE. The request was successful.

RETURN_ERROR - An acknowledgement of an INVOKE. The request was unsuccessful.

Error

This option is currently not available, reserved for future use.

NOTE

Available for H.323.

original_called_screening

The possible values for the original_called_screening field are:

SC_NOTSCREENED (User provided, not screened)

SC_VERIFYPASS* (User provided, verified and passed)

SC_VERIFYFAIL (User provided, verified and failed)

SC_NETWORKPROVIDED* (network provided)

diverting_to_presentation

For ISUP, indicates the intention of the calling parties presentation of the calling number (originating_address) to the called party. The values are:

PR_ALLOWED (Presentation Allowed)

PR_RESTRICTED (Presentation Restricted)

PR_NOTAVAILABLE (Number not available due to interworking)

PR_RESERVED

notification_option (outgoing calls only)

For ISUP outgoing calls a subsequent exchange may divert the call, in which case the originator may be notified that the diversion has occurred. The diverting exchange may sometimes supply a "Call diversion Information" parameter containing

"notification options", which define how much details the originating subscriber should be told about the diversion.

The definition of parameter values may vary depending upon national variants. Values defined by ITU include:

```
Unknown
Presentation not allowed
Presentation allowed with redirection number
Presentation allowed without redirection number
```

It is not possible to use this field to set notification options in messages generated by Aculab, but it will be set if it is encountered in received messages. If a number of conflicting values are received in different messages during a single call sequence, only the most restrictive value will be presented at the API.

redirection_addr (H.323 calls only)

Contains the number of the diverted-to user. Presentation is dependant upon the presentation restriction rules set by diverted-to user.

`diverting_number`

For diversion using `ETS DIVERTING_LEG_INFO2`, the **`diverting_number`** structure can be supplied with the appropriate protocol values as below:

`presented_number_unscreened`

For ETS only, the `presented_number_unscreened` field should be supplied with one of the following values:

```
0  ets_196_presentation_allowed_number
1  ets_196_presentation_restricted
2  ets_196_number_not_available_due_to_interworking
3  ets_196_presentation_restricted_number
```

`party_number`

For ETS only, the `party_number f`ield should be supplied with one of the following values:

```
0  ets_196_unknown_party_number
1  ets_196_public_party_number
2  ets_196_nsap_encoded_number
3  ets_196_data_party_number
4  ets_196_telex_party_number
5  ets_196_private_party_number
8  ets_196_national_standard_party_number
```

`public_type_of_number`

For ETS only, the `public_type_of_number` field should be supplied with one of the following values:

```
0  ets_196_public_type_of_num_unknown
1  ets_196_public_type_of_num_international_number
2  ets_196_public_type_of_num_national_number
3  ets_196_public_type_of_num_network_specific_number
4  ets_196_public_type_of_num_subscriber_number
6  ets_196_public_type_of_num_abbreviated_number
```

private_type_of_number

For ETS only, the `private_type_of_number` field should be supplied with one of the following values:

```
0  ets_196_private_type_of_num_unknown
1  ets_196_private_type_of_num_level2_regional_number
2  ets_196_private_type_of_num_level1_regional_number
3  ets_196_private_type_of_num_ptn_specific_number
4  ets_196_private_type_of_num_local_number
6  ets_196_private_type_of_num_abbreviated_number
```

original_called_number

For diversion using ETS DIVERTING_LEG_INFO2, the original_called_number structure can be supplied with the appropriate protocol values as below:

presented_number_unscreened

For ETS only, the presented_number_unscreened field should be supplied with one of the following values:

```
0  ets_196_presentation_allowed_number
1  ets_196_presentation_restricted
2  ets_196_number_not_available_due_to_interworking
3  ets_196_presentation_restricted_number
```

party_number

For ETS only, the `party_number f`ield should be supplied with one of the following values:

```
0  ets_196_unknown_party_number
1  ets_196_public_party_number
2  ets_196_nsap_encoded_number
3  ets_196_data_party_number
4  ets_196_telex_party_number
5  ets_196_private_party_number
8  ets_196_national_standard_party_number
```

public_type_of_number

For ETS only, the `public_type_of_number` field should be supplied with one of the following values:

```
0  ets_196_public_type_of_num_unknown
1  ets_196_public_type_of_num_international_number
2  ets_196_public_type_of_num_national_number
3  ets_196_public_type_of_num_network_specific_number
4  ets_196_public_type_of_num_subscriber_number
6  ets_196_public_type_of_num_abbreviated_number
```

private_type_of_number

For ETS only, the `private_type_of_number` field should be supplied with one of the following values:

```
0  ets_196_private_type_of_num_unknown
1  ets_196_private_type_of_num_level2_regional_number
2  ets_196_private_type_of_num_level1_regional_number
3  ets_196_private_type_of_num_ptn_specific_number
4  ets_196_private_type_of_num_local_number
6  ets_196_private_type_of_num_abbreviated_number
```

Special diversion/reroute function

For QSIG only there is feature for responding to call reroute requests from the network. When the network indicates an EV_EXT_DIVERSION event, call_feature_details() will allow an application to see the call reroute request. To respond to a network request for a call reroute see call_feature_send().

feature_type should be set to FEATURE_DIVERSION before calling the function.

If there is diversion information then *operation* will be set to either:

OP_DIVERTING_LEG_INFO2 for the previously supported QSIG diversion information. Reception of this message says that the call was diverted from another number and will show details of where the call was diverted from when available (in *diverting_from_addr*).

Or

OP_CALL_REROUTE_REQ indicating the receipt from the network of a call reroute request. This is a request for the call to be made instead to a different destination address.

diverting_to_addr will show the new number for the requested reroute. *diverting_from_addr* will indicate the number that the call is being rerouted from.

In both cases (OP_DIVERTING_LEG_INFO2 and OP_CALL_REROUTE_REQ) the *diverting_reason* will show the reason for diverting and will be one of:

DIVERTING_BSY

DIVERTING_RNR

DIVERTING_UNKNOWN

DIVERTING_UNCONDITIONAL

diverting_counter will show how many times the call has been diverted, which will be 1 or the number of times the call has been diverted if the call has been diverted more than once.

operation_type will be set to the value of INVOKE.

Return values

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

NOTE

For AT&T the valid fields to use are diverting_from_type, diverting_from_plan, diverting_from_presentation, diverting_from_screening, diverting_reason and diverting_from_addr.

10.4 feature_hold_xparms - Structure for Hold\Retrieve(Reconnect) Information

This field can only be used during the Active stage of a call and should not be used for making an outgoing call.

Synopsis

```
typedef struct feature_hold_xparms
{
  ACU_INT        command;
  ACU_INT        cause;
  union
  {
    struct
    {
      ACU_INT         ts;
      ACU_INT         raw;
      DISPLAY         display;
    } sig_q931;
    struct
    {
      ACU_INT         near_end_hold_status;
      ACU_INT         remote_hold_status;
      ACU_INT         media_hold_status;
    } sig_h323;
  } unique_xparms;
} FEATURE_HOLD_XPARMS;
```

Parameters

Command

For `feature_hold_xparms`, the command field 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`

A union contains protocol specific information that can be transmitted when sending one of the above responses.

cause

This field is used when rejecting a hold request or a reconnect request, and contains the generic reason for the rejection of the incoming or outgoing call going to the `EV_IDLE` or `EV_REMOTE_DISCONNECT` state and will be one of the standard set of `LC_xxxx` clearing causes.

See Appendix E for further details on clear causes.

ts

Is used to specify the timeslot for the call to be held or reconnected. Normally used for ETS300.

raw

This field contains the protocol specific cause for rejecting a hold or a reconnect request. See Appendix J for details on raw formats.

display

The display field can be used to transmit information that may be displayed by the user. (`MAXDISPLAY = 34`)

display example:

```
display.ie[0] = 0x04 (four bytes follow)
display.ie[1] = 0x41
```

```
display.ie[2] = 0x42
display.ie[3] = 0x43
display.ie[4] = 0x44
```

This supplies IA5 information “ABCD”.

near_end_hold_status
Is used to specify the current status of a near end call hold. If set to non zero then a near end hold is in progress.

remote_hold_status
Is used to specify the current status of a remote hold. If set to non zero then a remote hold is in progress.

media_hold_status
Is used to specify the current status of a media hold. If set to non zero then a media hold is in progress.

10.5 feature_transfer_xparms - Call Transfer Information

This field can only be used during the Active stage of a call and should not be used for making an outgoing call.

Synopsis

```
typedef struct feature_transfer_xparms
{
  char                control;
  union
  {
    struct
    {
      ACU_INT         operation;
      ACU_INT         operation_type;
      ACU_INT         error;
      union
      {
        struct
        {
          ACU_INT     LinkID;
        } ets;
      } specific;
    } sig_q931;
    struct
    {
      ACU_INT          failure_code;
      ACU_CODEC        codecs[MAXCODECS];
    } sig_sip;
    struct
    {
      ACU_INT          operation_type;
      ACU_CHAR         link_id[MAXADDR];
      ACU_CHAR         destination_addr[MAXADDR];
      ACU_CHAR         destination_alias[MAXADDR];
      ACU_CHAR         originating_addr[MAXADDR];
      ACU_CHAR         originating_alias[MAXADDR];
      ACU_INT          reason;
      ACU_INT          operation;
    } sig_h323;
  } unique_xparms;
} FEATURE_TRANSFER_XPARMS;
```

Parameters

control

NOTE

Only applicable to ETS300, QSIG and H.323

Used to control how the information is used, for example, `CONTROL_NEXT_CC_MESSAGE` includes specified parameters from this call in the next message. See Appendix I.4 for a typical example. The possible values are:

`CONTROL_DEFAULT`
Facility information element is sent immediately by the current call control function

`CONTROL_NEXT_CC_MESSAGE`
Feature information element is sent in the next call control message that is called (e.g. if `call_disconnect()` is called, it will be attached to the `disconnect message()`

`CONTROL_DEFERRED`
Used with the `call_feature_send()` function, used to delay sending of the facility message until further feature information elements have been added via subsequent `call_feature_send()` function calls.

CONTROL_DEFERRED_SETUP
Used with the call_feature_openout() function, used to delay sending of the setup message until further feature information elements have been added via subsequent call_feature_openout() function calls.

CONTROL_EXTRA_INFO
Used with the call_feature_send() function. Can be used multiple times after CONTROL_DEFERRED to add further feature information elements to the facility message.

CONTROL_EXTRA_INFO_SETUP
Used with the call_feature_send() function. Can be used multiple times after CONTROL_DEFERRED_SETUP to add further feature information elements to the setup message.

CONTROL_LAST_INFO
Used with the call_feature_send() function. Used to indicate this is the last feature information element to be added to the facility message

CONTROL_LAST_INFO_SETUP
Used with the call_feature_send() function. Used to indicate this is the last feature information element to be added to the setup message

Q931 specific information

Operation
The operation field indicates which transfer message is being transmitted.

OP_EXPLICIT_ECT_EXECUTE

OP_ECT_LINK_ID_REQUEST

operation_type
The operation_type field indicates the type of the operation and can take one of the following values:

INVOKE - A request for a service

RETURN_RESULT - An acknowledgement of an INVOKE. The request was successful.

RETURN_ERROR - An acknowledgement of an INVOKE. The request was unsuccessful.

Error
The error field should be used to indicate the reason when a RETURN_ERROR operation type is used. Values include:

FE_NOT_SUBSCRIBED - When service is not subscribed to

FE_NOT_AVAILABLE - Either a looping condition has been identified or internal network restrictions mean that the request cannot be accepted

FE_INVALID_CALL_STATE - Either the call is not in the active state or call is not in Held state

FE_SS_INTERACTION_NOT_ALLOWED - Another supplementary service has been activated and interaction is not permitted in this instance

linked
Link ID is a value assigned to a call by the network side. The value is assigned when the call is involved in a transfer.

SIP specific information

failure_code –
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.

codecs

The application should leave this zeroed if wishes to I) use default codecs if it is the being transferred party or ii) use the codecs offered if it is the transferred-to party. Otherwise setting this field will effect the codecs used in the resulting transferred call.

H.323 specific information

operation

The operation field indicates which transfer message is being transmitted. Valid values are

`OP_EXPLICIT_ECT_EXECUTE`

`OP_ECT_EXECUTE`

`OP_ECT_LINK_ID_REQUEST`

`OP_ECT_SETUP`

`OP_ECT_ABANDON`

operation_type

The operation_type field indicates the type of the operation and can take one of the following values:

`INVOKE` - A request for a service

`RETURN_RESULT` - An acknowledgement of an INVOKE. The request was successful.

`RETURN_ERROR` - An acknowledgement of an INVOKE. The request was unsuccessful.

link_id

The Link ID is supplied by the transferred-to party and identifies the current call transfer. The transferred party presents this number to the transferred-to party when making the transferred call.

destination_addr

The address to which the call is being transferred.

destination_alias

The alias of the endpoint to which the call is being transferred.

originating_addr

The address of the endpoint instigating the transfer.

originating alias

The alias of the endpoint instigating the transfer.

All other fields are reserved for future expansion.

10.6 raw_data_struct - Raw Data information

To *include* raw data information the *feature_information* field should be supplied with the value FEATURE_RAW_DATA.

Synopsis

```
typedef struct raw_data_struct
{
        ACU_INT         length;
        ACU_UCHAR       data[MAXRAWDATA];
        ACU_UCHAR       more;
        ACU_INT         nsf_length;
        ACU_UCHAR       nsf_data[MAXNETSPECIFICFACIL];
} RAW_DATA_STRUCT;
```

The *raw_data_struct* structure must be used in the following way to transmit raw data information in call_feature_openout(), *detail* or *send*:

Parameters

Length

This is the length in bytes of the data being supplied in the data field. The max value is subject to the current data function In most instances the current maximum is 128 bytes.

Data

The data field should be supplied with raw octets of an application specific information element that is required in the setup message (See appendix I.1). The length of this information should be supplied in the length field.

More

If the more field contains a non-zero value then this is an indication to the receiver that more data will follow belonging to the same block.

Nsf_length;

The nsf_length field should be supplied with the length of data needed to specify Network Specific Facilities.

Nsf_data;

The nsf_data field should be supplied with raw octets of Network Specific Facilities

Examples for non NSF and protocol specific information is in Appendix J

Example for NSF

Use call_feature_send when a call is in the connected state to send Network Specific Facilities (NSF) and call_feature_details to get NSF feature details.

```
memset(&feature, 0, sizeof(raw_data_struct));
feature.raw_data.nsf_data[0] = 0x00;
feature.raw_data.nsf_data[1] = 0x49;
feature.raw_data.nsf_data[2] = 0x0E;
feature.raw_data.nsf_data[3] = 0xFF;
feature.raw_data.nsf_length = 4;
```

10.7 mlpp_xparms (ETS300 and QSIG only)

Multilevel precedence pre-emption is used to request or invoke facilities in preference to calls with lower precedence levels. ITU-T recommendation Q.955 refers.

Synopsis

```
typedef struct mlpp_xparms
{
  char                control;
  ACU_INT             operation;
  ACU_INT             operation_type;
  ACU_INT             error;
  ACU_INT             Prec_level;
  ACU_INT             LFB_Indictn;
  char                MLPP_Svc_Domn[5];
  ACU_INT             StatusRequest;
  ACU_INT             Preempt;
} MLPP_XPARMS;
```

Parameters

operation

For `call_feature_openout()`, the operation must be set to `OP_MLPP_CALL_REQUEST`

For `call_feature_send()`, the operation may be set to:

`OP_MLPP_CALL_REQUEST` to send MLPP call request supplementary services

`OP_MLPP_PREEMTION` to send MLPP pre-emption supplementary services

For `call_feature_details()`, either `OP_MLPP_CALL_REQUEST` or `OP_MLPP_PREEMTION` may apply.

operation_type

For `call_feature_openout()`, the operation type must be set to `INVOKE`

For `call_feature_send()`, the operation type may be:

`RETURN_RESULT` or `RETURN_ERROR` when sending MLPP call request supplementary

`INVOKE or RETURN_ERROR` when sending MLPP pre-emption supplementary

For `call_feature_details()`, either `INVOKE`, `RETURN_RESULT` or `RETURN_ERROR` may apply.

error

For `call_feature_send()`, when sending MLPP call request supplementary services, and for `call_feature_details()`, the following error messages may apply:

`MLPP_Error_userNotSubscribed`	0
`MLPP_Error_rejectedByNetwork`	1
`MLPP_Error_unauthorizedPrecedenceLevel`	44

prec_level

For `call_feature_openout()`, the precedence level may be:

`MLPP_Prec_level_flashOverride`	0
`MLPP_Prec_level_flash`	1
`MLPP_Prec_level_immediate`	2
`MLPP_Prec_level_priority`	3
`MLPP_Prec_level_routine`	4

LFB_Indictn

For `call_feature_openout()`, LFB indication may be:

`MLPP_LFB_Indictn_lfbAllowed`	0

```
MLPP_LFB_Indictn_lfbNotAllowed          1
MLPP_LFB_Indictn_pathReservd            2
```

MLPP_Svc_Domn[5]

For `call_feature_openout()`, MLPP service domain must be filled out with 5 octets. The first two octets provide the International ID; the next three octets provide the MLPP Domain identification.

NOTE

LFB_Indictn and MLPP_Svc_Domn do not need to be assigned when the initial setup message is sent.

StatusRequest

For `call_feature_details()` and `call_feature_send()`, when sending MLPP call request supplementatry services when *`operation_type`* is set to `RETURN_RESULT`, status request may be:

```
MLPP_StatusRequest_successCalledUserMLPPSubscriber          1
MLPP_StatusRequest_successCalledUserNotMLPPSubscriber  2
MLPP_StatusRequest_failureCaseA                             3
MLPP_StatusRequest_failureCaseB                             4
```

Preempt

For `call_feature_send()` and `call_feature details`, when sending MLPP pre-emption supplementary services when *`operation`* is set to `OP_MLPP_CALL_REQUEST`, preempt may be:

```
MLPP_preempt_circuitReservedForReuse                1
MLPP_preempt_circuitNotReservedForReuse             2
```

All other fields are reserved for future expansion.

10.8 Non_standard_data_xparms

Synopsis

```
typedef struct non_standard_data_xparms
{
  ACU_INT        id_type;
  ACU_INT        length;
  ACU_CHAR  data[MAXRAWDATA];
  union
  {
    struct
    {
      ACU_USHORT      cc;
      ACU_USHORT      ext;
      ACU_USHORT      code;
    } h221_id;
    struct
    {
      ACU_USHORT      length;
      ACU_ULONG       id[MAXOID];
    } object_id;
  } id;
} NON_STANDARD_DATA_XPARMS;
```

Non standard data specific information

The *non_standard_data_xparms* structure must be used in the following way to provide H.225 information in `call_feature_openout()`, *detail* or *send*:

Parameters

id_type

Should be set to one of the following values:

`NON_STANDARD_ID_TYPE_H221` if an H.221 identifier is to be used

`NON_STANDARD_ID_TYPE_OBJECT` if an object ID is to be used

length

This is the length in bytes of the data being supplied in the data field.

data

The *data* parameter is used to include additional protocol dependant or application specific raw data information.

h221_id

Should contain the H.221 identifier details when `id_type` is set to `NON_STANDARD_ID_TYPE_H221.`

The H.221 identifier includes the elements:

`cc` – T.35 country code

`ext` – T35 extension (assigned nationally)

`code` – manufacturer code (assigned nationally)

object_id

Should contain the object ID details when `id_type` is set to `NON_STANDARD_ID_TYPE_OBJECT.`

The object ID includes the elements:

length – the length of the data

id – the data to be set

10.9 raw_msg_xparms – Send / Receive of raw messages / parameters

NOTE

ISUP 6.5.0 and later only

This structure allows an application to receive ISUP messages in a special Type, Length, Value format. The structure can also be used to insert/alter/remove ISUP parameters in outbound messages, or even send an entire ISUP message for message types not directly supported by Aculab ISUP.

The exact message format transmitted to the network depends upon the ISUP variant selected in the SS7 stack configuration file, and any codec extensions in use.

The use of Type, Length, Value formats in `raw_msg_xparms` allows an application developer to control the data content of messages, without the need to manage ISUP message pointers or determine which parameters are Fixed, Variable or Optional for a given ISUP variant.

To send ISUP raw message / parameter information via `call_feature_openout()` (or `call_feature_send()`), the *`feature_information`* (or *`feature_type`*) field should be supplied with the value `FEATURE_RAW_MSG`.

To retrieve raw messages the *`feature_type`* field should be supplied with the value `FEATURE_RAW_MSG` before calling `call_feature_details()`.

Synopsis

```
typedef struct raw_msg_xparms
{
  ACU_TOKEN           raw_msg_seq;
  ACU_USHORT          length;
  ACU_UCHAR           data[MAXRAWMSGDATA];
} RAW_MSG_XPARMS;
```

Parameters

`raw_msg_seq` (only valid after `call_feature_details()` returns)

This field contains a sequence number that can be paired with the value provided by the field of the same name in `struct state_xparms`. If a call control event occurs as a result of a received network message, an application can use this field to match the call control event with the raw message retrieved with `call_feature_details()`. The first received message for a call is allocated a sequence value of one, with each subsequent received message being given a sequence number one greater than the previous value.

`length`
This is the length in bytes of the data being supplied in the data field.

`data`
This field contains either message or parameter information. The format of this field depends upon the function being used and the value supplied in the `message_control` field of `struct feature_detail_xparms` and `struct feature_out_xparms`.

One or more parameters are encoded as sequences of bytes in Type, Length, Value format. Messages consist of a single message type byte, followed by zero or more parameters.

Values for message type bytes are derived from Table 4/Q.763 or equivalent. Parameter type bytes are derived from Table 5/Q.763 or equivalent.

To alter the parameters sent in an existing message, use the value `CONTROL_NEXT_CC_MESSAGE` in the `message_control` field, and send the required

parameters as sequences of bytes in Type, Length, Value format. To remove a parameter, specify a Length value of zero and omit the Value field.

NOTE

If a mandatory parameter is removed, it must be replaced by a subsequent parameter for the ISUP message to remain valid.

The data returned by `call_feature_details()` always consists of an entire message received from the network; i.e. a single message type byte, followed by zero or more parameters in Type, Length, Value format.

The data supplied in `call_feature_openout()` (or `call_feature_send()`) consists of an entire raw message when the `message_control` field of `struct feature_out_xparms` (or `struct feature_detail_xparms`) is set to one of:

`CONTROL_DEFAULT`

`CONTROL_DEFERRED_MESSAGE`

NOTE

If an entire raw message is sent in call_feature_openout(), it is sent *instead of an IAM*.

The data supplied in `call_feature_openout()` (or `call_feature_send()`) consists of a series of parameters when the `message_control` field of `struct feature_out_xparms` (or `struct feature_detail_xparms`) is set to one of:

`CONTROL_NEXT_CC_MESSAGE`

`CONTROL_DEFERRED`

`CONTROL_EXTRA_INFO`

`CONTROL_LAST_INFO`

NOTE

If the message structure selected by the current ISUP variant (and any ISUP codec extensions) prohibits the use of a parameter, it will be dropped by ISUP. If a mandatory parameter is missing from a message, the message will not be transmitted.

Example 1

Insert two Generic Number parameters (type 0xc0), one of length 6 bytes, the other of length 7 bytes. The data to be sent consists of a series of parameters, so no message type octet is specified. The parameters must be applied to the next call control message, so a message control value of `CONTROL_NEXT_CC_MESSAGE` will be used.

Set `raw_msg.length` to: `17`
Set `raw_msg.data` to: `0xc0,0x06,0x06,0x84,0x00,0x31,0x75,0x09,0xc0,0x07,0x05,0x04,0x00,0x42,0x86,0x31,0x88`
Set `feature_out_xparms.message_control` (or `feature_detail_xparms.message_control`) to: `CONTROL_NEXT_CC_MESSAGE`

Example 2

Remove a Calling Party Number (type 0x0a) from an IAM, and replace the Called Party Number (type 0x04) with the digits “666”.

The data to be sent consists of a series of parameters, so no message type octet is specified. The length for the Calling Party Number is sent as zero, to remove the parameter. The parameters must be applied to the next call control message, so a message control value of `CONTROL_NEXT_CC_MESSAGE` will be used.

Set `raw_msg.length` to: `8`

Set `raw_msg.data` to: `0x0a,0x00,0x04,0x04,0x84,0x00,0x66,0x06`

Set `feature_out_xparms.message_control` to: `CONTROL_NEXT_CC_MESSAGE`

Example 3

Send a Confusion (CFN) message (type 0x2f), including Cause Indicators (type 0x12) of length 3 octets. The cause value in this example is *#103* (Parameter non-existent or not implemented, passed on); the cause diagnostics refer to the Calling Party's Category (type 0x09). The data to be sent consists of an entire message to be sent immediately, so the message control value `CONTROL_DEFAULT` will be used.

Set `raw_msg.length` to: `6`

Set `raw_msg.data` to: `0x2f,0x12,0x03,0x80,0xe7,0x09`

Set `feature_detail_xparms.message_control` to: `CONTROL_DEFAULT`

10.10 call_waiting_xparms – Send and receive call waiting information

This function is used to indicate that the called number is busy and that the called party knows you are waiting.

NOTE
H.323 only

Synopsis

```
typedef struct call_waiting_xparms
{
  ACU_USHORT            waiting_calls;
  union
  {
    UNIQUEX_IPTEL          sig_iptel;
  } ACU_PACK_DIRECTIVE     unique_xparms;
} ACU_PACK_DIRECTIVE CALL_WAITING_XPARMS;
```

This function can be used to send an `ALERTING me`ssage containing embedded call waiting information, as defined in H.450.6

On reception of a call waiting message the extended event EV_EXT_CALL_WAITING is generated. The number of waiting calls can then be queried using `call_feature_details`.

Parameters

`waiting_calls`
This is the number of waiting calls.

10.11 restart_channels_xparms

The channel restart feature can be used to send an ISDN restart message on a specified port.

Synopsis

```
typedef struct restart_channels_xparms
{
  union
  {
    struct
    {
      ACU_UINT        restart_class;
      ACU_INT    ts;       /* included for later use */
    } sig_q931;
  } unique_xparms;
} RESTART_CHANNELS_XPARMS;
```

This feature is supported for NI-2 signalling networks. The net value in the `restart_channels_xparms` passed to `call_feature_send` is used to specify on which port the restart message is to be sent.

Parameters

`Class`

The values this can be set to are:

`RESTART_CLASS_SINGLE_INTERFACE`

`RESTART_CLASS_ALL_INTERFACES`

The class parameter is used to specify whether the restart is for all channels on a port or all channels on an interface. For an NFAS configuration there can be more than one port grouped together in the interface.

NOTE

This feature is generally meant to be used only when a normal call release (using call_disconnect/call_release) has failed.

10.12 addressed_non_standard_data_xparms – Send/receive connectionless non-standard data

This structure is used to send connectionless non-standard data on H.323 ports.

Synopsis

```
typedef struct addressed_non_standard_data_xparms
{
  ACU_USHORT                  sequence_number;
  ACU_CHAR                    remote_address[ACU_MAX_IP_ADDRESS];
  NON_STANDARD_DATA_XPARMS    data;
} ACU_PACK_DIRECTIVE ADDRESSED_NON_STANDARD_DATA_XPARMS;
```

Three types of non-standard message are currently supported:

Message	Feature type
Connectionless `FACILITY`	`FEATURE_CONNECTIONLESS_FACILITY`
H.225 RAS non-standard message	`FEATURE_NSM_RAS`
H.225 RAS XRS	`FEATURE_XRS`

Reception of these message types must be enabled using `call_enable_connectionless`. In all cases an `ACU_CALL_EVT_CONNECTIONLESS` will be generated on the port when data is available to be read, and the data can then be queried using `call_get_connectionless()`.

Parameters

`sequence_number`

This is the sequence number of the message for RAS NSM and XRS messages. If set to zero a sequence number will be chosen by the driver for sent messages; this is recommended for NSM RAS messages.

`remote_address`

The remote system to which the message should be sent or from which the message was received.

`data`

The contents of the message. See section `non_standard_data_xparms` for details.

10.13 feature_activation_xparms – Send/receive a Feature Activation IE

When a call is in the connected state, this structure maybe used by `call_feature_send()/call_feature_details()` to send/receive a Feature Activation information element (IE) in a call control message.

Synopsis

```
typedef struct feature_activation_xparms
{
  union
  {
    struct
    {
      ACU_UCHAR       ie[MAXFEATUREACTIVATION]
      ACU_UCHAR;      message_type;
    } sig_q931;
  } ACU_PACK_DIRECTIVE unique_xparms;
} ACU_PACK_DIRECTIVE FEATURE_ACTIVATION_XPARMS;
```

Parameters

`ie`

Contains the length and contents of the Feature Activation information element.

`ie[0]`	number of bytes following
`ie[1 to MAXFEATUREACTIVATION-1]`	protocol specific information

`message_type`

Indicates the call control message that the Feature Activation information element should be sent in.

NOTE

This feature is only supported by the ETS300 protocol and enabled using the `-cFA` firmware configuration switch. Please see the firmware release notes for more details. Requires v6.5.9 call drivers (or newer) and v4.7.20 ETS300 firmware (or newer).

NOTE

The `message_type` field has been added for future compatibility. Currently the only supported type is `Q931_INFORMATION`.

NOTE

When calling `call_feature_send()` it is not possible to defer the sending of a Feature Activation information element into other call control messages. For the reason, `feature_detail_xparms.message_control` must be set to `CONTROL_DEFAULT`.

10.14 information_request_xparms - Send/receive an Information Request IE

When a call is in the connected state, this structure maybe used by `call_feature_send()/call_feature_details()` to send/ receive an Information Request information element (IE) in a call control message.

Synopsis

```
typedef struct information_request_xparms
{
  union
  {
    struct
    {
      ACU_UCHAR       ie[MAXINFORMATIONREQUEST]
      ACU_UCHAR;      message_type;
    } sig_q931;
  } ACU_PACK_DIRECTIVE unique_xparms;
} ACU_PACK_DIRECTIVE INFORMATION_REQUEST_XPARMS;
```

Parameters

`ie`

Contains the length and contents of the Information Request information element.

`ie[0]`	number of bytes following
`ie[1 to MAXINFORMATIONREQUEST-1]`	protocol specific information

`message_type`

Indicates the call control message that the Information Request information element should be sent in.

NOTE

This feature is only supported by the ETS300 protocol and enabled using the `-cFA` firmware configuration switch. Please see the firmware release notes for more details. Requires v6.5.9 call drivers (or newer) and v4.7.20 ETS300 firmware (or newer).

NOTE

The `message_type` field has been added for future compatibility. Currently the only supported type is `Q931_INFORMATION`.

NOTE

When calling `call_feature_send()` it is not possible to defer the sending of an Information Request information element into other call control messages. For the reason, `feature_detail_xparms.message_control` must be set to `CONTROL_DEFAULT`.

10.15 name_presentation_xparms - Send/receive SS-CNIP and SS-CONP

The Calling and Connected Name Presentation supplementary service (SS-CNIP and SS-CONP) is outlined in the ETS300 238 and ECMA-164 specifications. The name_presentation_xparms structure allows the Calling, Called, Connected and Busy Name Presentation operations to be conveyed in various call control messages.

To send the Calling Name Presentation operation in a SETUP message:

The `call_feature_openout()` function (with `message_control` set to `CONTROL_DEFAULT or CONTROL_DEFERRED_SETUP`) or the `call_feature_send()` function (with `message_control` set to `CONTROL_EXTRA_INFO_SETUP or CONTROL_LAST_INFO_SETUP`) is used.

To send the Called, Connected or Busy Name Presentation operation in either an ALERTING, CONNECT or DISCONNECT message:

The `call_feature_send()` function (with `message_control` set to `CONTROL_NEXT_CC_MESSAGE`) is used, followed by a call to the API function that will trigger the appropriate protocol message (I.e. call_incoming_ringing() for an ALERTING message, call_accept() for a CONNECT message or call_disconnect() for a DISCONNECT message).

To retrieve the Calling, Called, Connected or Busy Name Presentation operation received in a SETUP, ALERTING, CONNECT or DISCONNECT message:

The `call_details()` function is called after the API issues the `EV_INCOMING_CALL_DET`, `EV_OUTGOING_RINGING`, `EV_CALL_CONNECTED` or `EV_REMOTE_DISCONNECT` event. The result from `call_details()` will have its `feature_information` element set to `FEATURE_NAME_PRESENTATION` if a Name Presentation operation was received. The `call_feature_details()` with `feature_type` set to `FEATURE_NAME_PRESENTATION` should then be called to retrieve a populated `name_presentation_xparms` structure.

Synopsis

```
typedef struct name_presentation_xparms
{
ACU_UCHAR   operation;
ACU_UCHAR   type;
ACU_UCHAR   data_length;
ACU_UCHAR   data[MAX_NAME_PRESENTATION_DATA];
ACU_UCHAR   character_set;
} ACU_PACK_DIRECTIVE NAME_PRESENTATION_XPARMS;
```

Parameters

operation

Indicates whether the Name Operation is Calling, Called, Connected or Busy. The following values are applicable:

```
OP_NAME_PRESENTATION_CALLING
OP_NAME_PRESENTATION_CALLED
OP_NAME_PRESENTATION_CONNECTED
OP_NAME_PRESENTATION_BUSY
```

type

Indicates how the Name Presentation operation is constructed. The following values are applicable:

```
NAME_PRESENTATION_TYPE_ALLOWED_SIMPLE
NAME_PRESENTATION_TYPE_ALLOWED_EXTENDED
NAME_PRESENTATION_TYPE_RESTRICTED_SIMPLE
NAME_PRESENTATION_TYPE_RESTRICTED_EXTENDED
```

```
NAME_PRESENTATION_TYPE_NAME_NOT_AVAILABLE
NAME_PRESENTATION_TYPE_RESTRICTED_NULL
```

data_length

This element indicates the amount of octets stored in the `data` array. For `NAME_PRESENTATION_TYPE_NAME_NOT_AVAILABLE` and `NAME_PRESENTATION_TYPE_RESTRICTED_NULL` types, `data_length` is zero. For all other types, data_length maybe any value from 1 to 50 inclusive.

data

This array contains a string of octets relating the name of the Calling, Called, Connected or Busy party. This array is not NULL terminated. `data_length` indicates the amount of octets stored.

character_set

Only applicable when `type` is set to `NAME_PRESENTATION_TYPE_ALLOWED_EXTENDED` or `NAME_PRESENTATION_TYPE_RESTRICTED_EXTENDED`. It can take any value from 0 to 255 inclusive. Typically, the `character_set` will be set to 1, indicating the octets in the `data` use the iso8859-1 character set.

NOTE

This feature is only supported by QSIG and is enabled using the `-cFNP` firmware configuration switch. Requires call drivers v6.5.42 (or newer) and QSIG firmware v1.9.22 (or newer).

11 Function usage

The following section shows how the functions provided may be used to make and monitor calls and the requirements of an application using those functions.

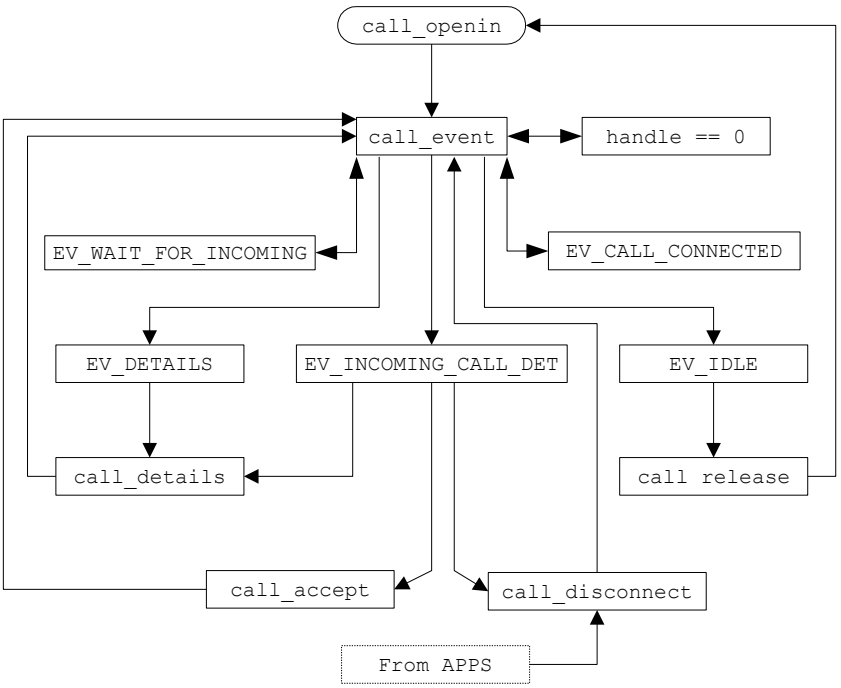
11.1 Event driven applications

Using the V6 driver, all call control is event driven. An application will only receive events for calls that it opened. The existing models of "thread per call" and "single call event thread" can be used in V6. Additionally, it is possible to use event queues to run any arbitrary group of calls in a single thread.

11.2 Call control using events

Wait for incoming call

This diagram shows how an incoming call is set up using the event model.



The application calls the `call_openin()` API function. The driver initiates the wait for incoming call and returns control to the application with the call handle. The application is now free to issue new commands to the device driver.

To ascertain the progress of a call the application should call the `call_event()` function. The device driver will return the state of the call and a call handle on which the event occurred.

The arrival of an incoming call is signalled by the event `EV_INCOMING_CALL_DET`, whereupon the application issues the `call_details()` function that returns information about the incoming call. The application must now decide whether to accept (`call_accept()`) or disconnect (`call_disconnect()`) the incoming call. If the call is accepted by the application the driver connects the call and the `EV_CALL_CONNECTED` event will be generated.

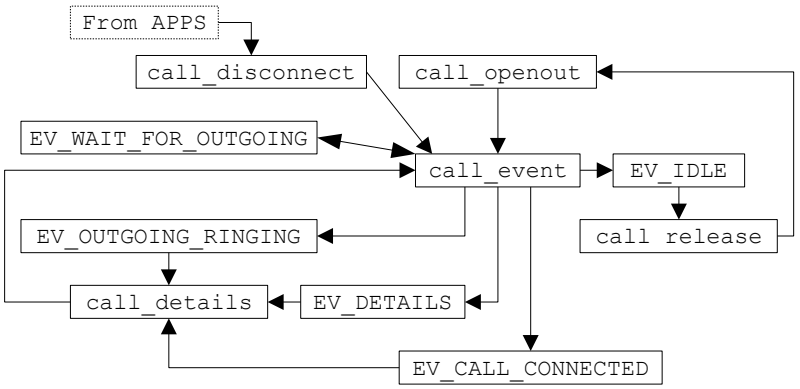
If the call is disconnected, the driver disconnects the call and the `EV_IDLE` event will be emitted. The application must release the handle (`call_release()`) when the

EV_IDLE state is received.

If the driver receives additional call details, the EV_DETAILS event will be generated.

Initiate outgoing call

This diagram shows how an outgoing call is set up using the event model.



The application calls the call_openout() API function. The device driver initiates the outgoing call and returns control to the application. The application is now free to issue new commands to the device driver.

To ascertain the progress of a call the application calls the call_event() function. The device driver will return the state of the call and a call handle on which the event occurred. The EV_OUTGOING_RINGING event will occur when the call has terminated on a subscriber.

This event may not occur depending upon the signalling system and the response time of the called party. The connection of an outgoing call is signalled by the event; EV_CALL_CONNECTED, whereupon the application may optionally invoke the function call_details() if the ts field of outdetails was set to -1, so that the timeslot on which the outgoing call was made can be ascertained.

The application must release the handle (by calling call_release()) when the EV_IDLE state is received.

11.3 Exception handling

As a rule whenever making calls using the Aculab API, the return codes should always be checked.

Certain conditions can arise, through passing incorrect parameters, unavailable timeslots, network problems, operating systems issues, etc, that will be indicated by function return codes.

A list of return codes is found in Appendix A. It is good practice to handle all possible exceptions, also it has proved useful to have some sort of logging or indication to the user during an exception condition.

11.4 Event queues

Creating multiple queues allows an application to arbitrarily divide up events to suit the chosen application model. `acu_allocate_event_queue()` is used to create a new event queue ready to receive events from call *`handles`*, port notifications, and global notifications.

Resources are associated with an event queue using a variety of functions. Calls can be associated with a queue at creation using the *`queue_id`* parameter, for example, in `call_openout()` or `call_openin()`. Similarly, port and global notification events can be associated with a *`queue_id`* using the appropriate `set_*_notification_queue()` call. You can also use the *`notification_queue`* parameter, for example, as used in `acu_open_card()`. Please refer to the full description for `acu_allocate_event_queue()` in the resource API guide or the appropriate call function for further details.

NOTE

There is no need to create an event queue per call handle. Unless otherwise specified, each call handle is associated with the default call event queue for the relevant port. This in turn, unless otherwise specified, will be the global call event queue. Default event queues are also automatically set up for port and global notifications.

12 Supplementary services library

ETS supplementary services

ETS supplementary services are provided through either DLL & LIB files (Windows) or shared objects (Linux). Functions are provided that build ASN.1 supplementary service octet strings. These octet strings can be passed to call control functions that support `FEATURE_FACILITY`, (e.g. `call_feature_openout()`, `call_feature_send()`).

ETS MWI (message waiting indication)

Definitions

`Controlling User`
The user that can activate and deactivate the mwi. Also known as the voice mailbox.

`Receiving User`
The user that the message-waiting indicator is intended for.

Overview

MWI is activated and deactivated by the Controlling User (typically the voice mailbox). The Activate and Deactivate messages are sent to the Receiving User (the user for whom the messages are waiting). Once the Receiving Users network receives an Activate or Deactivate message, it sends an Indicate message to the Receiving User's userside to inform it if messages are available or not.

These MWI messages should be attached to the call control message of a virtual call (see Appendix G:)

12.1 ets_mwi_activate() - make mwi activate message

The message generated by this function call should be sent from the controlling user to the network to indicate that a message is available.

Synopsis

```
int ets_mwi_activate(struct ets_mwi_xparms *mwip);

typedef struct ets_mwi_xparms
{
  int                     length;
  unsigned char           data[225];
  struct ETS_PartyNumber  receivingUserNr;
  struct ETS_PartyNumber  controllingUserNr;
  int                     numberOfMessages;
} ETS_MWI_XPARMS;
```

Input Parameters

`ets_mwi_activate()` takes a pointer, *`mwip`*, to a structure, `ets_mwi_xparms`. The structure must be initialised in the following way before invoking the function.

`receivinguserNr (see ETS_PartyNumber structure 12.4)`
This is the number of the user that will receive the message waiting indication (See `ETS_PartyNumber` for further details).

`controllingUserNr`
This field is not used in `ets_mwi_activate`.

`numberOfMessages` - (optional, see `ETS_PartyNumber` structure `12.4)`
The number of messages available for the receiving user.

Return Value

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

`length`
Will contain the number of octets contained in the field data

`data`
Will contain octets that form the mwi message created.

12.2 ets_mwi_deactivate() - make mwi deactivate message

The message generated by this function call should be sent from the controlling user to the network to remove the message waiting indication.

Synopsis

```
int ets_mwi_deactivate(struct ets_mwi_xparms *mwip);

typedef struct ets_mwi_xparms
{
  int                     length;
  unsigned char           data[225];
  struct ETS_PartyNumber  receivingUserNr;
  struct ETS_PartyNumber  controllingUserNr;
  int                     numberOfMessages;
} ETS_MWI_XPARMS;
```

Input Parameters

`ets_mwi_deactivate()` takes a pointer, *`mwip`*, to a structure, `ets_mwi_xparms`. The structure must be initialised in the following way before invoking the function.

receivinguserNr (see `ETS_PartyNumber` structure 12.4)

This is the number of the user that will have their message waiting indication deactivated (See `ETS_PartyNumber` for further details).

controllingUserNr - (optional, see `ETS_PartyNumber` structure 12.4)

This is the number of the controlling user from which the mwi deactivation is requested.

`numberOfMessages`
This field is not used in `ets_mwi_deactivate`.

Return Value

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

`length`
Will contain the number of octets contained in the data field.

`data`
Will contain octets that form the mwi message created.

12.3 ets_mwi_indicate() - make mwi indicate message

The message generated by this function call should be sent from the receiving users network to the receiving user, to inform of the messages waiting.

Synopsis

```
int ets_mwi_indicate(struct ets_mwi_xparms *mwip);

typedef struct ets_mwi_xparms
{
  int                     length;
  unsigned char           data[225];
  struct ETS_PartyNumber  receivingUserNr;
  struct ETS_PartyNumber  controllingUserNr;
  int                     numberOfMessages;
} ETS_MWI_XPARMS;
```

Input Parameters

`ets_mwi_indicate()` takes a pointer, *`mwip,`* to a structure, `ets_mwi_xparms`. The structure must be initialised in the following way before invoking the function.

`receivingUserNr`
This field is not used in `ets_mwi_indicate.`

`controllingUserNr` - (optional, see `ETS_PartyNumber` structure 12.4)

This is the number of the controlling user.

`numberOfMessages (optional)`
The number of messages available for the receiving user.

Return Value

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

`length`
Will contain the number of octets contained in the data field.

`data`
Will contain octets that form the mwi message created.

12.4 ETS_PartyNumber

This section describes how to fill out the `ETS_PartyNumber` structure. This structure is used in conjunction with the `ets_mwi_xparms` structure.

```
typedef struct ETS_PartyNumber
{
  char      num[20];
  int       type;
  int       sub_type;
} ETS_PARTYNUMBER;
```

num

Should contain a null terminated string, which is equal to the required partynumber.

type

Should contain one of the following values which indicates the type of number:

type_UnknownPartyNumber	0	
type_PublicPartyNumber	1	*
type_DataPartyNumber	2	
type_TelexPartyNumber	3	
type_PrivatePartyNumber	4	*
type_NationalStandardPartyNumber	5	

sub_type

* is only applicable when the type selected is private or public party, as follows:

`type_PublicPartyNumber` - the following sub types can be selected:

sub_type_unknown	0
sub_type_InternationalNumber	1
sub_type_NationalNumber	2
sub_type_NetworkSpecificNumber	3
sub_type_SubscriberNumber	4
sub_type_AbbreviatedNumber	6

`type_PrivatePartyNumberthe` - following sub types can be selected:

sub_type_unknown	0
sub_type_level2RegionalNuber	1
sub_type_level1RegionalNuber	2
sub_type_pTNSpecificNumber	3
sub_type_localNumber	4
sub_type_AbbreviatedNumber	6

If any other type is selected, `sub_type` is ignored.

QSIG supplementary services

QSIG supplementary services are provided through either DLL & LIB files (Windows) or shared objects (Linux). Functions are provided that build ASN1 supplementary service octet strings. These octet strings can be passed to call control functions that support `FEATURE_FACILITY`. (e.g. `call_feature_openout()`, `call_feature_send()`)

Qsig MWI (message waiting indication)

Definitions

`Message Centre` - The unit that can activate and deactivate the MWI depending on messages stored upon it.

`Served User` - The user whom has messages waiting for them, and therefore receives the activate or deactivate message.

OverView

MWI is activated and deactivated by the message centre. These messages are sent to the served user (the user for whom the messages are waiting). These messages are sent via connectionless call control message (see `call_send_connectionless()`)

12.5 qsig_mwi_activate() - make MWI Activate message

The message generated by this function call should sent be from the controlling user to the served user to indicate that a message is available.

Synopsis

```
int qsig_mwi_activate(struct qsig_mwi_xparms *mwip);

typedef struct qsig_mwi_xparms
{
  int            length;
  unsigned char  data[225];
  char           servedUserNr[20];
  char           originatingNr[20];
} QSIG_MWI_XPARMS;
```

Input Parameters

`qsig_mwi_activate()` takes a pointer, *`mwip`*, to a structure, `qsig_mwi_xparms`. The structure must be initialised in the following way before invoking the function.

`servedUserNr` - This is the number of the user that will receive the message waiting indication.

`originatingNr` - (optional) the number of the user that's has left the message.

Return Value

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

`length` - will contain the number of octets contained in the field data

`data` - will contain octets that form the MWI message created.

12.6 qsig_mwi_deactivate() - make MWI Deactivate message

The message generated by this function call should sent from the controlling user to the network to remove the message waiting indication.

Synopsis

```
int qsig_mwi_deactivate(struct qsig_mwi_xparms *mwip);

typedef struct qsig_mwi_xparms
{
  int            length;
  unsigned char  data[225];
  char           servedUserNr[20];
  char           originatingNr[20];
} QSIG_MWI_XPARMS;
```

Input Parameters

`qsig_mwi_deactivate()` takes a pointer, *`mwip,`* to a structure, `qsig_mwi_xparms`. The structure must be initialised in the following way before invoking the function.

`servedUserNr` - This is the number of the user that will receive the message waiting indication.

`originatingNr`
This field is not used in `qsig_mwi_deactivate()`.

Return Value

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

`length` - will contain the number of octets contained in the field data

`data` - will contain octets that form the MWI message created.

Appendix A: Error codes

The following lists the error codes returned by the Aculab APIs. Some errors are internal to the driver occurring only during initialisation and will never be seen by an application.

	Error	Description
0	ERR_NO_ERROR	There was no error
-2	ERR_HANDLE	The handle supplied is invalid or closed, or there are no more handles available.
-3	ERR_COMMAND	The command specified is invalid or was not expected.
-4	ERR_NET	The network outlet number specified is invalid.
-5	ERR_PARM	Inconsistency in the call parameter/illegal parameter
-6	ERR_RESPONSE	Application Failed to respond within response time.
-7	ERR_NOCALLIP	call_details() was issued with no call in progress.
-8	ERR_TSBAR	The specified timeslot is barred from use or an illegal timeslot number was specified.
-9	ERR_TSBUSY	The specified timeslot is already in use or there are no free timeslots.
-10	ERR_CFAIL	Command Failed. An error was detected during the execution of the current command.
-11	ERR_SERVICE	The specified service octet or associated additional information octet is unsupported or is invalid.
-12	ERR_BUFF_FAIL	The driver has run out of data buffer resources. This error should never be seen during normal operation.
-13	ERR_DNLD_ZAP	Failed waiting for board bootsrap code to respond. Check that firmware is appropriate for the hardware.
-14	ERR_DNLD_NOCMD	The command you have attempted could not be executed as there is no signalling system firmware executing on the given network port.
-15	ERR_DNLD_NODNLD	A firmware download was attempted but the signalling system firmware was already installed and running.
-16	ERR_DNLD_GEN	A general error occurred during the downloading of the signalling system firmware. This error should not be seen during normal operations.
-17	ERR_DNLD_NOSIG	The signalling system firmware was downloaded successfully but failed to execute after a start command.

	Error	Description
-18	ERR_DNLD_NOEXEC	The signalling system firmware was downloaded successfully but failed to execute after a start command.
-19	ERR_DNLD_NOCARD	No card is present in the system.
-20	ERR_DNLD_SYSTAT	The signalling system firmware that was downloaded successfully detected an error after a start command. For CAS protocols this may be due to a DSP not being fitted to the Aculab card. Make sure that each card is provided with a suitable clock signal.
-21	ERR_DNLD_BADTLS	The device driver does not support the downloaded signalling system, possibly due to downloading E1 firmware to a T1 port.
-22	ERR_DNLD_POST	Board failed power on self-test.
-23	ERR_DNLD_SW	Switch setup error.
-24	ERR_DNLD_MEM	The call control library could not allocate enough memory from the operating system to start the download procedure.
-25	ERR_DNLD_FILE	The file presented for firmware downloading could not be found by the operating system.
-26	ERR_DNLD_TYPE	The file presented for firmware downloading is not considered to be suitable for downloading to the Aculab card.
-27	ERR_LIB_INCOMPAT	Call driver doesn’t support the version of library API call attempted.
-28	ERR_DRV_INCOMPAT	API call attempted with an incompatible driver (pre v4.0 call drivers).
-29	ERR_DRV_CALLINIT	Another process attempted to call 'call_init' while other processing are accessing the driver.
-30	ERR_TS_BLOCKED	Timeslot blocked
-31	ERR_NO_SYS_RES	Out of OS resources
-32	ERR_PORT_BLOCKED	Port blocked
-40	ERR_INVALID_ADDR	Tried to access invalid address
-41	ERR_INVALID_PORT	Tried to access invalid port
-42	ERR_MANAGEMENT_RPC	Failed to startup Management RPC session
-43	ERR_SESSION_RPC	Failed to startup Session RPC session
-44	ERR_NO_SERVICE	The service did not respond
-45	ERR_NO_BOARD	The board did not respond
-46	ERR_BOARD_UNLOADED	The board has not been downloaded
-47	ERR_BOARD_VERSION	Board software incompatible with host

	Error	Description
		software
-48	ERR_DNLD_CRCERRORS	CRC error detected in board firmware during download
-50	ERR_MAX_APP_LIMIT	Tried to start more than the supported number of applications
-51	ERR_INVALID_FW_PARM	Unrecognised firmware switch
-52	ERR_UNSUPPORTED	Unsupported operation attempted
-53	ERR_NOT_IMPLEMENTED	Function isn't implemented yet
-54	ERR_NO_PLUGINS	Call library found no plugins for communicating with drivers
-55	ERR_DUPLICATE_PLUGINS_FOUND	Call library found multiple plugins with the same id
-56	ERR_NO_PORTS	The call library was unable to find any ports on this card
-57	ERR_DNLD_ALREADY_IN_USE	The port is in use by another host
-58	ERR_DNLD_NOT_RESPONDING	The port is not responding
-59	ERR_NO_FREE_CHANNELS	There are no free channels on this port
-501	ERR_FAIL	An error occurred for which there is no better error code
-502	ERR_NO_MEMORY	Operation failed because insufficient memory was allocated
-503	ERR_CARD_NOT_FOUND	Specified card could not be found
-505	ERR_INVALID_RESOURCE	An invalid resource was specified
-506	ERR_ALREADY_OPEN	Application attempted to open a resource that's already open
-507	ERR_SERVER_NOT_RESPONDING	The server is not responding
-508	ERR_CARD_EJECT_PENDING	The operation has failed because the card is waiting to be ejected
-509	ERR_TIMEOUT	The operation timed out
-510	ERR_STILL_IN_USE	The resource could not be closed as it is still in use
-511	ERR_INVALID_CARD	An invalid card id was specified
-512	ERR_INVALID_CONFIG	A configuration problem occurred
-513	ERR_BUFF_SIZE	The buffer provided is not big enough
-514	ERR_RESOURCE_RELEASED	The resource has been released
-515	ERR_ALREADY_EXISTS	This resource already exists
-516	ERR_LIBRARY_NOT_LOADED	A required library is not loaded or doesn't contain the required function
-517	ERR_ENVIRONMENT_NOT_SET	A required environment variable doesn't exist

	Error	Description
-518	ERR_FILE_ACCESS	File I/O failed
-519	ERR_BOARD_COMM_FAILURE	Communication with board failed
-520	ERR_FILE_FORMAT	Unable to parse file
-521	ERR_BOOTLOADER_FAILED	The firmware bootloader failed
-522	ERR_INTERRUPTED	The operating system has interrupted a system function
-523	ERR_FUNCTION_NOT_FOUND	The specified function is not available
-524	ERR_SERVICE_DEPENDENCY_FAILED	A dependency of the specified service has failed
-525	ERR_SERVICE_UNKNOWN	An unknown service was specified
-526	ERR_SERVICE_RUNNING	The specified service is running
-527	ERR_SERVICE_NOT_RUNNING	The specified service is not running
-528	ERR_SERVICE_EXISTS	The specified service already exists
-529	ERR_SERVICE_START_FAILED	The specified service failed to start
-530	ERR_SERVICE_STOP_FAILED	The specified service failed to stop
-531	ERR_FILE_NOT_FOUND	The specified file could not be found

Appendix B: Service Octets

This appendix describes the standardised set of service octets and additional information octets known to the device drivers.

The values are based on the set of service octets and additional information incorporated for use in: ETS300, TNA, FETEX150 and other Q931 protocols. Any new signalling systems can use the following values and any mapping of these values to actual requirements will take place within the device driver. If more control is required the Bearer, Low Layer and High Layer information elements can be used instead.

Below is a table of the complete list of service octets and additional information octets. This table ***does not*** imply support by the device driver or the signalling system. The table is arranged; service octet with the indented values being the additional information octets associated with the service octet.

```
TELEPHONY         service octet - telephony
ISDN_3K1          3.1khz telephony
ANALOGUE          analogue
ISDN_7K           7Khz telephony

ABSERVICE         service octet - a/b services
FAXGP2            group fax 2
FAXGP3            group fax 3

X21SERVICE        service octet - X.21 Services
UC4               UC 4
UC5               UC 5
UC6               UC 6
UC19              UC 19

FAXGP4            service octet - Fax Group 4
VIDEO64K          service octet - 64Kbits Videotext
DATA64K           service octet - 64Kbits data
X25SERVICE        service octet - X.25 Services
UC8               UC 8
UC9               UC 9
UC10              UC 10
UC11              UC 11
UC13              UC 13
UC19K2            19.2k

TELTEXT64         service octet - Teletext 64
MIXEDMODE         service octet - Mixed Mode
TELEACTION        service octet - Teleaction
GRAPHIC           service octet - Graphic Telephone
VIDEOTEXT         service octet - Videotext
VIDEOPHONE        service octet - Videophone
SOUND_3K1         3.1khz sound
SOUND_7K          7Khz sound
IMAGE             image
```

Appendix C: DASS service indicator codes (SIC)

The SIC is one or two octets in length. If bit 8 is set to 0 the SIC is one octet in length; if set to 1 a second octet follows.

The first octet (sic1) contains routing information, as follows:-

Fixed Combinations

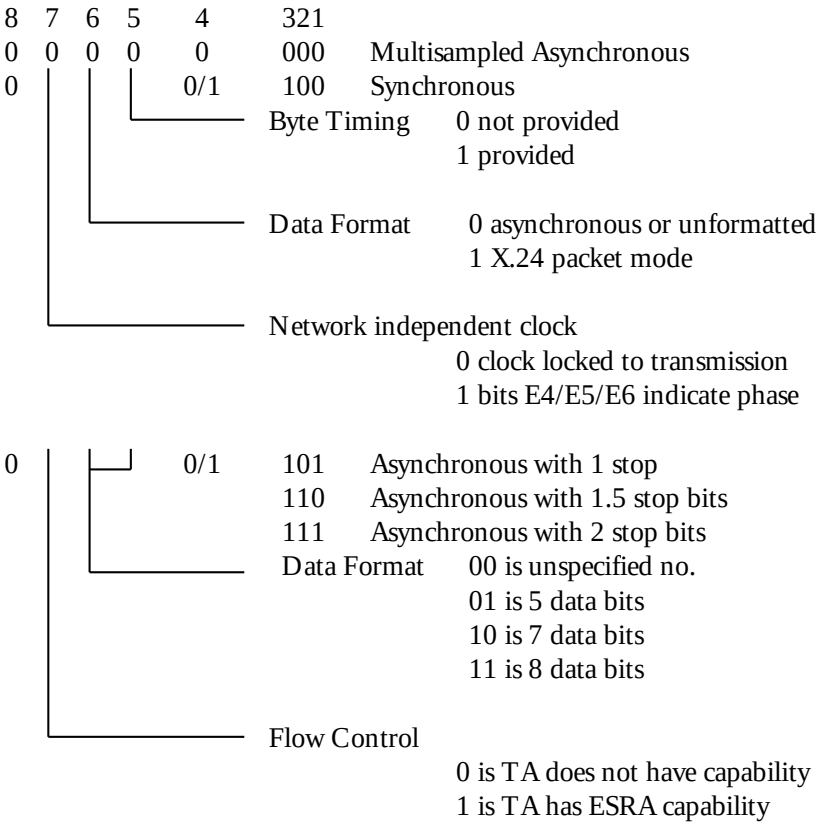
Bits			
8 765		4321	
0 000	speech	0000	A-Law 64 kbit/s (telephony)
0 001	speech	0000	A-Law 64 kbit/s (category 2)
0 001	speech	0010	A-Law 64 kbit/s PCM (category 1)
1 001	speech	1000	3.1 kHz audio at 9.6 kbit/s
1 001	speech	1001	3.1 kHz audio at 8 kbit/s
1 001	speech	1010	3.1 kHz audio at 7.2 kbit/s
1 001	speech	1011	3.1 kHz audio at 4.8 kbit/s
1 001	speech	1100	3.1 kHz audio at 3.6 kbit/s
1 001	speech	1101	3.1 kHz audio at 2.4 kbit/s
1 001	speech	1110	3.1 kHz audio at 1.2 kbit/s
1 001	speech	1111	3.1 kHz audio at 0.6 kbit/s
1 011	data	0000	300 bit/s
1 011	data	0001	200 bit/s
1 011	data	0010	150 bit/s
1 011	data	0011	134.5 bit/s
1 011	data	0100	110 bit/s
1 011	data	0101	100 bit/s
1 011	data	0110	75 bit/s
1 011	data	0111	50 bit/s
1 011	data	1000	75/1200 bit/s (calling->called)
1 011	data	1001	1200/75 bit/s (calling->called)
1 011	data	0000	384 kbit/s
0 010	data	0001	48 kbit/s 6+2 rate adaption
0 010	data	0010	9.6 kbit/s 6+2 rate adaption
0 010	data	0100	4.8 kbit/s 6+2 rate adaption
0 010	data	0101	2.4 kbit/s 6+2 rate adaption
0 010	data	1000	64 kbit/s
0 010	data	1011	8 kbit/s in bit 1
0 010	data	1100	4.8 kbit/s 5-octet frame in bit 1

Bits			
8 765		4321	
0 010	data	1101	2.4 kbit/s ½ 5-oct frame in bit 1
0 010	data	1110	8 kbit/s multi-sampled
1 010	data	1111	64 kbit/s multi-sampled

Plus any of these Bits		
8	765	
1	010	Data
1	100	Teletex
1	101	Videotex
1	110	Facsimile
1	111	SSTV

With any of these Bits	
4321	
0000	64 kbit/s
0001	56 kbit/s
0010	48 kbit/s
0011	32 kbit/s
0100	19.2 kbit/s
0101	16 kbit/s
0110	14.4 kbit/s
0111	12 kbit/s
1000	9.6 kbit/s
1001	8 kbit/s
1010	7.2 kbit/s
1011	4.8 kbit/s
1100	3.6 kbit/s
1101	2.4 kbit/s
1110	1.2 kbit/s
1111	600 bit/s

Second Octet



Bit 4 indicates duplex mode:

0 - full duplex

1 - half duplex

Appendix D: DPNSS service indicator codes (SIC)

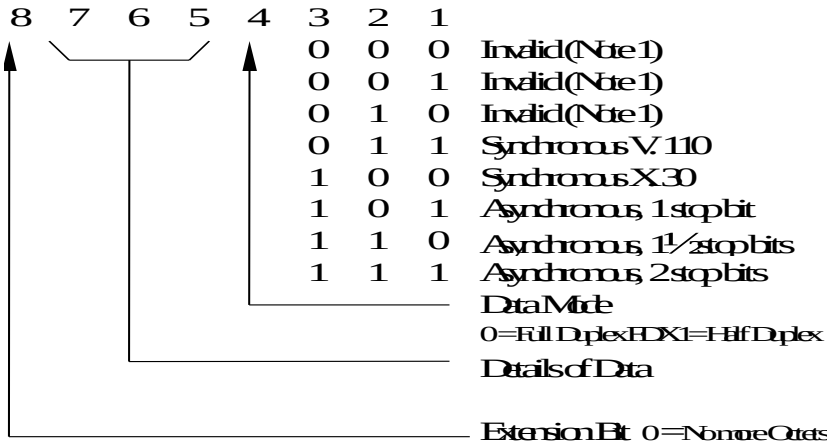
The SIC is one or two octets in length. If bit 8 is set to 0 the SIC is one octet in length; if set to 1 a second octet follows.

SIC 1 (the first SIC octet) contains routing information, as follows:-

Bits			
8765		4321	
0 000	invalid		64 kbit/s PCM G. 711 A-Law or analogue
0 001	speech	0000	32 kbit/s ADPCM G. 721
0 010	speech	0000	64 kbit/s PCM G. 711 u-Law or analogue
1 010	data	0000	64 kbit/s
1 010	data	0001	56 kbit/s
1 010	data	0010	48 kbit/s
1 010	data	0011	32 kbit/s
1 010	data	0100	19.2 kbit/s
1 010	data	0101	16 Kbit/s
1 010	data	0110	14.4 kbit/s
1 010	data	0111	12 kbit/s
1 010	data	1000	9.6kbit/s
1 010	data	1001	8 kbit/s
1 010	data	1010	7.2 kbit/s
1 010	data	1011	4.8 kbit/s
1 010	data	1100	3.6 kbit/s
1 010	data	1101	2.4 kbit/s
1 010	data	1110	1.2 kbit/s
1 010	data	1111	0.6 kbit/s
1 011	data	0000	300 bit/s
1 011	data	0001	200 bit/s
1 011	data	0010	150 bit/s
1 011	data	0011	invalid
1 011	data	0100	110 bit/s
1 011	data	0101	invalid
1 011	data	0110	75 bit/s
1 011	data	0111	50 bit/s
1 011	data	1000	75 bit/s calling to called & 1200 bit/s called to calling
1 011	data	1001	1200 bit/s calling to called & 75 bit/s called to calling
0 100 0 101			Codes 100 – 111 are used when interworking with DASS2. If a PBX receives one of these codes, it will treat the call as if code 010 has been received and repeat the SIC

Bits			
8765		4321	
0 110 0 111			unchanged

Octet 2:



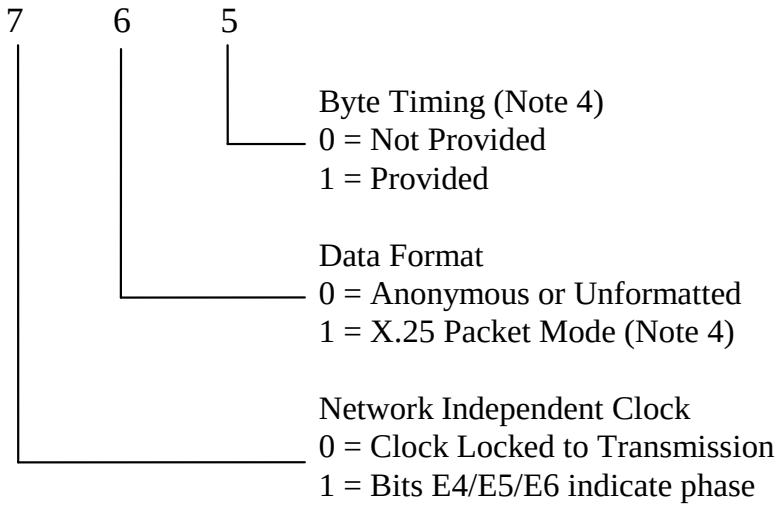
NOTE

Codes 000, 001 and 010 may be used on the Public Network for asynchronous data rate adaptation not complying with ERSA ("ECMA Standard Rate Adaptation").

NOTE

The terms Full Duplex and Half Duplex are used as in CCITT Recommendations X.30 and V.110.

Details of Data for Synchronous



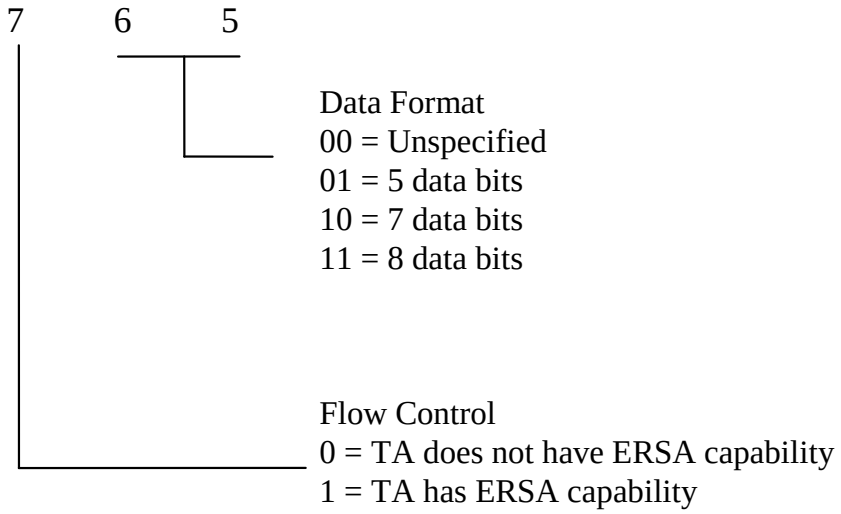
NOTE

This bit is used to indicate whether the originating end is byte aligned to the X.30 frame (typically a character orientated device). If 1, the originating end is byte timed and requires to communicate to a byte timed device. If 0, the originating end is not byte timed (this will apply for V series interfaces, X.21 bis interfaces, and X.21 interfaces that do not have byte timing implemented). If byte timing is provided, the E7 bit in the X.30 frame is used for multiframe synchronisation at 600 bit/s and 1200 bit/s. If byte timing is not provided, the multiframes need not start on true character boundaries.

NOTE

X.25 Packet Mode signifies that data packets plus flags are present in the data bits of the X.30 frames.

Details of Data for Asynchronous



Appendix E: Standard clearing causes

This appendix describes the standardised set of clearing causes supported by the device drivers and available to the application.

Generic clearing causes

The values are based on a set of common clearing causes found in most signalling systems and have been incorporated for use in the call driver. Any new signalling systems and device drivers will also use the standard clearing causes and any mapping of these values to actual requirements will take place within the device driver. In addition, the list of clearing causes may be extended as requirements dictate.

Below is a complete list of generic clearing causes.

```
LC_NORMAL
LC_NUMBER_BUSY
LC_NO_ANSWER
LC_NUMBER_UNOBTAINABLE
LC_NUMBER_CHANGED
LC_OUT_OF_ORDER
LC_INCOMING_CALLS_BARRED
LC_CALL_REJECTED
LC_CALL_FAILED
LC_CHANNEL_BUSY
LC_NO_CHANNELS
LC_CONGESTION
LC_TCP_CONNECT_FAILED
LC_SSL_ERROR
LC_SSL_PEER_CERT_NOT_TRUSTED
LC_SSL_PEER_CERT_INVALID
```

Below are examples of protocol specific, or raw, cause values which helps to show the mapping between these and the generic values above.

Please refer to the appropriate specification for a complete list of protocol specific cause values.

NOTE

This table DOES NOT imply support by every signaling system.

Raw Causes (Q931)

1	Unassigned unallocated number
6	Channel unacceptable
16	Normal Call Clearing
17	User Busy
18	No User Responding
19	No answer from user (user alerted)

21	Call Rejected
22	Number Changed
27	Destination out of Service
28	Invalid Number Format
29	Facility rejected.
30	Response to Status Enquiry
31	Normal Unspecified
34	No Circuit/Channel Available
38	Network Out of Order
41	Temporary Failure
42	Switching Equipment Congested
43	Access information discarded
44	Requested Channel not Available
47	Resource not available
50	Requested facility not subscribed.
57	Bearer Capability not Authorised
58	Bearer Capability not Available
63	Service no Available
65	Bearer Service not Implemented
66	Channel Type not Implemented
69	Requested facility not implemented
70	Restricted Digital Information Only
79	Service not Implemented
81	Invalid Call Reference Value
82	Identified Channel does not Exist
88	Incompatible Destination
95	Invalid Message Specified
96	Mandatory Information Element Missing
97	Message Type non Existent
98	Message Type not Compatible with Call State or not Implemented
99	Information Element non Existent or not Implemented
100	Information Element Content Error
101	Message Type not Compatible with Call State
102	Recovery on Timer Expiry
111	Protocol Error
127	Interworking Unspecified

Raw to generic cause mapping (Q931)

This shows how raw clearing causes are mapped to generic clearing causes

16	LC_NORMAL
1	LC_NUMBER_UNOBTAINABLE
22	LC_NUMBER_CHANGED;
38	LC_OUT_OF_ORDER;
18 & 19	LC_NO_ANSWER;
17	LC_NUMBER_BUSY;
21	LC_CALL_REJECTED;
44	LC_CHANNEL_BUSY;
34	LC_NO_CHANNELS;
42	LC_CONGESTION;
default	LC_CALL_FAILED;

NOTE

All values shown for Q931 clearing causes are in decimal.

Raw causes (DASS2\DPNSS)

0	Number Unobtainable
1	Address Incomplete
2	Network Termination
3	Service Unavaliable
4	Subscriber Incompatible
5	Subscriber Changed Number
6	Invalid Request for S.Service
7	Congestion
8	Sub Engaged
9	Subscriber out of service
10	Incoming Calls Barred
11	Outgoing Calls Barred
18	Remote procedure error
19	Service Incompatible
20	Acknowledgement
21	Signal Not Understood
22	Signal Not Valid
23	Service Temporarily Unavailable
24	Facility Not Registered
25	Reject
26	Message Not Understood
27	Signalling System Incompatible

28	Route Out of Service
29	Transferred
30	NAE Error
31	No Replay from Subscriber
32	Service Termination
35	Channel Out of Service
36	Priority Forced Release
41	Access Barred
45	DTE Controlled not ready
46	DTE Uncontrolled not ready
48	Subscriber Call Termination
50	ET Isolated
51	Local Procedure Error

Raw to generic cause mapping (DASS2\DPNSS)

This shows how raw clearing causes are mapped to generic clearing causes

```
48      LC_NORMAL
0       LC_NUMBER_UNOBTAINABLE
5       LC_NUMBER_CHANGED;
9       LC_OUT_OF_ORDER;
31      LC_NO_ANSWER;
8       LC_NUMBER_BUSY;
25      LC_CALL_REJECTED;
10      LC_INCOMING_CALLS_BARRED;
7       LC_CONGESTION;
default LC_CALL_FAILED;
```

NOTE

All values shown for DASS\DPNSS clearing causes are in decimal

Raw causes (ISUP)

0x01	Unallocated (unassigned) number
0x02	No route to transit network
0x03	No route to destination
0x04	Send special information tone
0x05	Misdialled truck prefix
0x10	Normal call clearing
0x11	User busy
0x12	No user responding
0x13	No answer from user
0x15	Call rejected
0x16	Number changed
0x1B	Destination out of order
0x1C	Address incomplete

0x1D	Facility rejected
0x1F	Normal unspecified
0x22	No circuit available
0x26	Network out of order
0x29	Temporary failure
0x2A	Switching equipment congestion
0x2C	Requested channel not available
0x2F	Resource unavailable, unspecified
0x32	Requested facility not subscribed
0x37	Incoming calls barred within CUG
0x39	Bearer capability not authorized
0x3A	Bearer capability not presently available
0x3F	Service not available, unspecified
0x41	Bearer capability not implemented
0x45	requested facility not implemented
0x46	Only restricted digital information bearer capability is available
0x4F	Service not implemented
0x57	Called user not member of CUG
0x58	Incompatible destination
0x5B	Invalid transit network selection
0x5F	Invalid message unspecified
0x61	Message type nonexistent or not implemented
0x63	Parameter nonexistent or not implemented discarded
0x66	Recovery on timer expiry
0x67	Parameter nonexistent or not implemented passed on
0x6f	Protocol error, unspecified
0x7f	Interworking, unspecified

Raw to generic cause mapping (ISUP)

This shows how raw clearing causes are mapped to generic clearing causes

```
0x10\0x1F       LC_NORMAL
0x01            LC_NUMBER_UNOBTAINABLE
0x16            LC_NUMBER_CHANGED
0x1B            LC_OUT_OF_ORDER
0x12\0x13       LC_NO_ANSWER
0x11            LC_NUMBER_BUSY;
0x15            LC_CALL_REJECTED;
0x2C            LC_CHANNEL_BUSY;
0x22            LC_NO_CHANNELS;
0x2A            LC_CONGESTION;
```

NOTE

All values shown for ISUP clearing causes are in hexadecimal.

Raw Causes (H.323)

H.323 may return either an H.225 Release Complete Reason or a Q.931 Clearing Cause. The `raw_type` parameter in the H.323 unique xparms identifies which it is (H225_RCR or Q931_CAUSE). Additionally there is a helper function in the IP Telephony library called `ipt_translate_h225rcr` that converts H.225 Release Complete Reasons to their nearest Q.931 Clearing Cause equivalent (as defined in the H.323 standard)

`H225_RCR`

1	bandwidth was taken away or ARQ denied
2	Gatekeeper resources have been exhausted
3	a transport path to the destination was not found
4	the call was rejected at the destination
5	invalid revision
6	rejection by the called party's Gatekeeper
7	the Gatekeeper was unreachable
8	Indicates a lack of Gateway resources
9	Indicates an address with an invalid format
10	call is dropping due to LAN crowding
11	Busy
12	the call is being dropped for an undefined reason
13	the call is being dropped due to a facility call deflection
14	the call is being dropped due to a security denial
15	the called party is not registered
16	the caller is not registered

`Q931_CAUSE`

0x01	unallocated (unassigned) number
0x03	a route to the specified destination was not found
0x10	normal call clearing
0x11	the user is busy
0x12	No user responding
0x13	alerting was successful but there was no answer
0x15	call was rejected
0x1C	a number with an invalid format was used
0x3D	the requested facilities were rejected
0x1E	indicates a normal response to a status enquiry
0x1F	a normal but unspecified cause
0x22	no circuit or channel was available

0x29	temporary failure
0x2A	switching equipment congestion
0x2F	the requested resource was unavailable
0x42	Channel type not implemented
0x58	an imcompatible destination was specified
0x60	a mandatory information element was missing in the message
0x61	the message type field was non-existent or the specified message was not implemented.
0x63	an information element was non-existent
0x64	information element contained invalid content
0x65	the message was not compatible with the call state
0x66	recovery on timer expirey

Raw to generic cause mapping (H.323)

This shows how raw clearing causes are mapped to generic clearing causes

```
H225_RCR
12              LC_NORMAL
3\15                 LC_NUMBER_UNOBTAINABLE
11              LC_NUMBER_BUSY
4\6\14          LC_CALL_REJECTED
1\2\8\10             LC_CONGESTION
5\7\9\13\16 LC_CALL_FAILED


Q931_CAUSE

0x10\0x1F            LC_NORMAL
0x01\0x03            LC_NUMBER_UNOBTAINABLE
0x12\0x13            LC_NO_ANSWER
0x11                 LC_NUMBER_BUSY
0x15                 LC_CALL_REJECTED
0x22                 LC_NO_CHANNELS
0x2A                 LC_CONGESTION
default         LC_CALL_FAILED
```

Raw causes (IP Telephony - SIP)

300	multiple choices
301	moved permanently
302	moved temporarily
303	see other
305	use proxy
380	alternative service
400	bad request

401	unauthorized
402	payment required
403	forbidden
404	not found
405	method not allowed
406	not acceptable
407	proxy authentication required
408	request timeout
409	conflict
410	gone
411	length required
413	request message body too large
414	request uri too large
415	unsupported media type
420	bad extension
421	extension required
422	session timer small
480	temporarily not available
481	call leg or transaction does not exist
482	loop detected
483	too many hops
484	address incomplete
485	ambiguous
486	busy here
487	transaction cancelled
488	not acceptable here
500	internal server error
501	not implemented
502	bad gateway
503	service unavailable
504	gateway timeout
505	sip version not supported
600	busy everywhere
603	decline
604	does not exist anywhere
606	global not acceptable

Raw to generic cause mapping (SIP)

This shows how raw clearing causes are mapped to generic clearing causes

2**	LC_NORMAL
3**	LC_NUMBER_CHANGED
486	LC_NUMBER_BUSY
480,481	LC_NUMBER_UNOBTAINABLE
488	LC_CALL_REJECTED
603	LC_CALL_REJECTED
default	LC_CALL_FAILED

Appendix F: Q931 and ISUP

What is Q931?

The ITU document Q931 (DSS1) specifies layer 3 call control for an ISDN user-network interface. This specification is used by different organisations and countries as the basis for layer 3 call control for their networks. These implementations usually have their own specification document and may not implement all the features described in Q931.

Signalling Systems Based On Q931

ETS300 (EuroISDN)

TNA (New Zealand)

FETEX150 (Singapore)

AT&T (North America)

NI2 (North America)

IDAP (Hong Kong)

INS (Japan)

QSIG (Private Signalling Protocol)

Aculab support for Q931

Aculab's Version 5 Call Control API allows support for Q931 based features through the use of Q931 structures. Call control drivers for all of the above signalling systems allow the use of Q931 specific structures and API, in addition to features supported in previous releases of the drivers.

ISUP

Many of the information elements available in Q931 protocols are available in ISUP\SS7 in the same format.

Common Fields For Q931 and ISUP Structures

Fields in these structures allow an application to transparently send and receive Q931 or ISUP based information. These fields are:

```
struct bearer
{
  ACU_UCHAR ie[MAXBEARER];
  ACU_UCHAR last_msg;
};

struct hilayer
{
  ACU_UCHAR ie[MAXHILAYER];
  ACU_UCHAR last_msg;
};

struct lolayer
{
  ACU_UCHAR ie[MAXLOLAYER];
  ACU_UCHAR last_msg;
};

struct progress_indicator
{
  ACU_UCHAR ie[MAXPROGRESS];
  ACU_UCHAR last_msg;
};
```

```
struct notify_indicator
{
  ACU_UCHAR ie[MAXBEARER];
  ACU_UCHAR last_msg;
};
struct keypad
{
  ACU_UCHAR ie[MAXNUM];
  ACU_UCHAR last_msg;
};

struct display
{
  ACU_UCHAR ie[MAXDISPLAY];
  ACU_UCHAR last_msg;
};
```

Sending Information

When encoding these fields for API calls such as `call_openout()`, `xcall_incoming_ringing()` or `call_notify()` the procedure is similar. The protocol information that will be transmitted transparently should be placed in the ie field of the appropriate structure as follows

ie[0] number of bytes following

ie[1..MAX] protocol specific information

bearer example (all values are hexadecimal)

```
bearer.ie[0] = 0x03 (three bytes follow)
bearer.ie[1] = 0x80
bearer.ie[2] = 0x90
bearer.ie[3] = 0xA3
```

This supplies a bearer code for 64kbit A-law speech call.

The `last_msg` field should not be used when sending information

```
bearer.last_msg = 0
```

hilayer example

```
    hilayer.ie[0] = 0x02    (two bytes follow)
hilayer.ie[1] = 0x91
hilayer.ie[2] = 0x81

This supplies high layer information indicating telephony.
```

The `last_msg` field should not be used when sending information

```
   hilayer.last_msg = 0
```

```
lolayer example
   lolayer.ie[0] = 0x03 (three bytes follow)
   lolayer.ie[1] = 0x00
   lolayer.ie[2] = 0xC0
   lolayer.ie[3] = 0x90
```

This supplies low layer information indicating speech, out-band negotiation and 64 kbit.

The `last_msg` field should not be used when sending information

```
   lolayer.last_msg = 0
```

progress_indicator example

progress_indicator.ie [0] = 0x02 (two bytes follow)

```
progress_indicator.ie [1] = 0x80
progress_indicator.ie [2] = 0x83
```

This supplies progress information indicating “Origination address is non ISDN”.

The `last_msg` field should not be used when sending information

```
progress_indicator.last_msg = 0
```

notify_indicator example

```
notify_indicator.ie [0] = 0x01 (one byte follows)
notify_indicator.ie [1] = 0x80
```

This supplies notify information indicating “User Suspended”.

The `last_msg` field should not be used when sending information

```
notify_indicator.last_msg = 0
```

keypad example (ISUP only)

```
keypad.ie[0] = 0x03 (three bytes follow)
keypad.ie[1] = 0x23
keypad.ie[2] = 0x31
keypad.ie[3] = 0x32
```

This supplies IA5 information “#12”.

The `last_msg` field should not be used when sending information

keypad.last_msg = 0

display example

```
display.ie[0] = 0x04 (four bytes follow)
display.ie[1] = 0x41
display.ie[2] = 0x42
display.ie[3] = 0x43
display.ie[4] = 0x44
```

This supplies IA5 information “ABCD”.

The `last_msg` field should not be used when sending information

```
display.last_msg = 0
```

Receiving Information

When extracting information from these fields the information will be in a similar format.

`ie[0]`	number of bytes following
`ie[1..MAX]`	protocol specific information

The `last_msg` field will contain details of the protocol message that delivered this information. Some information (such as progress) can arrive more than once during the lifetime of a call.

Values for these messages include

```
Q931_ALERTING              0x01
```

```
Q931_CALL_PROCEEDING        0x02
Q931_PROGRESS               0x03
Q931_SETUP                  0x05
Q931_CONNECT                0x07
Q931_SETUP_ACK              0x0D
Q931_USER_INFO              0x20
Q931_HOLD                   0x24
Q931_HOLD_ACK               0x28
Q931_HOLD_REJECT            0x30
Q931_RETRIEVE               0x31
Q931_RETRIEVE_ACK           0x33
Q931_RETRIEVE_REJECT        0x37
Q931_DISCONNECT             0x45
Q931_RELEASE                0x4D
Q931_FACILITY               0x62
Q931_NOTIFY                 0x6E
Q931_INFORMATION            0x7B
```

Using Subaddress Information

The information transmitted in the `dest_subaddr`, `orig_subaddr` and `conn_subaddr` fields is different in format from the `destination_addr`, `originating_addr` and `connected_addr` fields.

String handling functions from the standard library can be used to manipulate the `destination_addr`, `originating_addr` and `connected_addr` fields but not the subaddress fields.

To use the `dest_subaddr`, `orig_subaddr` and `conn_subaddr` fields correctly requires knowledge of the way these fields are structured.

structure

The Called Party Subaddress (`dest_subaddr`), Calling Party Subaddress (`orig_subaddr`) and Connected Party Subaddress (`conn_subaddr`) are structured in a similar way.

Bits

8	7	6	5	4	3	2	1	Octets	
0	1	1	1	0	0	0	1	1	Called Subaddress
0	1	1	0	1	1	0	1	or 1	Calling Subaddress
0	1	0	0	1	1	0	1	or 1	Connected Subaddress
L	L	L	L	L	L	L	L	2	Length
1	T	T	T	E	0	0	0	3	
S	S	S	S	S	S	S	S	4-MAX.	

Octet 1 contains the code for the information element. In hexadecimal format this is 0x71 for Called Party Subaddress and 0x6D for the Calling Party Subaddress.

NOTE

When supplying subaddress information through Aculab's API this octet should not be included. The first octet supplied should be the length field, octet 2.

Octet 2 contains the length of the information element. The value will be the number of octets following octet 2.

Octet 3 contains two user-controlled fields

```
1. T T T          Type of subaddress
   0 0 0          NSAP (CCITT Rec. X.213/ISO 8348 AD2)
   0 1 0          User specified


2.     E          Odd/even Indicator
       0          Even number of address signals
       1          Odd number of address signals
```

This field is used when type of subaddress is 'user specified' and the coding is BCD. If the subaddress value is NSAP then this field has no significance.

Octet 4 and above contains the subaddress information.

If the type of subaddress is NSAP then the first octet of subaddress information (Octet 4) will contain an AFI field (Authority and Format Identifier). This is likely to be coded as 50 in hexadecimal.

How to send subaddress information

The Call Control driver expects the subaddress information in a similar format to that described above. Everything described from 'octet 2' onwards needs to be supplied in the `dest_subaddr`, `orig_subaddr` or `conn_subaddr` fields. The first octet that contains the code for the information element should not be supplied (Octet 1 in the diagram of the structure). The choice of whether to use 'NSAP' or 'user defined' types of subaddress is up to the user. If 'NSAP' is to be used then there may need to be an AFI value supplied or else the decoding party may not receive the information as intended.

Example

To transmit the hexadecimal values {31, 32, 33} in the `dest_subaddr` with a *type* of NSAP

'0xnn' represents a hexadecimal value

`dest_subaddr[0]` = length (octet 2 of the Called Party Subaddress description)

`dest_subaddr[1]` = 0x80((0x80|(000<<4)=0x80)

`dest_subaddr[2]` = 0x50(AFI = 0x50)

`dest_subaddr[3]` = 0x31 first value

`dest_subaddr[4]` = 0x32 second value

`dest_subaddr[5]` = 0x33 third value

the value required for the length field in `dest_subaddr[0]` will be 5 (5 octets following).

How to read subaddress information

Received subaddress information has the same format as that for subaddress sent on an outgoing call. To extract the information requires knowledge as to which type of

subaddress has been sent. If the information has been sent with the type of subaddress set to ‘NSAP’ then the first subaddress information octet may contain the AFI field (usually 50 in hexadecimal).

Appendix G: Call independent signalling for euro ISDN & QSIG

QSIG and EuroISDN allow bearer independent signalling connections or virtual calls.

See ETS300 196 8.3.2 for details on EuroISDN.

See ISO 11582 and 7.3 for details on QSIG

Open for an incoming virtual call

To open for an incoming call that is a virtual call, some extra parameters must be used when using `call_openin()`.

The cnf field must have the `CNF_TSVIRTUAL` field set to indicate that a virtual call is expected.

The timeslot field ts must be set to –1.

Open for an outgoing virtual call

To make an outgoing virtual call the function `call_feature_openout()` should be used. To turn this call into a virtual call the `feature_information` field in the `feature_out_xparms` structure should be set in the following way:

EuroISDN

```
feature_information = FEATURE_REGISTER + … .
```

QSIG

```
feature_information = FEATURE_VIRTUAL + … .
```

Example

To make a virtual call in QSIG and send Facility information

```
feature_information = FEATURE_VIRTUAL + FEATURE_FACILITY
```

NOTE

In EuroISDN the outgoing call will go to the connected state immediately.

Accepting a virtual call

EuroISDN

A virtual call will enter the connected state immediately after an incoming call has been detected, `EV_INCOMING_CALL_DET` using `call_event()` for example). It is not necessary to use `call_accept()` to move to the connected state.

QSIG

A virtual call will go through many of the usual call states before going to the connected state. It is necessary to use `call_accept()` to move the call into the connected state.

Clearing a virtual call

`call_disconnect()` can be used to clear a virtual call. A virtual call will clear immediately (`EV_IDLE using call_state()`) rather than go to the remote disconnect state (`EV_REMOTE_DISCONNECT` using `call_event()`).

Appendix H: Generic functional procedures (facility)

Introduction

Some Q.931 based protocols support supplementary services using the Facility information element. This information element when transmitted in a call control message, can be used to invoke and control supplementary services, by sending data based upon different protocols such as Remote Operations Protocol. The call control driver has the ability to include Facility Information elements in `SETUP`, `ALERTING`, `CONNECT`, `DISCONNECT`, `RELEASE` and `FACILITY` messages by using the `call_feature_openout()`, `call_feature_enquiry()` and `call_feature_send()` functions. In addition the `call_feature_details()` function can be used to extract Facility information from the call.

Specifications

The high level definition of the information to be sent in the Facility information element can be obtained from different standards, depending upon the signalling system.

The following documents describe the coding and structure of the Facility information element for two signalling systems.

EuroISDN ETS300 196 Generic functional protocol

QSIG (ISO/IEC 11582, ECMA 165, ETS300 239) Generic functional protocol

The description of each supplementary service is usually provided in a separate standard. These standards also describe the specifics of what kind of data can be sent and how it is structured. Examples of these would be

Call Diversion for EuroISDN ETS300 207-1

Call Diversion for QSIG ISO13872

Abstract Syntax Notation One (ASN.1) is commonly used in supplementary service standards. This is defined in CCITT X.208.

ETSI (http://www.etsi.org/) publish ETS300 documents for EuroISDN and QSIG (among other standards).

ISO publish ISO documents (http://www.iso.ch/).

ECMA documents are available free of charge from http://www.ecma.ch/

Appendix I: Network side call transfer

Introduction

The API functions `call_hold()`, `call_enquiry()` and `call_transfer()` allow a user (the transferring party) to transfer a call to another party. This section describes the way the transferred to or network side handles these messages

NOTE

The following examples apply to Primary Rate EuroISDN only, unless otherwise noted.

Handling a call transfer consists of four stages.

Acknowledgement of a Call Hold request

Accepting an Enquiry Call

Handling a linkID request

Handling a Call Transfer request

After a call has been put on hold, there may also be a request to reconnect or retrieve the call. The following describes the way in which to handle these situations.

I.1 Acknowledgement of a Call Hold request

A request for Call Hold is indicated by the arrival of an extended event `EV_EXT_HOLD_REQUEST`. Either a `HOLD_ACKNOWLEDGE_CMD` or `HOLD_REJECT_CMD` message must answer this request.

Hold Acknowledge

`HOLD_ACKNOWLEDGE_CMD` accepts the Hold Request and allows the user to use that timeslot for another call. This can be accomplished by using the `call_feature_send()` API call as follows

```
struct feature_details_xparms fd;
INIT_ACU_CL_STRUCT(&fd);
fd.handle = state_handle;
fd.feature_type = FEATURE_HOLD_RECONNECT;
fd.feature.hold.command = HOLD_ACKNOWLEDGE_CMD;
rc = call_feature_send(&fd);
```

Hold Reject

`HOLD_REJECT_CMD` declines the Hold Request.

```
INIT_ACU_CL_STRUCT(&fd);
fd.handle = state_handle;
fd.feature_type = FEATURE_HOLD_RECONNECT;
fd.feature.hold.command = HOLD_REJECT_CMD;
fd.feature.hold.cause = 0;
/* reject with cause value 79
   Service or option not implemented, unspecified*/
fd.feature.hold.sig_q931.raw = 79;
rc = call_feature_send(&fd);
```

I.2 Accepting an Enquiry Call

If Hold and Transfer are supported then the application must be ready to accept an incoming call from the user. Under some protocols and with some equipment this means that a second call handle must be opened for a call using the same timeslot as the call on hold.

Accepting an enquiry call is exactly the same as any other incoming call.

I.3 Handling a linkID request

Indication of a Link ID request

NOTE

H.323 Gateway Mode must be enabled to facilitate network operation of H.323 calls.

The arrival of information pertaining to this process is indicated by the arrival of the extended event, `EV_EXT_TRANSFER_INFORMATION`. This event will occur on the handle of the enquiry call.

A call to `call_feature_details()` with `feature_type` set to `FEATURE_TRANSFER` will return the following information for the link ID request

```
operation : OP_ECT_LINK_ID_REQUEST
operation_type : INVOKE
```

Returning a Link ID value

It is the network's responsibility to respond to a link ID request by returning a link ID value. This value is used to identify the enquiry call. The link ID will be included by the user side in the transfer request to identify the other call involved in the transfer.

```
INIT_ACU_CL_STRUCT(&fd);
fd.handle = state.handle;
fd.feature_type = FEATURE_TRANSFER;
fd.feature.transfer.unique_xparms.sig_q931.operation_type = RETURN_RESULT;

fd.feature.transfer.unique_xparms.sig_q931.operation =
OP_ECT_LINK_ID_REQUEST;
fd.feature.transfer.unique_xparms.sig_q931.specific.ets.LinkID = 12;
rc = call_feature_send(&fd);
```

Rejecting a Link ID request

If the network is unable to assign a linkID, it should reject the linkID request in the following way.

```
INIT_ACU_CL_STRUCT(&fd);
fd.handle = state.handle;
fd.feature_type = FEATURE_TRANSFER;
fd.feature.transfer.unique_xparms.sig_q931.operation_type = RETURN_ERROR;
fd.feature.transfer.unique_xparms.sig_q931.operation =
OP_ECT_LINK_ID_REQUEST;
fd.feature.transfer.unique_xparms.sig_q931.error = FE_RESOURCE_UNAVAILABLE;
rc = call_feature_send(&fd);
```

Forwarding a Link ID request

NOTE

This example only applies to H.323 calls.

If H.323 Gateway Mode is enabled, H.323 link ID requests can be sent using call_feature_send. This enables H.323 Call Transfer to be forwarded across a network gateway and can be achieved as follows:

```
INIT_ACU_CL_STRUCT(&fd);
fd.handle = state.handle;
fd.feature_type = FEATURE_TRANSFER;
fd.message_control = <message control>
fd.feature.transfer.unique_xparms.sig_h323.operation_type = INVOKE;
fd.feature.transfer.unique_xparms.sig_h323.operation =
OP_ECT_LINK_ID_REQUEST;


rc = call_feature_send(&fd);
```

The `message_control` field of `feature_transfer_xparms` can be set to `CONTROL_DEFAULT` to send the link ID request immediately or to `CONTROL_NEXT_CC_MESSAGE` to defer the link ID request until the next call control message is sent.

I.4 Handling a Call Transfer request

Indication of Transfer request

NOTE

H.323 Gateway Mode must be enabled for network operation of H.323 calls.

The arrival of information pertaining to this process is indicated by the arrival of the extended event, `EV_EXT_TRANSFER_INFORMATION`.

A call to `call_feature_details()` with `feature_type` set to `FEATURE_TRANSFER` will return the following information for Call Transfer using the Implicit Linkage procedure from ETS300 369

```
operation : ECT_EXECUTE
operation_type : INVOKE
LinkID : <value assigned in stage 3>
```

This indicates that a request for transfer has occurred. This should occur on the handle of a call on hold. The other call involved in the call transfer is identified by the Link ID.

Transfer – Parties A, B and C

If A, B and C are the three parties involved in the transfer and A – B is the initial call that is on hold, then A – C is the enquiry call. A successful call transfer will result in a new connection for parties B and C and clear the two calls involving A.

Accepting a Call Transfer

Step 1

To accept the transfer, party B must be held and party c must be in either a connected or alerting state. The party that made the transfer request requires an acknowledgement that the call was transferred successfully. This can be achieved using the `feature_type control` parameter as follows:

```
INIT_ACU_CL_STRUCT(&fd);
fd.handle = state.handle;
fd.feature_type = FEATURE_TRANSFER;
/* Defer till next call control message */
fd.feature.transfer.control = CONTROL_NEXT_CC_MESSAGE;
fd.feature.transfer.unique_xparms.sig_q931.operation_type = RETURN_RESULT;
fd.feature.transfer.unique_xparms.sig_q931.operation =
OP_EXPLICIT_ECT_EXECUTE;
rc = call_feature_send(&fd);
```

The information that will be transmitted in the next `DISCONNECT` message has now been set up.

Step 2

The next step sends the `DISCONNECT` message. This automatically includes the information that will tell the user that the call transfer was successful. To do this just requires the use of `call_disconnect()`. The other call must also be cleared with `call_disconnect()`.

Rejecting a Call Transfer Request

To reject a call transfer request a `FACILITY` message must be returned to the requesting party indicating the reason for refusing the request.

This can be achieved as follows

```
INIT_ACU_CL_STRUCT(&fd);
fd.handle = state.handle;
fd.feature_type = FEATURE_TRANSFER;
fd.feature.transfer.unique_xparms.sig_q931.operation_type = RETURN_ERROR;
fd.feature.transfer.unique_xparms.sig_q931.operation =
OP_EXPLICIT_ECT_EXECUTE;
fd.feature.transfer.unique_xparms.sig_q931.error = ERROR;
rc = call_feature_send(&fd);
```

The `error` value can be one of the following

`FE_NOT_SUBSCRIBED 0`
When the service has not been subscribed

`FE_NOT_AVAILABLE 3`
Either a looping condition has been identified

Or internal network restrictions mean that the request cannot be accepted

`FE_INVALID_CALL_STATE 7`
Either the call is not in the active state

Or call is not in Held state

`FE_SS_INTERACTION_NOT_ALLOWED 10`
Another supplementary service has been activated and interaction is not permitted in this instance

`FE_LINKID_NOT_ASSIGNED_BY_NETWORK` 25

The link ID received in the transfer request was not issued by the network.

Responding to a Reconnect Request

Similar to Hold, a Reconnect Request is indicated by the arrival of an extended event, `EV_EXT_RECONNECT_REQUEST`. Again this requires a response – either `RECONNECT_ACKNOWLEDGE_CMD` or `RECONNECT_REJECT_CMD`.

Forwarding a Call Transfer request

NOTE

This example only applies to H.323 calls.

If H.323 Gateway Mode is enabled, H.323 call transfer requests can be sent using call_feature_send. This enables H.323 Call Transfer requests to be forwarded across a network gateway and can be achieved as follows:

```
INIT_ACU_CL_STRUCT(&fd);
fd.handle = state.handle;
fd.feature_type = FEATURE_TRANSFER;
fd.message_control = <message control>
fd.feature.transfer.unique_xparms.sig_h323.operation_type = INVOKE;
```

```
fd.feature.transfer.unique_xparms.sig_h323.operation = OP_ECT_EXECUTE;

strcpy(fd.feature.transfer.unique_xparms.sig_h323.link_id, <link_id>);

strcpy(fd.feature.transfer.unique_xparms.sig_h323.destination_addr,
<dest_addr>)

strcpy(fd.feature.transfer.unique_xparms.sig_h323.destination_alias,
<dest_alias>)

rc = call_feature_send(&fd);
```

Where `<dest_addr>` and *`<dest_alias>`* are the address and alias of the destination endpoint and *`<link_id>`* is the link ID to be forwarded.

The `message_control` field of `feature_transfer_xparms` can be set to `CONTROL_DEFAULT` to send the link ID request immediately or to `CONTROL_NEXT_CC_MESSAGE` to defer the link ID request until the next call control message is sent.

I.5 Handling a Call Transfer Setup request

Indication of transferred call setup

NOTE

The following applies to H.323 only.

NOTE

H.323 Gateway Mode must be enabled for network operation of H.323 calls.

The arrival of information pertaining to this process is indicated by the arrival of the extended event `EV_EXT_TRANSFER_INFORMATION`.

A call to `call_feature_details()` with `feature_type` set to `FEATURE_TRANSFER` will return the following information for the Call Transfer setup

```
link_id:  <value assigned by transferred-to endpoint>
source_address: <address of transferred endpoint>
destination_address: <address of transferred-to endpoint>
```

A call transfer setup message can be forwarded to the destination endpoint by calling `call_feature_openout` with `feature_type` set to `FEATURE_TRANSFER`, `sig_h323.operation` set to `OP_ECT_SETUP` and `sig_h323.operation_type` set to `INVOKE` as follows:

```
INIT_ACU_CL_STRUCT(&fd);
...
/* setup call openout parms */
...
fd.feature_type = FEATURE_TRANSFER;
fd.feature.transfer.unique_xparms.sig_h323.operation_type = INVOKE;
fd.feature.transfer.unique_xparms.sig_h323.operation = OP_ECT_SETUP;
strcpy(fd.feature.transfer.unique_xparms.sig_h323.link_id, <link ID>)
rc = call_feature_openout(&fd);
```

Where *<link ID>* is the link ID provided by the transferred-to endpoint.
It is the network's responsibility to respond to a call transfer setup request with either a call transfer connect or a call transfer reject.

I.6 Handling a Call Transfer Setup response

Indication of transferred call setup

NOTE

The following example applies to H.323 only.

NOTE

H.323 Gateway Mode must be enabled for network operation of H.323 calls.

The arrival of information pertaining to this process is indicated by the arrival of the extended event `EV_EXT_TRANSFER_INFORMATION` or `EV_TRANSFER_REJECT`. No feature details are available for these events; the user application must match response events with a previous request.

Accepting a Call Transfer Setup

Step 1

A call transfer setup can be accepted using call_feature_send as follows:

```
INIT_ACU_CL_STRUCT(&fd);
fd.handle = state.handle;
fd.feature_type = FEATURE_TRANSFER;
/* Defer till next call control message */
fd.message_control = CONTROL_NEXT_CC_MESSAGE;
fd.feature.transfer.unique_xparms.sig_h323.operation_type = RETURN_RESULT;
fd.feature.transfer.unique_xparms.sig_h323.operation = OP_ECT_SETUP;
rc = call_feature_send(&fd);
```

The information that will be transmitted in the next `Q931` message has now been set up. The `message_control` field may alternatively be set to `CONTROL_DEFAULT`, in which case the call transfer setup accept message will be appended to a Q931 `ALERTING` message and sent immediately.

Step 2

The next step sends the `CONNECT or ALERTING` message. This automatically includes the information that will tell the user that the call transfer was successful. To do this just requires the use of `call_accept()` or `call_incoming_ringing()`, respectively.

Appendix J: Raw data format

Introduction

This section describes how to fill out the `raw_data_struct`, which is used by the `call_feature_*` functions.

NOTE
Currently only supported on QSIG, ETS300, NI-2 and AT&T

Generic Structure

Below is the general format that needs to be followed in order to apply raw data to a protocol message.

Octet 1	Embedded (0x01) or Append (0x02)
Octet 2	Codeset shift octet (e.g. 0x9E or 0x96)
Octet 3	Information element
Octet 4	Length
Octet 5	Data octet 1
Octet 5+n	Data octet 1+n

Once the `raw_data_struct` has been filled out, calling either `call_feature_send()` or `call_feature_openout()` will send the data. It's possible to have multiple embedded octets, but only one appended octet. If an appended element is included it must be after any embedded octets in the structure.

Codesets supported

Codeset 6 supported by QSIG and ETS300

0x96	Locking shift
0x9E	Non locking shift

Codeset 7 supported by ETS300

0x97	Locking shift
0x9F	Non locking shift

Codeset 5, 6 and 7 supported by NI-2

Only ‘Append’ supported

0x95 Locking shift

0x96 Locking shift

0x97 Locking shift

Codeset 6 and 7 supported by AT&T

0x96 Locking shift

0x97 Locking shift

Example Appended Data

Below shows how to fill out the `raw_data_struct` in order to append the following octets on to the end of a protocol message (`0x96,0x05,0x02,0x12,0x1E`)

```
raw_data_struct.data[0] = 0x02;          /* Appended information element */
raw_data_struct.data[1] = 0x96;          /* locking shift */
```

```
raw_data_struct.data[2] = 0x05;          /* Information Element */
raw_data_struct.data[3] = 0x02;          /* Length */
raw_data_struct.data[4] = 0x12;          /* Data */
raw_data_struct.data[5] = 0x1E;          /* Data */

raw_data_struct.length = 6;
```

Example Embedded Data

Below shows how to fill out the `raw_data_struct` in order to embed the following octets in a protocol message (`0x9E,0x4D,0x02,0x2F,0xD1`)

```
raw_data_struct.data[0] = 0x01;          /* Embedded information element */
raw_data_struct.data[1] = 0x9E;          /* Non locking Shift */
raw_data_struct.data[2] = 0x4D;          /* Information Element */
raw_data_struct.data[3] = 0x02;          /* Length */
raw_data_struct.data[4] = 0x2F;          /* Data */
raw_data_struct.data[5] = 0xD1;          /* Data */

raw_data_struct.length = 6;
```

Example Embedded and Appended Data.

Below shows how to fill out the `raw_data_struct` in order to embed the octets (`0x9E,0x14,0x02,0x01,0x02,0x9E,0x21,0x01,0x4E`) and append the octets (`0x96,0x12,0x02,0x03,0x04,0xE1,0x01,0x03`) in a protocol message

```
raw_data_struct.data[0] = 0x01;          /* Embedded information element */
raw_data_struct.data[1] = 0x9E;          /* Non locking Shift */
raw_data_struct.data[2] = 0x14;          /* Information Element */
raw_data_struct.data[3] = 0x02;          /* Length */
raw_data_struct.data[4] = 0x01;          /* Data */
raw_data_struct.data[5] = 0x02;          /* Data */
raw_data_struct.data[6] = 0x01;          /* Embedded information element */
raw_data_struct.data[7] = 0x9E;          /* Non locking Shift */
raw_data_struct.data[8] = 0x21;          /* Information Element */
raw_data_struct.data[9] = 0x01;          /* Length */
raw_data_struct.data[10] = 0x4E;    /* Data */
raw_data_struct.data[11] = 0x02     ;    /* Appended information element */
raw_data_struct.data[12] = 0x96     ;    /* locking shift */
raw_data_struct.data[13] = 0x12     ;    /* Information Element */
raw_data_struct.data[14] = 0x02     ;    /* Length */
raw_data_struct.data[15] = 0x03     ;    /* Data */
raw_data_struct.data[16] = 0x04     ;    /* Data */
raw_data_struct.data[17] = 0xE1;    /* Information Element */
raw_data_struct.data[18] = 0x01     ;    /* Length */
raw_data_struct.data[19] = 0x03     ;    /* Data */

raw_data_struct.length = 20;
```

The above example shows 2 non locking shift elements that will be embedded into the message and one locking shift octet containing two information elements that will be appended to the message.

Appendix K: TiNG media configuration

K.1 Introduction

In conjunction with Prosody X and Prosody S, it is now possible for applications to take control of the media resources away from the call API. This allows the application to be responsible for the allocation and de-allocation of media resources, while the call API will simply configure the supplied media according to the parameters negotiated during the call setup.

NOTE
This is only applicable to IP Telephony protocols using the Call API

K.2 Benefits

There are several reasons why an application may wish to have separate control of the media resources

- Potentially more efficient use of resources
- Direct access to TiNG VMP objects, which is particularly advantageous for applications such as IP to IP gateways.
- Allows the application to bypass the TiNG Media Resource Manager
- For setups such as on-board H.323 it reduces the system load on the board, and may therefore reduce call setup times.

K.3 The traditional call API for IP telephony calls

Previously, an application's only access to the configured media on an IP Telephony call was via the stream and timeslot returned from the call API. While for traditional telephony this makes perfect sense, for IP Telephony calls it is encapsulating rather a lot.

For example, when using Prosody X and the traditional Call API there are a number of resources being reserved (although they will not necessarily be allocated) via the TiNG resource manager. These may include echo cancellation, multiple codec types, tone detection, multiple channels, TDM resources and tone generation. Once this call has been setup the only access to the media is via the stream and timeslot, effectively reducing the application's control of that media stream to switching only.

With this setup the media is allocated, configured and de-allocated by the call API. This may be a useful abstraction in an IP to TDM gatewaying scenario, and significantly aids porting of TDM based applications, but it can be limiting in other applications.

K.4 TiNG media configuration

Prosody X and Prosody S provide a powerful IP Media API called the Prosody RTP processing API. This allows for the creation of VMP (Voice Media Processing) objects that can handle incoming or outgoing IP media. These can then be connected to each other or to traditional Prosody channels etc.

The call API now allows these VMPs to be passed into it: one for incoming media and one for outgoing media. When the call has been setup, the VMPs will be configured according to the parameters negotiated during call setup (this includes the codec type, packet length, VAD settings and the remote RTP address). In short, the call API will only *configure* these parameters, the VMP objects are owned by the user application, which is responsible for allocating them, de-allocating, and connecting them in whatever way it sees fit.

Essentially 'under the hood' the call API uses:

```
int sm_vmprx_config_codec_xxx(…);
(e.g. sm_vmprx_config_codec_mulaw(SM_VMPRX_CODEC_MULAW_PARMS* parms);)

int sm_vmptx_config_codec_xxx(…);
(e.g. sm_vmptx_config_codec_mulaw(SM_VMPTX_CODEC_MULAW_PARMS* parms);)

int sm_vmptx_config(SM_VMPTX_CONFIG_PARMS* configp);
```

This means that while RTP and RTCP TOS will be configured from the `media_settings` set by the user, the other settings such as DTMF detection, echo cancellation, echo suppression TDM encoding etc. will not be configured.

K.5 How to use TiNG media configuration with a system wide port

Since the application now controls on which ports media will be allocated, the call API allows for a special system-wide IP Telephony port to be opened to support these calls. This is done via:

```
ACU_ERR call_open_iptel_port(CALL_OPEN_IPTEL_PORT_PARMS* port_parms)

typedef struct tCALL_OPEN_IPTEL_PORT_PARMS
{
  ACU_ULONG      size;              /* IN */
  ACU_INT        protocol_type;     /* IN */
  ACU_PORT_ID         port_id;      /* OUT */
} CALL_OPEN_IPTEL_PORT_PARMS;
```

Where `protocol_type` is either SIP (S_SIP) or H.323 (S_H323). The returned `port_id` is then used, like any other port, with `call_openin` and `call_openout`.

The incoming and outgoing TiNG media resources are created via the TiNG Prosody RTP processing API (see that documentation for more specific information).

```
int sm_vmprx_create(SM_VMPRX_CREATE_PARMS rxcreatep)

typedef struct sm_vmprx_create_parms
{
  tSMVMPrxId          vmprx;
  tSMModuleId         module;
} SM_VMPRX_CREATE_PARMS;

int sm_vmptx_create(SM_VMPTX_CREATE_PARMS txcreatep)

typedef struct sm_vmptx_create_parms
{
  tSMVMPtxId          vmptx;
  tSMModuleId         module;
} SM_VMPTX_CREATE_PARMS;
```

The vmptx and vmprx returned from these are then passed into `call_openout`, `call_accept`, `call_progress` or `xcall_incoming_ringing`. E.g.

```
out_parms.unique_xparms.sig_iptel.vmprxid = TiNG_TO_ACU_POINTER(vmprx);
out_parms.unique_xparms.sig_iptel.vmptxid = TiNG_TO_ACU_POINTER(vmptx);
```

or

```
accept_parms.unique_xparms.sig_iptel.vmprxid = TiNG_TO_ACU_POINTER(vmprx);
accept_parms.unique_xparms.sig_iptel.vmptxid = TiNG_TO_ACU_POINTER(vmptx);
```

NOTE

The macro TiNG_TO_ACU_POINTER **is provided to convert** tSMVMPtxId **and** tSMVMPtxId **to** ACU_POINTER

Before the `EV_CONNECTED` event is received the VMP objects will have been configured.

After the call has been released the application may destroy the VMP resources when it likes via:

int sm_vmptx_destroy(tSMVMPtxId vmptx)

int sm_vmprx_destroy(tSMVMPrxId vmprx)

K.6 On-board H.323

Obviously for on-board H.323 a system wide port does not make sense. In these cases calls are opened as normal on the on-board H.323 port. For outgoing calls VMP ids are passed in with call_openout, however for incoming calls it is necessary to set the VMP ids to `ACU_WILL_PROVIDE_VMP`.

```
in_parms.unique_xparms.sig_iptel.vmprxid = ACU_WILL_PROVIDE_VMP;
in_parms.unique_xparms.sig_iptel.vmptxid = ACU_WILL_PROVIDE_VMP;
```

This ensures that the call API will not attempt to allocate any resources before `EV_INCOMING_CALL_DET` is raised. The VMP ids to be configured can then be passed in with `call_accept`, `xcall_incoming_ringing` or `call_progress` as with the system port.

Appendix L: H.323 registration

This section includes guidance on H.323 registration with Aculab's H.323 products.

L.1 Adding Aliases

For best results, it is advised to add aliases before calling `set_h323_gatekeeper`. This means that after the first RCF is received all of your aliases will be registered, and removes the requirement to send additional RCFs for each `add_alias` request.

Aliases are registered with the service and persist across all applications using the system. For on board H.323 this means that the port registrations are valid for all applications using that port

L.2 Alias Format

Aliases are defined in URI format: `<scheme>:<alias name>`

Valid schemes are:

Scheme	Meaning
"h323"	H.323 URI
"mailto"	Email address
"http"	URL
"h323id"	H.323 ID
"tel"	E.164 number

Alternatively, an IP address or a hostname may be provided. If no scheme is given, the system will try to "guess". If it is entirely numeric it will be assumed to be an E.164 number, if it is a dotted quad it will be assumed to be an IP address, if we can resolve it, it will be assumed to be a hostname. If all of these fail, an `ERR_PARM` will be returned.

L.3 Removing Aliases and Clearing the Gatekeeper

When an application controlling registration exits, it should first remove it's aliases, then clear the gatekeeper (if it set it) and delete any aliases it registered. Alternatively, an application should on start up deal with the current registration state being active, and/or aliases already being present.

Removing an alias does not delete it from the system. This allows the application to check it's status if any problems develop during un-registration. Aliases, which are no longer used, should be deleted. This is somewhat analogous to `call_disconnect` and `call_release` -- there is a separation between un-registering and deleting associated resources.

Clearing the gatekeeper does not delete aliases, although it will remove all registered aliases. After clearing a gatekeeper, it is good practise to delete any aliases, which are no longer necessary. If they are not deleted, they will be re-registered the next time `set_h323_gatekeeper` is called. This is to facilitate manual gatekeeper failovers.

L.4 General Points to Note

It is possible to get a list of all current aliases by using `snapshot_registrations`. If you do not know for sure if another application has been storing aliases, or if you are unsure if a previous run of your application cleaned up correctly, this can be used to discover which aliases currently exist in the system.

We generally try to cope with applications trying to re-add aliases that are already present, and return the same handle.

If you are rapidly clearing and then setting the H.323 gatekeeper then it is required to wait for the un-registration to complete before calling `set_h323_gatekeeper`.

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

Certificate number FS722030
ISO 9001:2015