aculab

ACULAB.COM

# Aculab SS7

## Developer's Guide

MAN1200 Revision 6.16.1

## Document Revision

| Rev | Date | By | Detail |
|---|---|---|---|
| 1.0.0 | 28.04.06 | DJL | First issue |
| 1.0.1 | 12.06.06 | DJL | Updates to section 8 |
| 6.8.3 | 19.03.07 | WM/WN | Addition of Distributed TCAP information |
| 6.10.0B1 | 09.02.08 | NW/DSL | Addition of Sigtran M3UA |
| 6.10.1 | 12.09.08 | WM | Clarified continuity_check_ind field.<br>Removed hyperlinks to cross-referenced documents. |
| 6.10.2 | 14.10.08 | NW | Addition of SCCP routing information. |
| 6.10.3 | 30.10.08 | NW | Updated after review. |
| 6.11.0 | 14.09.10 | DSL | Fonts changed to Arial |
| 6.11.2 | 18.01.11 | DSL | Minor corrections |
| 6.11.11 | 06.10.11 | DSL | Minor corrections |
| 6.12.2 | 05.07.13 | DSL | IPv6 support. Additional ISUP information. |
| 6.13.0 | 27.10.14 | DSL | Minor corrections |
| 6.14.0 | 15.09.16 | DSL | Minor corrections |
| 6.15.1 | 31.08.18 | DSL | Add M2PA |
| 6.16.0 | 05.05.22 | DSL | Update page footers |
| 6.16.1 | 13.02.23 | DSL | Update title page. |

CONTENTS


1 Introduction ..... 6
2 Getting started: SS7 protocols and networks ..... 7
2.1 Protocol layers ..... 7
2.1.1 Message Transfer Part (MTP) ..... 8
2.1.2 Sigtran M3UA ..... 8
2.1.3 Sigtran M2PA ..... 8
2.1.4 User Parts ..... 8
2.2 Message routing and addressing ..... 9
2.2.1 Signalling Point Codes ..... 9
2.2.2 Service Information Octet ..... 9
2.2.3 Routing labels ..... 9
2.2.4 ISUP Circuit Identification Codes ..... 10
2.2.5 SCCP subsystems ..... 10
2.2.6 SCCP Global Titles ..... 10
2.3 Signalling components ..... 12
2.3.1 Signalling End Points ..... 12
2.3.2 Signalling Transfer Points ..... 12
2.3.3 Combined Transfer and End Points ..... 12
2.3.4 Signalling links and link sets ..... 12
2.3.5 Signalling relations ..... 13
2.3.6 Signalling routes and route sets ..... 13
2.4 Intelligent Network components ..... 14
2.4.1 Service Switching Point ..... 14
2.4.2 Service Control Point ..... 14
2.4.3 Intelligent Peripherals ..... 15
2.4.4 Other Intelligent Network components ..... 15
2.5 Signalling modes ..... 15
2.5.1 Fully associated signalling ..... 15
2.5.2 Quasi associated signalling ..... 16
3 Aculab SS7 stack architecture ..... 17
3.1 SS7 card firmwares ..... 17
3.2 SS7 kernel driver ..... 18
3.2.1 MTP Level 3 ..... 18
3.2.2 Sigtran M3UA ..... 18
3.2.3 Sigtran M2PA ..... 18
3.2.4 SCCP User Part ..... 18
3.2.5 TCAP stub ..... 18
3.2.6 ISDN User Part (ISUP) ..... 18
3.2.7 Other User Parts ..... 18
3.3 MTP testing User Part ..... 18
3.4 Call Control Driver and ISUP API ..... 19
3.5 TCAP API library ..... 19
3.6 SCCP API library ..... 19
4 LAN distribution of Aculab SS7 ..... 20
4.1 Distributed ISUP and TCAP applications ..... 20
4.2 Dual-redundant MTP ..... 21
4.3 SS7 with Prosody X ..... 22
5 Using Aculab MTP ..... 23
6 Using Aculab Sigtran M3UA ..... 24
6.1 Application Servers and Signalling Gateways ..... 24
6.2 Peer-to-peer Nodes ..... 25
6.3 Routing Keys and Routing Contexts ..... 26
6.4 Traffic Modes ..... 26
7 Using Aculab Sigtran M2PA ..... 27
8 Using Aculab ISUP ..... 28

**8.1 ISUP API ..... 28**
8.1.1 Mapping between Aculab API and ISUP protocol parameters ..... 28
**8.2 ISUP helper library ..... 35**
8.2.1 isup_get_next_parameter() ..... 35
8.2.2 isup_get_parameter() ..... 35
8.2.3 isup_get_pcompat_info() ..... 36
8.2.4 isup_lib_version() ..... 36
8.2.5 isup_set_parameter() ..... 36
8.2.6 Error codes ..... 37
**8.3 ISUP feature information ..... 38**
8.3.1 Diversion and forwarding information ..... 38
8.3.2 User to user information ..... 39
8.3.3 Raw parameters and messages ..... 40
**8.4 Transmission path switching ..... 41**
8.4.1 Originating exchanges ..... 41
8.4.2 Intermediate exchanges ..... 41
8.4.3 Destination exchanges ..... 41
8.4.4 Tones and announcements - general ..... 42
8.4.5 Ring tone ..... 42
**8.5 ISUP Continuity test ..... 43**
8.5.1 Inward continuity check (IAM) ..... 43
8.5.2 Inward continuity check test call (CCR) ..... 44
8.5.3 Outbound continuity test (IAM) ..... 44
8.5.4 Outbound continuity check test call (CCR) ..... 44
**8.6 System performance considerations ..... 45**
8.6.1 Maximising throughput ..... 45
8.6.2 Avoiding missed calls ..... 45
**8.7 Using Aculab ISUP with national variants ..... 46**
**9 Using Aculab SCCP ..... 47**
**9.1 SCCP Addressing ..... 47**
9.1.1 Routing on SSN ..... 47
9.1.2 Routing on Global Title ..... 47
**9.2 SCCP status information ..... 47**
**10 Using Aculab TCAP ..... 48**
**10.1 TCAP protocol and procedures ..... 48**
10.1.1 TCAP messages ..... 48
10.1.2 TCAP components and operations ..... 48
10.1.3 TCAP dialogues and transactions ..... 48
10.1.4 Unstructured Dialogue ..... 49
10.1.5 Structured dialogue ..... 49
10.1.6 Dialogue identifiers ..... 49
10.1.7 Transaction identifiers ..... 49
**10.2 Writing Aculab TCAP applications ..... 50**
10.2.1 Configuration parameters ..... 50
10.2.2 ASN.1 API parameters ..... 50
10.2.3 SCCP Service Access Points ..... 51
10.2.4 TCAP transaction structures ..... 51
10.2.5 Building messages ..... 51
10.2.6 Sending messages ..... 51
10.2.7 Receiving messages ..... 52
10.2.8 Decoding messages ..... 52
10.2.9 Message event notification ..... 52
10.2.10 Encoding/Decoding ASN.1 ..... 53
**11 Sample applications ..... 55**
**11.1 What the samples do ..... 55**
11.1.1 Call generator ..... 55
11.1.2 “SSP” Call handler ..... 55
11.1.3 “SCP” Call validation ..... 55
**11.2 The sample network ..... 58**

**11.3 Installation and configuration ..... 59**
11.3.1 Hardware and software installation ..... 59
11.3.2 Verifying network connectivity ..... 59
**11.4 Running the samples ..... 59**
11.4.1 Compiling and linking ..... 59
11.4.2 Running the SSP ..... 60
11.4.3 Running the SCP ..... 60
11.4.4 Generating calls ..... 60
**Appendix A : Code examples ..... 62**
**A.1 Example 1 – Flexible ISUP parameter edit on a supported message type ..... 62**
**A.2 Example 2 – Supporting a nationally significant message ..... 63**
**A.3 Example 3 – Enabling, receiving, and handling of EV_EXT_RAW_MSG ..... 64**
A.3.1 Enabling EV_EXT_RAW_MSG ..... 64
A.3.2 Receiving EV_EXT_RAW_MSG and processing raw messages ..... 64
**A.4 Example signalling trace ..... 66**

## Table of Figures

Figure 1 - SS7 and the OSI model ..... 7
Figure 2 - Signalling End Points and Signalling Transfer Points ..... 12
Figure 3 - Intelligent Network nodes ..... 14
Figure 4 - fully associated signalling between ISUP exchanges ..... 15
Figure 5 - Quasi associated signalling between ISUP exchanges ..... 16
Figure 6 - Simplified view of Aculab SS7 internal architecture ..... 17
Figure 7 - Distributed ISUP and Distributed TCAP architecture ..... 20
Figure 8 - Distributed ISUP and Distributed TCAP with dual redundant MTP ..... 21
Figure 9 - Distributed ISUP using Prosody X and dual redundant MTP ..... 22
Figure 10 - M3UA application server and signalling gateway ..... 24
Figure 11 - M3UA peer-to-peer ..... 25
Figure 12 - Unsuccessful call attempt ..... 56
Figure 13 - Successful call attempt ..... 57
Figure 14 - Network diagram for sample applications ..... 58
Figure 15 - Single-chassis implementation of the sample network using a 4-port PM module and two crossover cables (simpler networks could be tested using just two ports and a single cable) ..... 58

## List of tables

Table 1: ISUP messages sent in response to generic call control API calls ..... 29
Table 2: Generic call control API events raised on receipt of common ISUP protocol messages (not including “Extended” API events) ..... 29
Table 3: Generic call control API parameters vs. ISUP protocol parameters (the API parameters may appear in more than one API structure) ..... 30
Table 4: Error codes ..... 37

# 1 Introduction

This manual provides guidance for software developers using the Aculab SS7 products. It provides an overview of SS7 protocols, networks, national and international variants, and explains the ways in which the services provided by SS7 protocols are made available by the Aculab software interfaces.

Developers using Aculab SS7 will be required to program to various Application Program Interfaces (APIs). There are two main APIs for SS7, namely ISUP and TCAP, either or both of which may be used in a customer application. The terms “ISUP” and “TCAP” will be explained in detail in later sections of this manual, however, ISUP is essentially used for setup of circuit switched calls (e.g. voice calls), whilst TCAP is used for transaction based applications such as mobile text messaging.

For ISUP, the API used is Aculab’s generic ***Call control API guide***, the same API is used for other call control protocols provided by Aculab, such as ISDN or CAS. The TCAP API is described in a separate manual, ***Distributed TCAP API guide***.

Applications using ISUP for call setup will need to use the Aculab switch API, see the Aculab ***Switch API guide***, which is used to connect local circuitry or devices to the network on TDM timeslots controlled by the ISUP call control API.

During application development, developers may find that they have a need to understand how to install and configure Aculab SS7. This along with other aspects of system administration is covered in the ***SS7 Installation and administration guide***. As information that exists in other documents has, in general, not been replicated in this document, you may need to read the SS7 Installation and administration guide as well as the various Aculab API guides and associated documentation.

A sample application, complete with network configuration, is described in section 8, and is available to Aculab customers as downloadable source code. It illustrates the use of the various APIs and demonstrates how a single stand-alone computer can be used for the development and testing of applications.

Further sample code fragments are provided in Appendix A. These may be useful to programmers who need to develop applications that use some of the more advanced features of the ISUP or TCAP APIs.

Aculab also offer a comprehensive list of customer training courses, which should assist those wishing to further expand their knowledge of the topics covered in this manual. Please contact the Aculab Training Manager for details (email training@aculab.com).

# 2 Getting started: SS7 protocols and networks

This section provides a brief introduction to SS7 technology; it is not Aculab-specific. If you are already familiar with SS7, you may be able to skip over most of the detail. In subsequent sections, explanations will be provided showing how Aculab have implemented the SS7 protocols.

A full and detailed description of all aspects of SS7 protocols would be outside the scope of this document. If a more detailed description and understanding is needed, please refer to the ITU-T recommendations (Q.700 series). In the event of any ambiguities or contradictions in this document, the ITU-T recommendations should be regarded as the authoritative reference.

The following section is based on ITU-T international recommendations. Should you have an interest in a national variant of SS7, you should refer to appropriate national publications. National variants of SS7 are usually described in the form of a protocol specification based on the ITU-T recommendations.

## 2.1 Protocol layers

SS7 is a layered protocol conforming loosely to the widely recognised OSI seven-layer model. The protocol layers are known in SS7 as “levels”, and are divided between the Message Transfer Part levels 1 to 3, which correspond to OSI layers 1 to 3, and the user parts, which correspond to OSI layers 4 to 7. Figure 1 shows the layered approach to SS7, and how it maps to the OSI model.

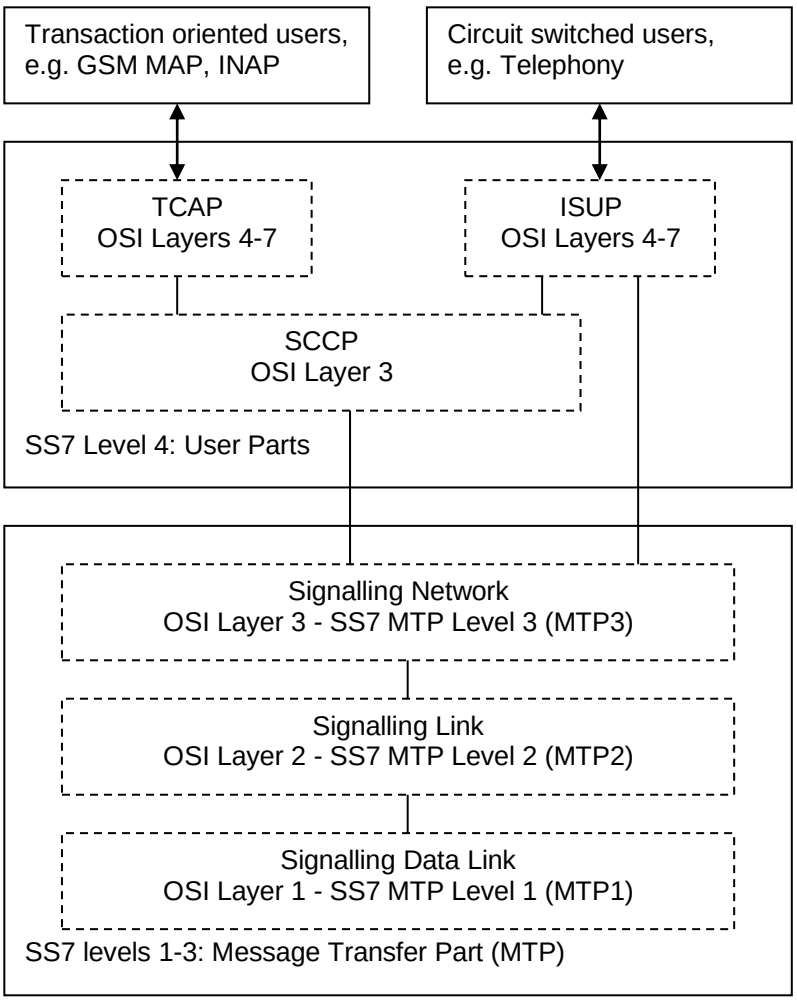


**Figure 1 - SS7 and the OSI model**

### 2.1.1 Message Transfer Part (MTP)

MTP is described in ITU-T recommendations Q.70x (currently, Q.700-Q.707). It provides the signalling functions equivalent to layers 1 to 3 of the OSI seven layer model.

#### 2.1.1.1 MTP level 1 – signalling data link

Level 1 defines the physical medium in terms of electrical and functional characteristics. A typical SS7 data link is a 64K bits per second timeslot within an E1/T1 TDM, although other physical mediums for Level 1 are also possible.

#### 2.1.1.2 MTP level 2 – signalling link

Level 2 defines the procedures for transfer of signalling messages over a level 1 data link. Level 2 provides detection and recovery of errors that may occur at level 1, and hence it provides a reliable link for transfer of messages.

#### 2.1.1.3 MTP level 3 – signalling network

Level 3 provides functions for sequenced transport of messages between individual signalling links and user parts. This includes a network routing (or "transfer") function for forwarding traffic received from a signalling link back into the network on a different link. It also provides transmission and reception of messages between local user parts and signalling links.

One of level 3's responsibilities is to maintain routing tables so that when a message needs to be transmitted, it can choose an appropriate signalling link that provides a route towards the destination.

### 2.1.2 Sigtran M3UA

M3UA is defined in the IETF specification RFC 4666. It carries the MTP3 service interface (ie that between MTP3 and SCCP or ISUP) over the IP network using Stream Control Transmission Protocol (SCTP).

The normal use of M3UA is to connect high throughput TCAP/SCCP applications to the SS7 network (of some telco). It can be used for ISUP but that is less common.

M3UA also supports a 'loopback' (IPSP) mode that directly connects two userparts without going through MTP3.

The Aculab M3UA supports the server (ie MTP3 interface) end of the connection, but it has restrictions that mean it is only really suitable for testing client (ie SCCP) applications.

**Note** **Since M3UA implements the MTP3 service interface it cannot be used to replace an MTP2 signalling linkset to interconnect two MTP3 systems.**

### 2.1.3 Sigtran M2PA

M2PA is defined in the IETF specification RFC 4165. It provides the equivalent of MTP2 signalling links over the IP network using Stream Control Transmission Protocol (SCTP).

### 2.1.4 User Parts

#### 2.1.4.1 ISDN User Part (ISUP)

ISUP is described in ITU-T recommendations Q.76x (currently, Q.760-Q.767), with further details on supplementary services in ITU-T Q.73x (currently, Q.730-737). ISUP provides the signalling functions for call setup and establishment of circuit-related services, including both data and voice circuits. Although the "ISDN" part of the acronym may imply otherwise, ISUP is used not only for ISDN calls, but also for analogue and POTS (Plain Old Telephone Service) calls.

In almost all cases, ISUP traffic is sent directly via MTP3. In rare circumstances, it is possible for ISUP traffic to be sent via SCCP as indicated in figure 1, but this is very unusual and has not been implemented by Aculab. Historically, ISUP superseded a number of other SS7 user parts including Telephony User Part (TUP), Data User Part (DUP), and various national user parts (NUPs).

#### 2.1.4.2 Signalling Connection Control Part (SCCP)

SCCP is described in ITU-T recommendations Q.71x (currently, Q.710-Q.716). It provides additional networking functionality over and above that provided by MTP3 or M3UA. This includes a connectionless service and a reliable connection oriented service.

A primary user of SCCP is TCAP (see section 2.1.4.3), which requires only the connectionless service. SCCP may also be encountered in other applications, such as some internal interfaces within a GSM network. As mentioned elsewhere, it is also possible for ISUP traffic to be sent over SCCP, but this mode is rarely encountered.

#### 2.1.4.3 Transaction Capabilities Application Part (TCAP)

TCAP is described in ITU-T recommendations Q.77x (currently, Q.771-Q.775). It is a transaction-oriented protocol and provides communications for non circuit-related services. TCAP applications include intelligent networking services such as number translation and mobile networking, for example, GSM MAP where it is used for management of cell handover, roaming, and delivery of text messages.

#### 2.1.4.4 Other User Parts

The list of user parts described in the above sections of the document is not exhaustive. SS7 also includes other user parts, some of which are effectively obsolete (see section 2.1.4.1). Aculab provides support for TCAP/SCCP and ISUP, which are the most commonly used. Other protocols may be supported through the SCCP and MTP3 API. Please contact Aculab support for further details.

## 2.2 Message routing and addressing

Some messages in an SS7 network, specifically those that are entirely handled within MTP2, relate implicitly to the adjacent protocol entity, i.e. the MTP2 entity at the other end of the signalling link. Such messages do not require any explicit addressing. Higher protocol layers may need to communicate with signalling points other than those which are immediately adjacent, and hence, explicit addressing is required based on signalling point codes and a service information octet, and any additional SCCP addressing as described below.

### 2.2.1 Signalling Point Codes

Each Signalling Point (SP) in an SS7 network is identified by a signalling point code, which is unique within the scope of the network. The number of bits in a point code varies between networks. The ITU-T recommendation for the international network is that point codes should be 14 bits long and this has been adopted in most national networks, but there are exceptions such as the USA and China, where 24 bit point codes are used.

### 2.2.2 Service Information Octet

When a message needs to be processed by a protocol layer above MTP2, the initial level of addressing is the Service Information Octet (SIO). The SIO is comprised in turn of two fields known as the Service Indicator (SI) and Network Indicator (NI). The SI identifies individual user parts, such as ISUP and SCCP. The NI specifies whether the User Part is international or national. For example, a Signalling Point could provide two different ISUP variants, one for the international network addressed by the international SI, and the other for local national traffic addressed by the national SI.

### 2.2.3 Routing labels

Messages that are to be handled by MTP3, or by the user parts, include a field called the “routing label”. The routing label includes an Originating Point Code (OPC) and a Destination Point Code (DPC), these are the sender’s and recipient’s signalling point codes.

The routing label may also contain a field known as the Signalling Link Selector (SLS). MTP3 includes procedures to ensure that all messages with the same SLS are delivered in the sequence in which they are sent. This usually requires that the messages be sent in sequence on the same signalling link. The user parts provide the SLS parameter, which enables them to request in-sequence delivery of streams of messages. Where appropriate, a User Part can allow MTP3 to load share traffic by assigning different SLS values to different messages.

An example of sequence delivery applicability would be the dialled digits that accompany a telephone call. It is clearly important that for each individual call the digits are delivered in the sequence they are sent. In fact, the ISUP protocol detail requires that all messages (not just dialled digits) for the same circuit (not just the same call) must be delivered in the sequence they are sent. This is achieved by using the same SLS for all messages relating to a given ISUP circuit. Different ISUP circuits can, however, use different SLS values, thus allowing

ISUP traffic to be load shared between available links.

Some MTP3 messages, relating to management of signalling links, contain a Signalling Link Code (SLC) instead of (and occupying the same position in the message as) the SLS. The SLC allows individual links in a link set, (see section 2.3.4) to be explicitly addressed. The protocol for messages that use an SLC is designed such that sequenced delivery is not a requirement. In some national variants, the SLC is a separate field.

### 2.2.4 ISUP Circuit Identification Codes

Within a Signalling Point that contains an ISUP User Part, there will be a number of different circuits, for example, individual timeslots within E1/T1 TDM trunks upon which calls may be established. To distinguish between these different circuits, all ISUP messages contain a Circuit Identification Code (CIC), which identifies the circuit to which the message refers.

Where ISUP is used for call setup over E1/T1 TDM circuits, there is a direct mapping between CICs and timeslots. This CIC numbers and the mapping to trunks and timeslots is usually obtained from the network provider. With the Aculab SS7 the CIC numbers are specified when the 'firmware' for a trunk is downloaded.

### 2.2.5 SCCP subsystems

Within a Signalling Point that contains an SCCP User Part, there may be several SCCP users. Each user is known as a “Sub System” and is assigned a Sub System Number (SSN) that is unique within the signalling point. Within a network, an SCCP user can therefore be uniquely addressed by its signalling point code and SSN. These can be further qualified by the NI to indicate whether the network is a local national network, or if it is the international network.

The following are examples of some of the SSNs assigned by ITU-T:

`SSN 0x0`=SSN not known/not used
`SSN 0x1`=SCCP Management
`SSN 0x2`=Reserved
`SSN 0x3`=ISUP
`SSN 0x4`=OMAP (Operations, Maintenance and Administration Part)
`SSN 0x5`=MAP (Mobile Application Part)
`SSN 0x6`=HLR (Home Location Register)
`SSN 0x7`=VLR (Visitor Location Register)
`SSN 0x9`=EIC (Equipment Identification Centre)
`SSN 0xA`=AUC (Authentication Centre)

Network-specific SSNs may also be used. The ITU-T recommendation is for these to be allocated descending values from `0xfe` downwards.

### 2.2.6 SCCP Global Titles

It is not always convenient to address SCCP users explicitly by point code and SSN, as this would require each and every Signalling Point in the network to have detailed knowledge of the point codes and SSNs of every other signalling point, together with a mapping between SSNs and the various services to which they correspond. In the case of international traffic, it would be very unlikely that each of the networks involved would be aware of individual point code allocations within one another’s networks. These problems are resolved by a more abstracted level of SCCP user addressing known as “Global Titles”, which removes the need to know the point code and SSN of the remote SCCP user.

An SCCP Global Title serves to uniquely identify an SCCP user or service, but does not require distant users of that service to know the actual point code and SSN of the network node where it is implemented.

As an SCCP message traverses the SS7 network each node need only determine the 'next hop' pointcode, this may be based on network availability, network or application loadshare or just delegating the actual routing to another system. The Global title may be rewritten as the message is forwarded, typically to add/remove a suffix or convert a 'service' GT into that of a specific instance of an application.

The first backwards message always contains the GT of the actual application that responded.

This is used for further messages (rather than the original 'service' GT) so that messages get routed to the same copy of the application.

Before delivery to an SCCP application, messages addressed by Global Title undergo a translation process known as Global Title Translation, which converts the Global Title into an explicit point code and SSN. The advantage of this approach is that it is only the translating node that needs to know the actual point codes and SSNs of the SCCP applications, and so only the translating node needs to keep track of their availability within the network.

Global titles are very much like ordinary telephone numbers, at times it is necessary to add (or remove) national prefixes, or convert (the equivalent of) 0800 numbers into geographic numbers.

## 2.3 Signalling components

All signalling points within an SS7 network contain MTP levels 1-3. Each individual signalling point may then function as a Signalling End Point and/or a Signalling Transfer Point, as shown in figure 2 and then described below.

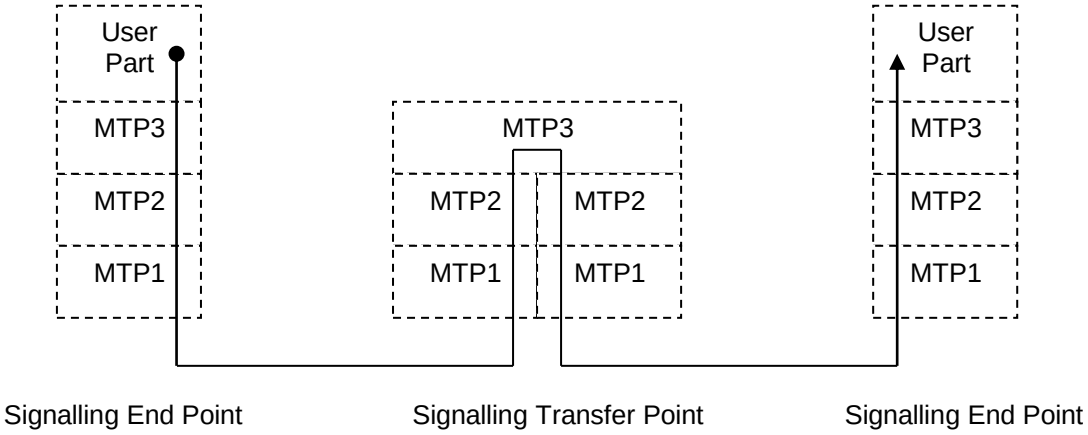


**Figure 2 - Signalling End Points and Signalling Transfer Points**

### 2.3.1 Signalling End Points

Some (not all) signalling points may contain one or more user parts capable of sending or receiving User Part network traffic. For example, an ISUP telephone exchange would include the ISUP User Part. Signalling points that include user parts are called either SEPs (Signalling End Points), or sometimes just SPs (Signalling Points).

### 2.3.2 Signalling Transfer Points

When a Signalling Point receives a message that needs to be delivered to MTP3 or a User Part, it may find that the destination point code in the routing label is not its own point code. Under these circumstances some signalling points have the ability to retransmit the message back out on another link that provides a route towards the desired destination. This is known as the “transfer” function, and such signalling points are known as Signalling Transfer Points (STPs).

STPs share some of the characteristics of SCCP nodes that perform Global Title Translation (see section 2.2.6), in that they both forward messages on towards their final destination. Global Title Translation is an SCCP User Part function, so strictly speaking it is not performed by a node that contains only an STP. However, it is not unusual for a Signalling Point that serves as an STP for messages addressed by point code (or point code and SSN), to also include the SCCP Global Title Translation function in which case it is often loosely referred to as an STP.

### 2.3.3 Combined Transfer and End Points

In some networks it is common to find a Signalling Point that is not only capable of the MTP3 transfer function, but also includes one or more user parts. For example, a network node may be capable of terminating ISUP traffic for its own signalling point code whilst acting as an STP for traffic addressed to other signalling points. Such a Signalling Point includes STP and SEP functionality, and is sometimes referred to as a Signalling Transfer End Point (STEP).

### 2.3.4 Signalling links and link sets

As described in section 2.1.1, signalling links are the medium by which two interconnected signalling points can exchange messages. There may be more than one parallel link connecting the same two signalling points, in which case the entire set of parallel links are known as a link set.

Each signalling link is assigned a Signalling Link Code (SLC) that uniquely distinguishes it from the other links in the same link set. The ITU-T recommendation for the international network is that an SLC should be four bits in length. Four bit SLCs have also been adopted by all major national specifications, so the number of links in a link set is normally restricted to 16.

### 2.3.5 Signalling relations

A signalling relation is the logical association between a User Part within a signalling point, and a remote user part in some other signalling point with which it is able to exchange messages.

A signalling relation may or may not correspond to a direct physical connection. Where the two user parts are interconnected via an STP, there is no direct signalling connection. If however the two user parts happen to be ISUP, then the switched voice/bearer circuits will form a direct physical connection that corresponds to the signalling relation. See also section 2.5, which provides more details about signalling via STPs.

### 2.3.6 Signalling routes and route sets

A signalling route is a predetermined path that a message takes from the originating User Part, through any number of STPs to the Signalling Point that contains destination User Part. For any given signalling relation there may be several alternative routes, in which case the signalling route set is the set of all routes that can be used for the relation. Individual routes within a route set can be assigned different priorities, allowing some to be used in preference to others, depending upon availability.

From the perspective of a Signalling Point that needs to send (or transfer) a message towards a remote User Part, the MTP only has visibility of the next “hop” in the route; subsequent hops are selected in turn by each STP that the message passes through. Thus, for the sake of configuring routing rules for an individual signalling point, it is only the immediately adjacent signalling points that need to be considered.

There is no limit to the number of routes in a route set, but most networks are designed to restrict the number of routing alternatives at an SP to a small number (often 2), which provides adequate fault-tolerance in the event of failure.

## 2.4 Intelligent Network components

By combining the circuit-switched capabilities of ISUP with the transaction-oriented features of TCAP, it becomes possible to build “intelligence” into the network, allowing calls to be handled based on dynamic criteria, possibly involving interaction with the caller.

There are various standards and protocols for intelligent networking, including ETSI Intelligent Network Application Protocol (INAP), Bellcore Advanced Intelligent Network (AIN) and other proprietary protocols. Most intelligent networks are designed around similar discrete component types including Service Switching Points, Service Control Points, and Intelligent Peripherals as illustrated in figure 3 and then described below.

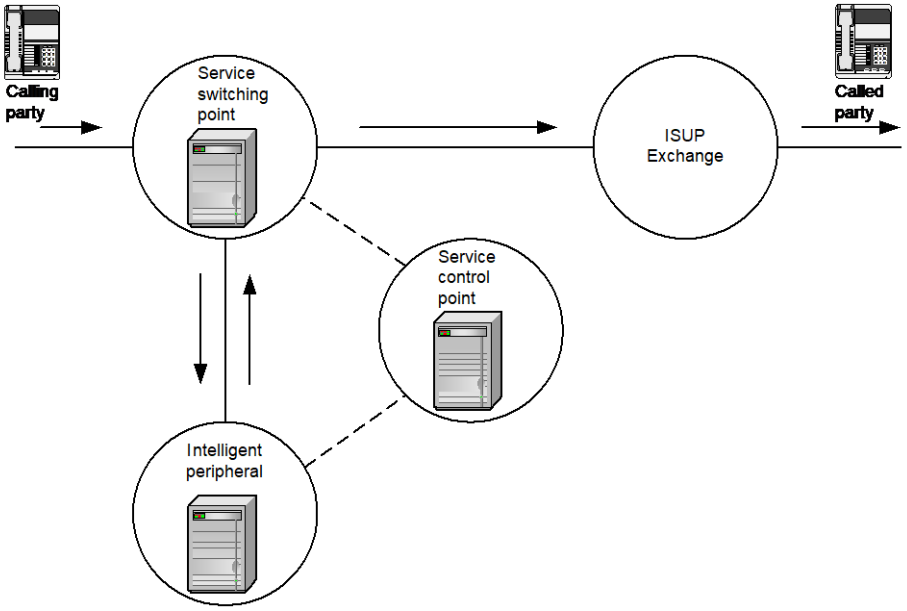


**Figure 3 - Intelligent Network nodes**

### 2.4.1 Service Switching Point

A Service Switching Point (SSP) is an ISUP exchange that includes the ability to make “intelligent” routing decisions, usually involving dialogue with a Service Control Point (see section 2.4.2). A common example is a call to an 800 freephone number, where the call passes through an SSP that in turn queries an SCP to translate the non-geographic freephone number into a geographic subscriber number towards which the call can be routed.

The term “SSP” is often used to describe any ISUP exchange that is capable of processing a call, but in this document “SSP” is used only in reference to exchanges with intelligent switching capability.

### 2.4.2 Service Control Point

A Service Control Point (SCP) may contain a database together with service control logic. Following the freephone example cited in section 2.4.1, the SCP would maintain a mapping of freephone numbers to actual subscriber numbers. An SCP may take account of other factors such as the caller’s geographic location or the load on individual call centres, but the SSP that queries a freephone number can simply “trust” on the SCP to make the right decision.

### 2.4.3 Intelligent Peripherals

In addition to SSPs and SCPs, an Intelligent Network would typically include Intelligent Peripherals that provide additional service functionality on-demand to SSPs or SCPs.

One common application of an Intelligent Peripheral is to provide a Specialised Resource Function (SRF) to interact by voice with the caller, possibly involving DTMF or speech recognition. In this scenario, the SCP would need to instruct the SRF of the actions it needs to perform as well as instructing the SSP to connect the speech paths. The SRF then responds to the SCP allowing it to make an appropriate decision about how the call should be handled.

Use of Intelligent Peripherals can offer cost savings for a network. For example, by using Intelligent Peripherals to perform DTMF or speech-recognition during call setup, the speech processing resource may be shared between many different calls as it is only used during the setup phase of each call. This contrasts with other, more traditional IVR (Interactive Voice Response) technologies that could require the speech resource to be assigned for the entire duration of the call.

### 2.4.4 Other Intelligent Network components

The components described so far are only a subset of the components likely to be encountered in a real network. Other component types may exist, such as Service Switching Control Points (SSCPs) that combine both SSP and SCP functions, and Service Data Points (SDPs) that provide a stand-alone data resource. A detailed description of all possible components is beyond the scope of this document.

## 2.5 Signalling modes

SS7 signalling involves delivery of a message from a User Part within a signalling end point to a peer User Part in some other signalling end point. The two end points may either be directly connected by a signalling data link, or they may be indirectly connected via one or more STPs. This distinction is illustrated in figures 4 and 5, and explained in more detail below.

### 2.5.1 Fully associated signalling

In fully associated mode, the traffic for a signalling relation is usually exchanged directly between the User Parts without passing through any STPs. In the case of ISUP, exchanges with directly interconnected voice/data circuits would have a direct signalling link set between each pair of exchanges.

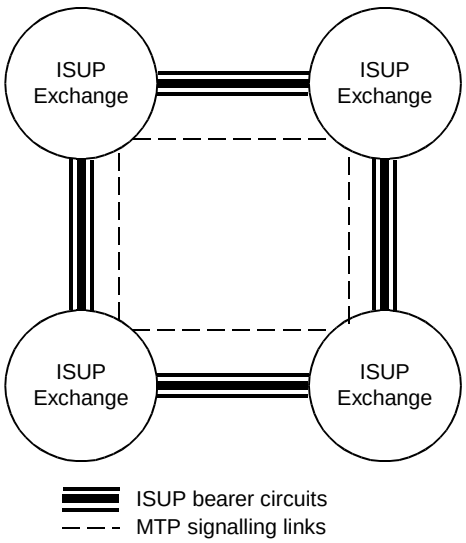


**Figure 4 - fully associated signalling between ISUP exchanges**

If the direct signalling link fails, an MTP3 route via another adjacent ISUP exchange (acting as an STP) would be used in order to maintain connectivity.

It is rare for TCAP/SCCP traffic to use fully associated signalling, but it is commonly used for ISUP in some regions (including the UK).

### 2.5.2 Quasi associated signalling

In quasi-associated mode, the traffic between the user parts passes through one or more STPs. In the case of ISUP exchanges with directly interconnected voice/data circuits to other exchanges, each exchange need only have a signalling link set to the adjacent STP, which can then route the signalling traffic (possibly via other STPs) towards the destination User Part.

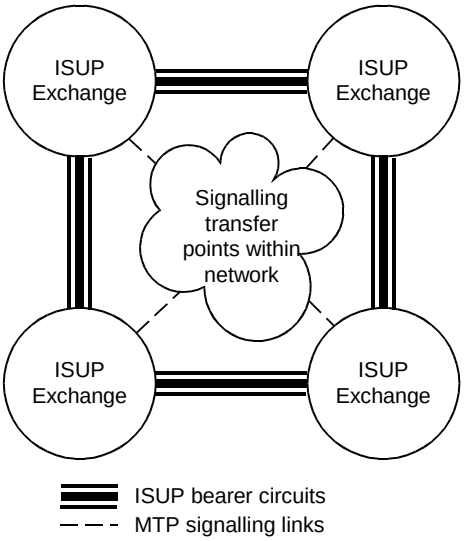


**Figure 5 - Quasi associated signalling between ISUP exchanges**

Quasi-associated signalling is widely used for TCAP/SCCP traffic, and is used for ISUP in some regions (including the USA).

# 3 Aculab SS7 stack architecture

The protocols and network features described in section 2 have been implemented by Aculab in such a way as to deliberately conceal most of the protocol detail from the user. This means that application developers can concentrate on the end-user services and applications.

In order to make best use of Aculab's SS7 it may help to have an appreciation of aspects of the implementation detail that is normally concealed by the APIs. This section explains the internal structure and rationale behind some of the major software components, and how they relate to the underlying operating system and processor environments, as well as to other Aculab components.

The major software components are illustrated in figure 6.

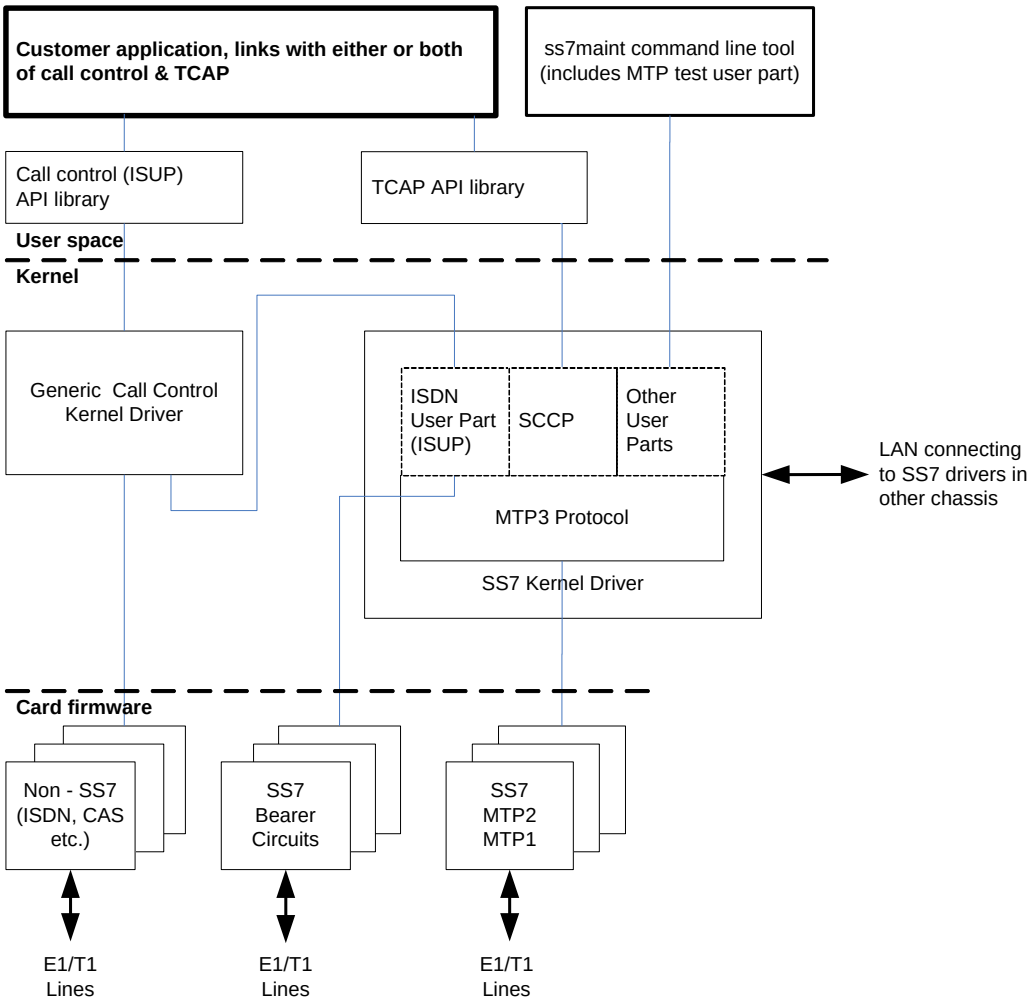


**Figure 6 - Simplified view of Aculab SS7 internal architecture**

This diagram is a simplified view and does not include all software components. The software architecture is subject to change as Aculab develops these products, and customers do not necessarily have any interface to the various software components that are shown.

## 3.1 SS7 card firmwares

The SS7 card firmware supports MTP1 data links and the physical layer protocol for voice/data timeslots within the E1/T1 TDM. Aculab SS7 allows signalling links and voice channels to share different timeslots within a common E1/T1, or to be assigned to separate E1/T1s if preferred.

SS7 MTP2 demands high volume traffic throughput (up to ~2600 messages per second per link) and very low (millisecond) latency. These place a high demand on the local processing hardware. However, most of the intense MTP2 traffic, such as Link Status Signal Units

(LSSU) and Fill In Signal Units (FISU), is handled entirely within MTP2 and never needs to be passed to MTP3.

To meet these demands MTP2 is implemented on its own processor and connects to the E1/T1 TDM links through the card's TDM switch. This reduces the load on both the card's general purpose CPU and the host system's CPU, while guaranteeing the required response times.

On the Prosody X V1 cards the MTP2 code runs on one of the DSPs (as would tone detect for CAS), on the V3 card it runs on soft CPUs on one of the FPGAs.

## 3.2 SS7 kernel driver

Most of the Aculab SS7 protocol stack runs on the application system in a large kernel driver. This makes it easy for multiple applications to use the services.

Any TCP and SCTP connections are made directly from the driver code (ie without using a daemon process). This significantly reduces the cost of sending and receiving data.

Aculab supplies an implementation of SCTP for windows that can only be used by SS7. On Linux the normal kernel SCTP implementation is used.

### 3.2.1 MTP Level 3

Handling of traffic by MTP3 in relation to any given MTP2 signalling link usually involves interaction with traffic relating to other signalling links. Put another way, MTP3 acts as a focal point for all links in a signalling point. For that reason, MTP3 it is not well-suited to being hosted on any one card "firmware", as that would localise it to the signalling traffic handled by the links on that card. Thus, in Aculab's SS7 MTP3 runs on the system host processor.

### 3.2.2 Sigtran M3UA

M3UA does not require any Aculab firmware. Instead, M3UA makes use of the host machine's network interfaces using SCTP. It is implemented in the kernel in order to interface to SCTP.

### 3.2.3 Sigtran M2PA

M2PA signalling links are implemented in kernel. They don't have the same latency requirements as MTP2 links, but a modern cpu can transfer very large numbers of messages.

### 3.2.4 SCCP User Part

The SCCP protocol is implemented in the kernel. This is necessary to route received messages to the correct user.

### 3.2.5 TCAP stub

Although most of the TCAP protocol is implemented in the userspace library, there is a small kernel component that parses receive messages so that they can be passed to the correct application.

### 3.2.6 ISDN User Part (ISUP)

The ISUP interface into MTP3 or M3UA passes through Aculab's generic call control driver and API. This allows the SS7 ISUP API to share common code-path with the company's other call control protocols (e.g. CAS, ISDN), and ensures consistency in API detail between ISUP and the other call control protocols.

### 3.2.7 Other User Parts

Traffic that is neither for ISUP nor TCAP may be accessible through additional Aculab libraries. One user of this interface is the MTP testing User Part as described in section 3.3.

## 3.3 MTP testing User Part

The Aculab "`ss7maint`" tool has been included in figure 1 to show its ability to provide one of the occasionally required user parts, namely the "MTP testing User Part" as described in ITU-T Q.782. In most situations, this User Part is used only during laboratory testing, so it need not be made a permanent part of a customer's application. Its inclusion in `ss7maint` relieves the customer's application of the need to implement that User Part.

`ss7maint` is a command-line tool with other functions in addition to that described above, including stack configuration, maintenance and debugging during application development as well as after deployment. An example protocol trace from `ss7maint` is provided in section A.4.

For a full description of `ss7maint`, please refer to the ***SS7 Installation and administration guide***.

## 3.4 Call Control Driver and ISUP API

This is the interface by which customers can implement call control applications using Aculab ISUP. It is exactly the same interface as is used for other Aculab call control protocols, such as ISDN and CAS. This provides a valuable simplification for many users, and especially for any application that may also have to run with other protocols. Application developers can usually regard ISUP as “just another protocol”.

## 3.5 TCAP API library

The TCAP API is implemented as a user-space library. The API is fully described in the Aculab ***Distributed TCAP API Guide***.

There is a small TCAP kernel component that allocates blocks of TCAP transaction id values to applications and routes received TCAP messages to the correct application. This allows multiple TCAP applications to use the same SSN.

The TCAP applications always connect to the SS7 driver using TCP. This means that the applications can be run on a different system.

## 3.6 SCCP API library

The SCCP API library allows direct access to both connectless and connection-oriented SCCP. The API is fully described in the ***SCCP API guide***.

The SCCP and TCAP API interfaces share a lot of common code.

# 4 LAN distribution of Aculab SS7

It can be advantageous to distribute the Aculab SS7 software, as well as user-written application software, across a Local Area Network (LAN) which can provide increased resilience and scalability. Aculab SS7 supports a number of different options for LAN distribution as described below.

## 4.1 Distributed ISUP and TCAP applications

For both ISUP and TCAP applications, it is possible for an application providing a large-scale service to be implemented using only a small number of signalling links, with all the links terminated in a single chassis. This can place considerable performance demands on the customer application, as well as the other software (including the Aculab components) running in the chassis. In the case of ISUP a large-scale application may imply a large number of cards, causing further problems relating to backplane capacity and power requirements. For both ISUP and TCAP applications such single-chassis systems are vulnerable, in the event of failure of that one chassis, to catastrophic failure of the entire system.

Distributed ISUP and TCAP overcome the problems listed above by spreading the load among a number of separate chassis. Each of the chassis operates independently, and failure of any one chassis does not affect the others, so in many cases low-cost chassis can be used which may also allow the possibility of overall cost saving.

When using distributed applications, there can be a number of different chassis, each running TCAP applications and/or ISUP applications, and an additional chassis (or two – see section 4.2) providing the signalling links. It is optionally possible to run ISUP or TCAP applications in the signalling chassis as well. The networking architecture for distributed ISUP is illustrated in figure 7.

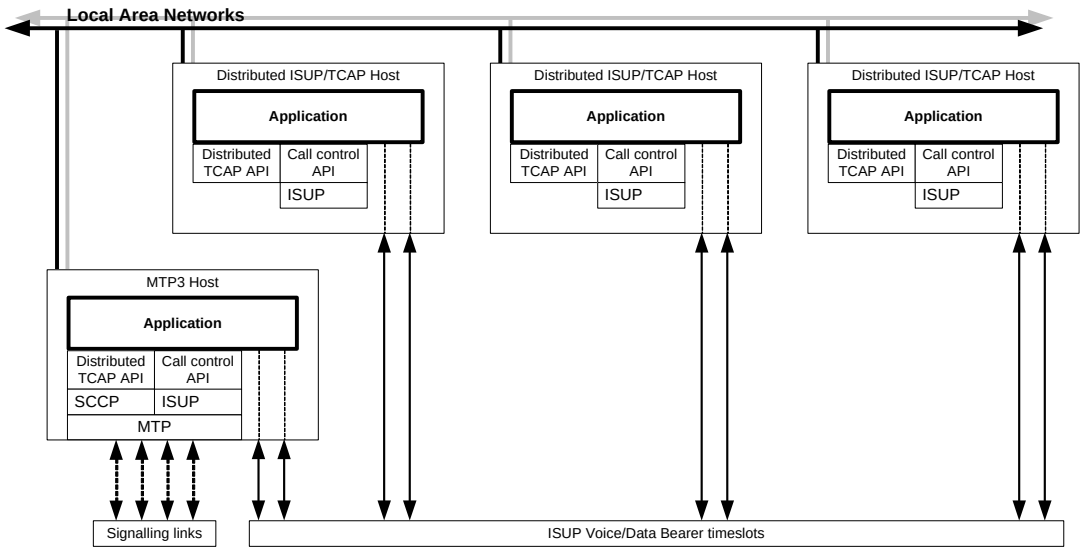


**Figure 7 - Distributed ISUP and Distributed TCAP architecture**

## 4.2 Dual-redundant MTP

Section 4.1 explained how TCAP and ISUP functionality can be distributed among a number of physically separate chassis, providing scalability as well as resilience for user's applications. When used as shown in Figure 7 however, the distributed applications still depend on the single MTP system that is shared by all application chassis. To overcome this potential weakness, it is also possible to add a second MTP host, thus eliminating any single point of failure.

The dual MTP configuration appears to the network as a single point code, with traffic load shared amongst all available links. In the event of failure of any one or more links, or failure of an entire MTP3 host chassis, the surviving links can continue to carry the traffic. The application hosts are unaffected by such failures. In the event that an ISUP chassis should fail, ISUP calls for circuits in other chassis are not disrupted, and this applies to calls in transient states as well as stable calls. The networking architecture for dual redundant MTP, together with distributed applications, is illustrated in figure 8.

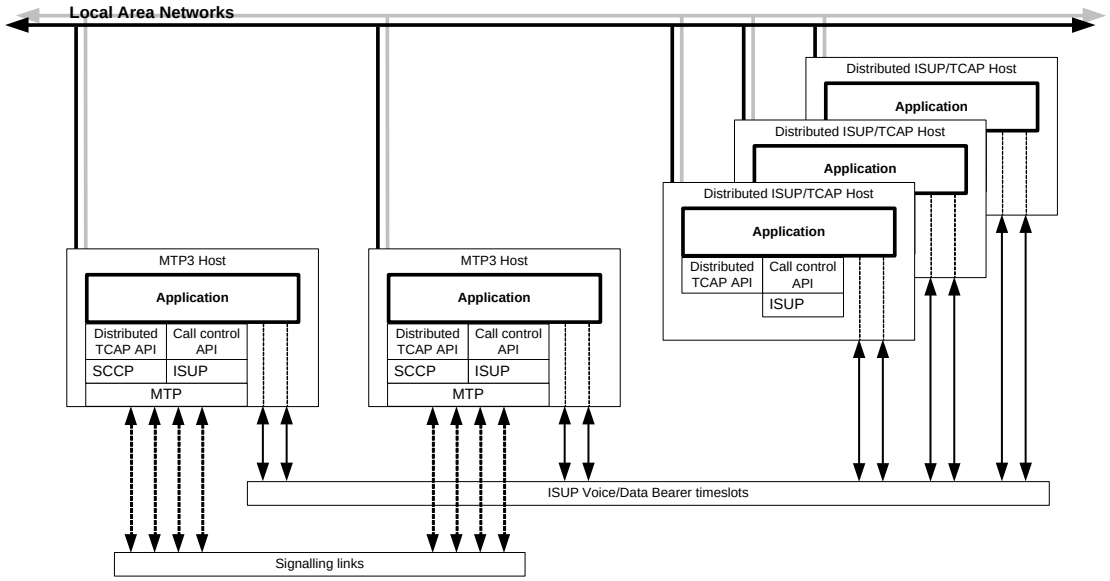


**Figure 8 - Distributed ISUP and Distributed TCAP with dual redundant MTP**

The link between the two MTP3 systems is implemented as a normal linkset typically with two 'signalling links' on two TCP connections - one established by each system. With two separate IP subnets this gives resilience against some failures of the local network infrastructure.

## 4.3 SS7 with Prosody X

The LAN distribution features already described can also be utilised in conjunction with Prosody X, in which case each Prosody X card can be configured as a stand-alone resource on the LAN and controlled by applications and protocol stacks running on one or more separate host chassis.

When using Prosody X for the provision of ISUP bearer circuits, the number of ISUP application hosts is independent of the number of remote Prosody X cards. At one extreme a separate host chassis could be used for each Prosody X card, and at the other extreme a single host chassis could control all ISUP circuits for a large number of Prosody X cards. The networking architecture is illustrated in figure 9.

Although not shown in figure 9, it is also possible for Prosody X cards carrying MTP signalling links to be remotely accessed via the LAN in exactly the same way as the Prosody X cards supporting ISUP voice/bearer circuits.

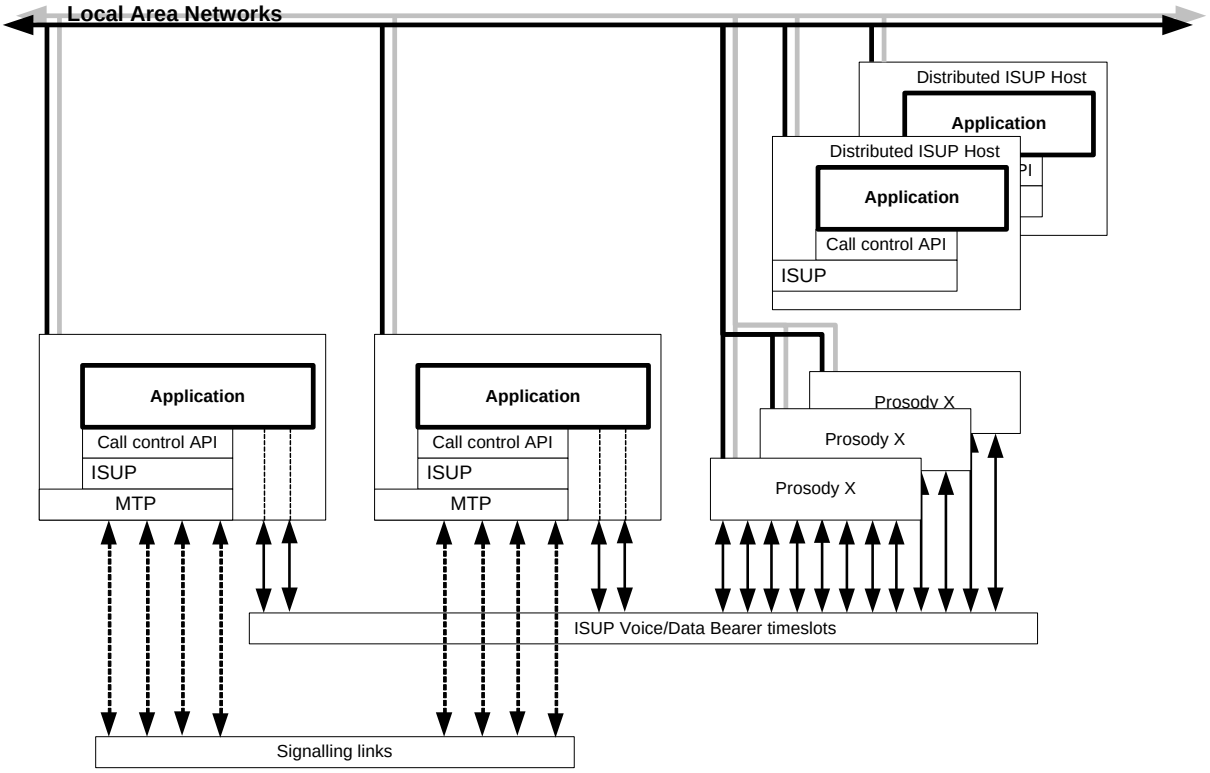


**Figure 9 - Distributed ISUP using Prosody X and dual redundant MTP**

# 5 Using Aculab MTP

The MTP has been implemented in such a way that for the majority of application programmers, whose chief concern is normally ISUP and/or TCAP, the MTP to a large extend can be disregarded.

Other applications that require a special MTP user part not supplied by Aculab may still be able to use the MTP3 through an MTP3 API. Please contact Aculab support for further details.

In either case, the MTP does have to be correctly configured with parameters that correspond to those of the peer MTP nodes to which it connects, and to specify the routing rules for delivery of User Part traffic via appropriate signalling links. For a full description of how to configure Aculab MTP, please refer to the ***SS7 Installation and administration guide***.

# 6 Using Aculab Sigtran M3UA

The IETF specify two methods of working for M3UA, application server / signalling gateway and peer-to-peer.

In an application server / signalling gateway configuration, MTP3 messages from a traditional SS7 signalling point are received in the gateway. At the gateway, the MTP3 messages are converted to M3UA messages and sent over SCTP/IP to the application server. The application server distributes the messages to the correct SCCP or ISUP user part. In other words the MTP3 service interface is extended over the IP network.

In contrast, the peer-to-peer method allows two applications to communicate with each other over M3UA without the need for any traditional SS7 network.

## 6.1 Application Servers and Signalling Gateways

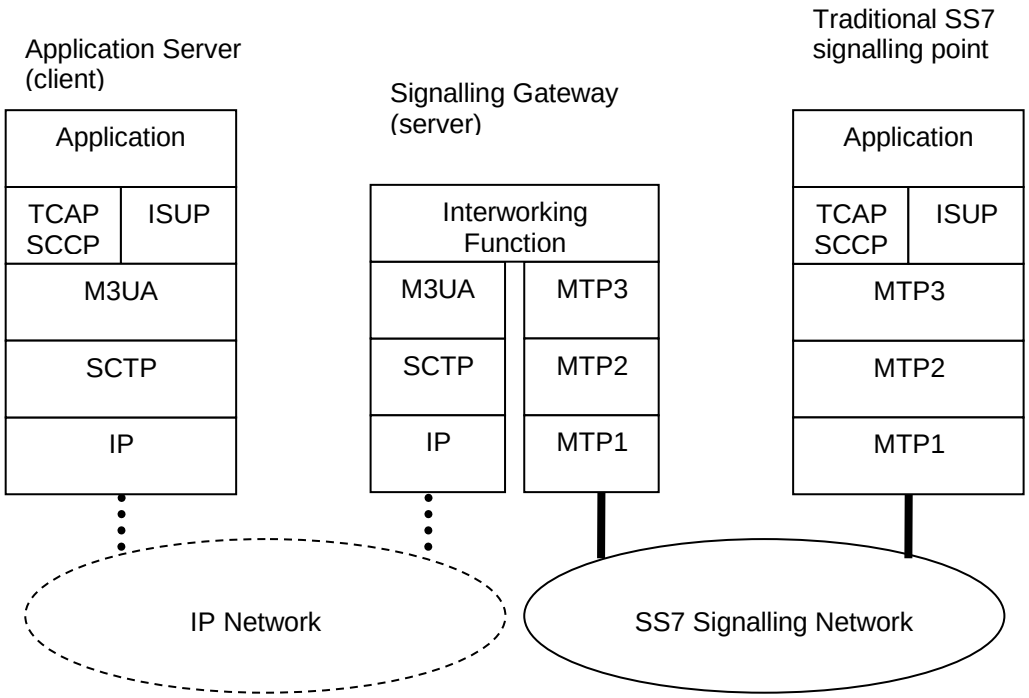


**Figure 10 - M3UA application server and signalling gateway**

To configure an application server, an [M3UA] section is required in the stack configuration file containing a [CLIENT] subsection using the '`host=hostname`' parameter to identify the IP address of the signalling gateway to connect to.

To have any effect, a minimum of one routing key is also required to identify a remote signalling point in the SS7 signalling network. See section 6.3.

To configure a signalling gateway, an [M3UA] section and an [MTP3] is required. The [M3UA] section should contain a [SERVER] subsection. Since the Aculab SS7 MTP only supports signalling end point functionality, the local point code of the signalling gateway must match the local point code of any application servers that wish to connect.

Unless dynamic creation of routing keys is configured (`key_management=dynamic`) at least one routing key must be defined. See section 6.3.

For a full description of how to configure application servers or signalling gateways, please refer to the ***SS7 Installation and administration guide***.

## 6.2 Peer-to-peer Nodes

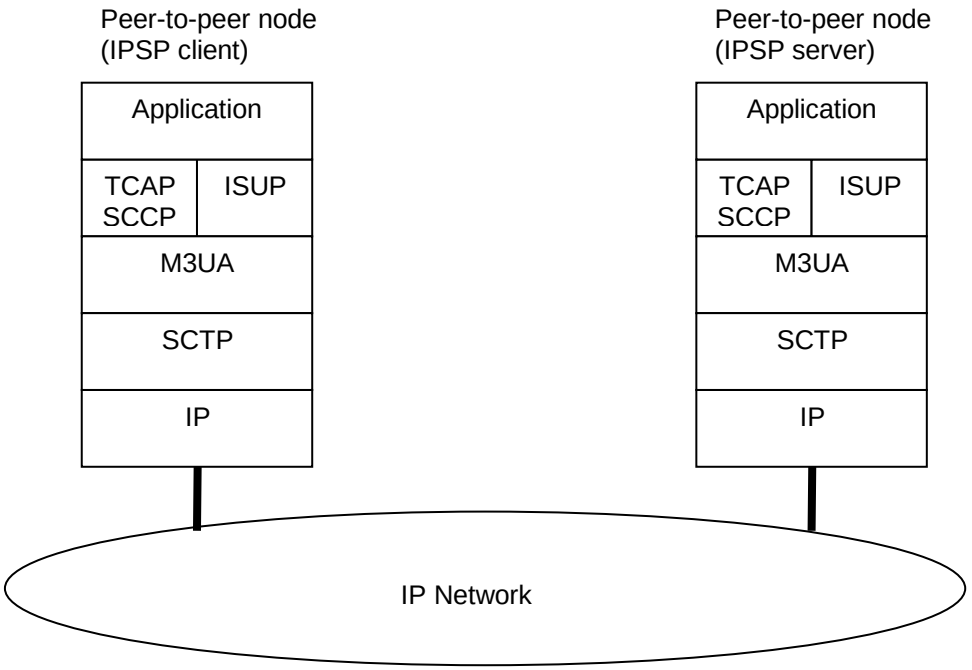


**Figure 11 - M3UA peer-to-peer**

In the peer-to-peer method of working, there is no need for a traditional SS7 signalling network. All communication between the SCCP or ISUP peers is done directly using M3UA over SCTP/IP.

Since there is no MTP3, messages can only be sent to the remote peer. M3UA contains no concept of a routing function. Messages can not be routed beyond the remote peer.

In the IETF specification, RFC 4666, a process that uses M3UA in peer-to-peer mode is defined as an IP Server Process (IPSP). IP Server Processes are defined within an [M3UA] section with either an [IPSP_CLIENT] or an [IPSP_SERVER] subsection.

IPSP clients always initiate M3UA associations to the remote peer, whereas IPSP servers expect the remote peer to begin the initiation. For a system to function correctly, an IPSP client must be configured at one peer node and an IPSP server at the other.

To configure an IPSP client the IP address of the remote host to connect to must be specified. This is configured using the '`host=`*`hostname`*' parameter. In addition, a routing key that identifies the point code of the remote peer must also be defined. See section 6.3.

For IPSP servers, routing keys are required unless dynamic creation of routing keys is configured (`key_management=dynamic`). See section 6.3.

For a full description of how to configure peer-to-peer nodes, please refer to the ***SS7 Installation and administration guide***.

## 6.3 Routing Keys and Routing Contexts

An M3UA server can distribute received traffic between its clients based on the SS7 point codes and service indicator (SI) of the received MTP3 routing label. A 'routing key' is a group of point code pairs and SI values that (in the terms of RFC 4666) identifies all the traffic for an Application Server. A 'routing context' is a 32bit unsigned integer that identifies a single routing key. Since the Aculab SS7 stack only supports a single local point code (per stack instance) the local point code is fixed.

An M3UA association (SCTP connection) can carry traffic for more than one routing key/context, and traffic for a single routing key/context can be sent over several M3UA associations. However an SS7 relation (both point codes + SI) can only occur in a single routing key.

If the routing key management procedures are not used then the same routing context should be configured at the client and server (i.e. they should have the same numeric value and identify the required traffic). If a client has only a single routing context configured, the Aculab M3UA code will (by default) wait for the first notify message from the server and use any routing context in it for further requests.

If the key management procedures are used then the routing keys configured at the client and server must exactly match (i.e. they must specify exactly the same point codes and SI values), but the routing contexts need not. The server can also be configured to dynamically create routing keys in response to requests from clients.

Typically a client will use local rules to determine which M3UA connection (or MTP3 link) is used for outbound data, whereas an M3UA server selects between its connected clients based on their registered routing keys (or those that are implicitly registered based on the client's IP address when the registration procedures aren't used).

The Aculab M3UA client implementation doesn't really have the notion of an 'Application server'. TCAP applications share a common SCCP using SCCP routing rules. So M3UA treats a routing key as a set of SS7 relations (point code pair + SI) that are processed (by M3UA) as a single entity, and a routing context as an integer handle for such a relation. TCAP applications need know nothing about these address groups – but, if required, may use the SS7 maintenance API to dynamically activate and deactivate individual routing contexts.

Routing keys are defined in the Aculab SS7 stack configuration file in a [ROUTING_KEY] subsection of the [M3UA] section. Each [ROUTING_KEY] section contains one or more [ADDRESS] subsections – each corresponds to a single grouping of destination address, service indicators and originating point code list within an M3UA routing key parameter to a registration request message. RFC 4666 marks the 'originating point code list' (i.e. the remote point codes) as optional, and allows a 'mask' to be applied to the point code. The Aculab M3UA code requires the originating point code list and does not support point code masks here (they are supported in DUNA/DAVA indications).

## 6.4 Traffic Modes

The three different traffic modes defined in the M3UA standard can be configured in the Aculab M3UA using the '`traffic_mode`' parameter. The three modes are:

- **Override** – indicates that the client or IPSP client requests all traffic. This traffic mode is used when there is only one client or when a number of clients are used in a working / standby configuration.
- **Loadshare** – indicates that this client or IPSP client wishes to share traffic with any other clients or IPSP clients. This traffic mode is used when the clients are configured to share the load. In the Aculab M3UA, loadsharing is done on the SLS value of the message.
- **Broadcast** – indicates that this client or IPSP client wishes to receive the same messages as any other client or IPSP client. Possible uses for this mode are hot-standby or monitoring of traffic.

All the users of single routing context must request the same traffic mode.

# 7 Using Aculab Sigtran M2PA

M2PA signalling links are configured by adding an [M2PA] subsection to the [MTP3] [DESTINATION] section of the ss7 configuration file. They are completely transparent to the user parts.

A linkset can contain multiple M2PA signalling links (or even a mixture of MTP2 and M2UA ones – but that would be silly!).

Multiple connections between the same pair of hosts normally require different port SCTP numbers. For tests between Aculab systems (eg loopback to 127.0.0.1) a password can be specified and then the same port number used for all the connections.

Because the socket interface uses the client-server model it isn't possible to have a socket repeatedly send out SCTP INIT chunks while ignoring the ABORT responses returned because there is no peer application. One end of the connection must be configured with `listen=y` to wait for the incoming SCTP connection.

# 8 Using Aculab ISUP

## 8.1 ISUP API

Aculab's ISUP API is exactly the same generic API used by other Aculab call control protocols described in the ***Call control API guide***. The information provided here supplements that API guide.

It is also necessary to ensure that ISUP is correctly configured with parameters that correspond to the peer ISUP to which it connects. For a full description of how to configure Aculab ISUP, refer to the ***SS7 Installation and administration guide***.

Some developers may need to have access to the individual raw protocol messages and parameters rather than using the Aculab generic abstraction. This includes those who are "specialised" in SS7 and gain no benefit from the generic API, and those who use the generic API but need to enhance it for a customised application or network feature. This is possible using the "Flexible ISUP API" extensions, which provides access to raw ISUP messages and parameters, and a library of "helper" functions for constructing these parameters.

Refer to the call control API for details of the flexible ISUP raw message and parameter interface. Further code examples can be found in Appendix A of this document.

### 8.1.1 Mapping between Aculab API and ISUP protocol parameters

One aim of the generic call control API is to abstract protocol-dependent signalling information so that the user can largely disregard the underlying protocol details. When an API function is called that requires an ISUP message to be sent, the SS7 driver will convert the information provided at the API into a suitable combination of ISUP messages and parameters. When an ISUP message is received, the SS7 driver converts it into suitable event and parameter notifications that are delivered via the API. The point to note is that, in each case, the driver converts between ISUP protocol and the generic API and hence the user does not need to understand the ISUP protocol details.

Tables 1 to 3 illustrate how some of the generic API functions and parameters may correspond to ISUP protocol messages. The conversion between API parameters and protocol equivalents is complex, depends upon state transitions within the drivers, and may change as Aculab develops the software so these tables should not be relied on under all circumstances. If an application needs explicit control over ISUP message and parameter content then it may be advisable to use the flexible ISUP API extensions (see section **Error! R eference source not found.**).

Message abbreviations used in tables 1 to 3:

| | |
|---|---|
| IAM | Initial Address Message |
| SAM | Subsequent Address Message |
| CPG | Call Progress |
| ACM | Address Complete Message |
| REL | Release Message |
| RLC | Release Complete Message |
| COT | Continuity Message |
| CCR | Continuity Check Request Message |
| RSC | Reset Circuit Message |
| GRS | Circuit Group Reset |
| BLO | Blocking Message |
| UBL | Unblocking Message |
| CGB | Circuit Group Blocking Message |
| CGU | Circuit Group Unblocking Message |

**Table 1: ISUP messages sent in response to generic call control API calls**

| Generic API Call | ISUP Protocol Message | Notes |
|---|---|---|
| call_openout | IAM | - |
| call_feature_openout | IAM | - |
| call_send_overlap | SAM | - |
| call_proceeding | ACM or CPG | 1 |
| call_incoming_ringing | ACM or CPG | 1 |
| call_progress | ACM or CPG | 1 |
| call_disconnect | REL or RLC | - |
| call_release | REL or RLC | - |
| call_accept | CON or ANM, or ACM followed by ANM | 2 3 |
| call_maint_ts_block | BLO | - |
| call_maint_ts_unblock | UBL | - |
| call_maint_port_block | CGB | - |
| call_maint_port_unblock | CGU | - |
| call_maint_port_reset | GRS or RSC | - |

### Table 1 notes:

1. ACM for the first backwards message, CPG for subsequent requests.
2. For ITU-T variants: ACM (if not already sent) followed by ANM. If the sending of CON is enabled in the stack configuration file CON will be sent instead of ACM and ANM.
3. For ANSI variants ANM is always sent.

**Table 2: Generic call control API events raised on receipt of common ISUP protocol messages (not including "Extended" API events)**

| ISUP Protocol Message | Possible API Events | Notes |
|---|---|---|
| IAM | EV_INCOMING_CALL_DET | 1 |
| ACM | EV_OUTGOING_PROCEEDING | - |
| SAM | EV_DETAILS | - |
| COT | EV_DETAILS | - |
| ANM | EV_CALL_CONNECTED | - |
| CON | EV_CALL_CONNECTED | - |
| REL | EV_REMOTE_DISCONNECT<br>EV_IDLE | - |
| RLC | EV_REMOTE_DISCONNECT<br>EV_IDLE | - |
| RSC | EV_REMOTE_DISCONNECT<br>EV_IDLE | 1 |
| GRS | EV_REMOTE_DISCONNECT<br>EV_IDLE | 1 |
| BLO | EV_REMOTE_DISCONNECT<br>EV_IDLE | 1 |

### Table 2 notes:

1. It may be valid for the remote ISUP to send this message for a circuit that is not associated with an Aculab call handle and in a suitable state for event notification, in which case the message may be processed and responded to without any event being generated.

**Table 3: Generic call control API parameters vs. ISUP protocol parameters (the API parameters may appear in more than one API structure)**

| API parameter name | Protocol parameter | Protocol messages | Notes |
|---|---|---|---|
| `service_octet`<br><br>`add_info_octet`<br><br>`bearer` | "Transmission Medium Requirement" parameter.<br><br>"User Service Information" parameter.<br><br>Q.931 "Higher Layer compatibility" in "Access Transport" parameter. | IAM | 1, 2 |
| `destination_addr` | "Address indicators" and "odd/even indicator" in "Called party number" parameter | IAM | 1 |
| `originating_addr` | "Address indicators" and "odd/even indicator" in "Calling party number" parameter. | IAM | 1 |
| `dest_natureof_addr` | "Nature of address indicator" in "Called party number" parameter. | IAM | - |
| `dest_numbering_plan` | "Numbering plan indicator" in "Called party number" parameter. | IAM | - |
| `dest_int_nw_ind` | "Internal network number indicator" in" Called party number" parameter. | IAM | - |
| `orig_natureof_addr` | "Nature of address indicator" in "Calling party number" parameter. | IAM | - |
| `orig_numbering_plan` | "Numbering plan indicator" in "Calling party number" parameter. | IAM | - |
| `orig_numbering_presentation` | "Address presentation restricted indicator" in "Calling party number" parameter. | IAM | - |
| `orig_numbering_screening` | "Screening indicator" in "Calling party number" parameter. | IAM | - |
| `orig_category` | "Calling party's category" parameter. | IAM | - |
| `orig_number_incomplete` | "Number incomplete indicator" in "Calling party number" parameter. | IAM | - |
| `connected_addr` | "Address indicators" and "odd/even indicator" in "Connected number" parameter. | ANM, CON | 1 |
| `conn_numbering_screening` | "Screening indicator" in "Connected number" parameter. | ANM, CON | - |
| `conn_natureof_addr` | "Nature of address indicator" in "Connected number" parameter. | ANM, CON | - |

| API parameter name | Protocol parameter | Protocol messages | Notes |
|---|---|---|---|
| conn_numbering_plan | "Numbering plan indicator" in "Connected number" parameter. | ANM, CON | - |
| conn_numbering_presentation | "Address presentation restricted indicator" in "Connected number" parameter. | ANM, CON | - |
| conn_number_req | "Connected line identity request indicator" in "Optional forward call indicators" parameter. | IAM | - |
| dest_subaddr | Q.931 "called sub-address" in "Access Transport" parameter. | IAM | - |
| orig_subaddr | Q.931 "calling sub-address" in "Access Transport" parameter. | IAM | - |
| hilayer | Q.931 "Higher Layer Compatibility" in "Access Transport" parameter. | IAM | - |
| lolayer | Q.931 "Lower Layer Compatibility" in "Access Transport" parameter. | IAM | - |
| progress_indicator | Q.931 "Progress" in "Access Transport" parameter. | IAM, ANM, CON, ACM, CPG | - |
| in_band | "In-band indicator" in "Optional backward call indicators" parameter. "In-band information" in "Event information" parameter. | ACM, CPG | - |
| nat_inter_call_ind | "National/international call indicator" in "Forward call Indicators" parameter. | IAM | - |
| interworking_ind | "Interworking indicator" in "Forward call Indicators" and "Backward call indicators" parameters. | IAM, ACM, CON, ANM, CPG | - |
| isdn_userpart_ind | "ISDN User part indicator" in "Forward call Indicators" and "Backward call indicators" parameters. | IAM, ACM, CON, ANM, CPG | - |
| isdn_userpart_pref_ind | "ISDN User part preference indicator" in "Forward call Indicators" parameter. | IAM | - |
| continuity_check_ind | "Continuity check indicator" in "Nature of connection indicators" parameter. | IAM | 7 |

| API parameter name | Protocol parameter | Protocol messages | Notes |
|---|---|---|---|
| satellite_ind | “satellite indicator” in “Nature of connection indicators” parameter. | IAM | - |
| charge_ind | “Charge indicator” in “Backward call indicators” parameter. | ACM, CON, ANM, CPG | - |
| dest_category | “Called party’s category” in “Backward call indicators” parameter. | ACM, CON, ANM, CPG | - |
| add_calling_num_qualifier_ind | “Number qualifier indicator” in “Generic number” parameter. | IAM | - |
| add_calling_num_natureof_addr | “Nature of address indicator” in “Generic number” parameter. | IAM | - |
| add_calling_num_plan | “Numbering plan indicator” in “Generic number” parameter. | IAM | - |
| add_calling_num_presentation | “Address presentation restricted indicator” in “Generic number” parameter. | IAM | - |
| add_calling_num_screening | “Screening indicator” in “Generic number” parameter. | IAM | - |
| add_calling_num_incomplete | “Number incomplete indicator” in “Generic number” parameter. | IAM | - |
| add_calling_num | “Address signals” and “odd/even indicator” in “Generic number” parameter. | IAM | 1 |
| diverting_indicator | “Redirecting indicator” in “Redirection Information” parameter. | IAM, REL | - |
| diverting_reason | “Redirecting reason” in “Redirection Information” parameter. | IAM, REL | - |
| original_diverting_reason | “Original redirecting reason” in “Redirection Information” parameter. | IAM, REL | - |
| diverting_counter | “Redirection counter” in “Redirection Information” parameter. | IAM, REL | - |
| diverting_to_addr | “Address signals” in “Redirection number” parameter. | ACM, ANM, CPG, REL | 1,4 |
| diverting_to_type | “Nature of address” in “Redirection number” parameter. | ACM, ANM, CPG, REL | 4 |

| API parameter name | Protocol parameter | Protocol messages | Notes |
|---|---|---|---|
| `diverting_to_plan` | “Numbering plan” in “Redirection number” parameter. | ACM, ANM, CPG, REL | 4 |
| `diverting_to_int_nw_indicator` | “Internal network number indicator” in “Redirection number” parameter. | ACM, ANM, CPG, REL | 4 |
| `diverting_from_addr` | “Address signals” in “Redirecting number” parameter. | IAM | 1 |
| `diverting_from_type` | “Nature of address” in “Redirecting number” parameter. | IAM | - |
| `diverting_from_plan` | “Numbering plan” in “Redirecting number” parameter. | IAM | - |
| `diverting_from_presentation` | “Address presentation restricted indicator” in “Redirecting number” parameter. | IAM | - |
| `diverting_from_screening` | “Screening indicator” in “Redirecting number” parameter. | IAM | |
| `diverting_to_presentation` | “Presentation restricted indicator” in “Redirection number restriction” parameter. | ACM, ANM, CON, CPG | 5,6 |
| `diverting_to_presentation` | “Presentation restricted indicator” in “Redirection number restriction” parameter. | ACM, ANM, CON, CPG | 5,6 |
| `notification_options` | “Notification subscription options” in “Call diversion information” parameter. | ACM, ANM, CON, CPG | 3 |
| `cause` | “Cause value” in “Cause indicators” parameter | REL | 1 |
| `raw (in cause_xparms)` | “Cause value” in “Cause indicators” parameter | REL | - |
| `location` | “Location” in “Cause indicators” parameter | REL | - |
| `ts_mask` | “Range and status” parameter | GRS, CGB, CGU | - |

**Table 3 notes:**

1. Parameter undergoes automatic translation within the API between Aculab generic format and ISUP protocol format.
2. Generic API does not support all values permitted by the protocol. Values not directly supported by generic API can be accessed using flexible ISUP.
3. Parameter may be retrieved from received messages, but the generic API does not allow this parameter to be encoded in any outgoing messages.
4. Parameter may be retrieved from various messages, but the generic API only allows it to be sent in REL, by using `call_feature_send()` followed by `call_disconnect()`.
5. Parameter may be retrieved from various messages, but cannot be encoded directly via the generic API.
6. As a configuration option, Redirection Number Restriction parameter can be included as a fixed value in all ACM messages.
7. The continuity_check_ind field may accessed via the generic API. It is retrieved from received IAM messages using `call_details()`, and set in outgoing IAM messages using `call_openout()` or `call_feature_openout()`. However, if set in outgoing messages, the ISUP protocol requires that an ISUP COT (Continuity) message is subsequently sent which can only be done by using flexible ISUP to compose the COT as a raw message.

## 8.2 ISUP helper library

To assist in the creation of applications that take advantage of Flexible ISUP features, an ISUP helper library has been created.

The ISUP library consists of the following files:

Windows:

- `isup_lib.dll`
- `isup_lib.lib`
- `isup_lib.h`

Linux:

- `libacu_isup.so.x.y.z` (where `x,y,z` are version numbers)
- `isup_lib.h`

The following functions are provided:

```
isup_get_next_parameter()
isup_get_parameter()
isup_get_pcompat_info()
isup_lib_version()
isup_set_parameter()
```

### 8.2.1 isup_get_next_parameter()

#### Synopsis

```
ACU_UCHAR *isup_get_parameter( RAW_MSG_XPARMS *raw, ACU_UCHAR *params);
```

This function should only be used on `RAW_MSG_XPARMS` received from ISUP via `call_feature_details()`, where the first byte of `data[]` is a message type.

#### Input Parameters

***raw***

Specifies the address of the `RAW_MSG_XPARMS` to be operated on.

***params***

Provides a pointer to the start point for the operation. NULL must be specified to begin parameter retrieval from the start of the received message. If a non-NULL value is specified, it must point to the type byte of a parameter. Iteration through all parameters in a message may be accomplished by providing the pointer returned by the previous invocation.

#### Return Values

The function returns a pointer to the type byte of a parameter, or NULL if no parameter was found.

### 8.2.2 isup_get_parameter()

#### Synopsis

```
ACU_UCHAR *isup_get_parameter( RAW_MSG_XPARMS *raw,
                               ACU_UCHAR type,
                               ACU_UCHAR *params);
```

This function should only be used on `RAW_MSG_XPARMS` received from ISUP via `call_feature_details()`, where the first byte of `data[]` is a message type.

#### Input Parameters

***raw***

Specifies the address of the `RAW_MSG_XPARMS` to be searched.

***type***

An ISUP parameter code from Table 5/Q.763 or equivalent.

***params***

Provides a pointer to the start point for the search operation. NULL must be specified to begin searching from the start of the received message. If a non-NULL value is specified, it must point to the type byte of a parameter. Searches for repeated parameters may be accomplished by providing a pointer to a previously located parameter.

### Return Values

The function returns a pointer to the type byte of a located parameter, or NULL if no parameter was found.

## 8.2.3 isup_get_pcompat_info()

### Synopsis

```
ACU_INT isup_get_pcompat_info( RAW_MSG_XPARMS *raw,
                               ACU_UCHAR type,
                               ACU_UCHAR *info);
```

Returns Parameter Compatibility Information instruction indicators for the specified parameter type, or a negative error code.

### Input Parameters

***raw***

Specifies the address of the `RAW_MSG_XPARMS` to be searched.

***type***

An ISUP parameter code from Table 5/Q.763 or equivalent.

***info***

Specifies a pointer to an ISUP Parameter Compatibility Information parameter. This pointer must refer to the type byte, which should have the value `0x39`.

### Return Values

A zero return value indicates success, while a negative value indicates failure. See the section on error codes that follow.

## 8.2.4 isup_lib_version()

### Synopsis

```
const char *isup_lib_version(void);
```

Returns a pointer to a version string.

## 8.2.5 isup_set_parameter()

### Synopsis

```
ACU_INT isup_set_parameter( RAW_MSG_XPARMS *raw,
                            ACU_UCHAR type,
                            ACU_UCHAR *param,
                            ACU_UINT len);
```

Places a parameter payload into a raw message in preparation for transmission. A negative error code is returned if the requested operation would result in a write operation beyond the `data[]` member of `RAW_MSG_XPARMS`.

### Input Parameters

***raw***

Specifies the address of the `RAW_MSG_XPARMS` that the parameter should be written to. In the absence of errors, the parameter will be placed after any existing data.

***type***

An ISUP parameter code from Table 5/Q.763 or equivalent.

***param***

Specifies a pointer to the data payload for the parameter.

***len***

This parameter specifies the length of the parameter payload, in octets.

### Return Values

A zero return value indicates success, while a negative value indicates failure. See the section on error codes that follow.

### 8.2.6 Error codes

The following table lists error codes that may be returned from the helper library:

**Table 4: Error codes**

| #define | Value | Description |
|---|---|---|
| `ISUP_SUCCESS` | 0 | Success |
| `ISUP_EBADPARAMS` | -1 | Bad parameters |
| `ISUP_EBOUNDSCHK` | -2 | Would require memory access beyond the `data[]` member of `RAW_MSG_XPARMS` |
| `ISUP_ENOTFOUND` | -3 | The requested object was not found |

## 8.3 ISUP feature information

The call control API functions **call_feature_openout()**,**call_feature_send()** and **call_feature_details()** are used to pass additional information between the application and the protocol stack. For outbound requests they allow the information for a single ISUP message be passed using multiple API calls avoiding the need for very large (and ever extending) parameter structures.

ISUP supports the following feature types:

**FEATURE_DIVERSION**: Send/receive diversion or forwarding information.

**FEATURE_USER_USER**: Negotiates the use of, and transfer of user to user information.

**FEATURE_RAW_MSG**: Gives direct access to ISUP messages and parameters:
- add additional parameters to ISUP messages.
- send complete ISUP messages.
- retrieve received ISUP messages.

When sending feature information, multiple features can be added in any order, but only one set of information for each feature type is saved.

**call_feature_openout()** will send the IAM unless **CONTROL_DEFERRED** or **CONTROL_DEFERRED_MESSAGE** is specified in which case it is deferred until a later **call_feature_send()** requests it be transmitted.

**call_feature_send()** will send any built or deferred message if **CONTROL_DEFAULT** or **CONTROL_LAST_INFO** is specified.

A **feature_type** of zero can simplify coding if multiple features are needed on an IAM. Specify zero with **CONTROL_DEFERRED** to **call_feature_openout()**, add the required features with **CONTROL_EXTRA_INFO**, finally use a feature type of zero with **CONTROL_LAST_INFO** to send the IAM.

**Note** **The CONTROL_*xxx*_SETUP requests are treated exactly the same way as their CONTROL_*xxx* counterparts.**

While feature information is available **call_details()** will return with the corresponding bit set (**FEATURE_DIVERSION**, **FEATURE_USER_USER** or **FEATURE_RAW_MSG**) in **feature_information**. To read the feature information use **call_feature_details()** with the correct bit set in **feature_type**. If no new information is queued it will return success, but with **feature_type** set to **0** (some of the fields are filled in with the state values returned on the previous call).

With ISUP there is no requirement that the applications reads the feature information in any specific order, nor that **call_details()** be called first. With an appropriate CBU (crystal ball unit) the feature information can be read without waiting for the relevant call control event.

### 8.3.1 Diversion and forwarding information

Call diversion (Q.730 section 8.7) and call forwarding (Q.732) ISUP supplementary services are similar procedures and use many of the same message parameters. Basically:

- A REL contains the new number to divert the call to (call diversion only).
- An IAM contains the original called number, and that of the system performing the redirect (etc).
- An ACM/CPG/ANM/CON/FAC informs the originating exchange that the diversion/redirection has taken place.

If **FEATURE_DIVERSION** is specified on **call_feature_openout()**, or on **call_feature_send()** with **CONTROL_LAST_INFO** or **CONTROL_EXTRA_INFO** while there is an IAM waiting to be sent, then the ISUP parameters redirecting number (code 0x0b), redirection information (code 0x13) and original called number (code 0x28) are added to the IAM message.

In all other cases the values are saved and the ISUP parameters redirection information (code 0x13) and redirection number (code 0x0c) added to the REL message send when **call_disconnect()** is later called.

Diversion information can also be added using FEATURE_RAW_MSG. That is the only way it can be added to the other message types, or that parameters other than the above can be added.

If a received IAM contains redirection information (code 0x13), redirecting number (code 0x0b) or original called number (code 0x18) the values are saved and **FEATURE_DIVERSION** indicated. If there is a redirection number, or the diverting indicator is non-zero and **rnr_value** has been configured, then the ACM will contain a redirection number restriction parameter with the configured value.

If a received ACM, CPG, ANM or CON contains call diversion information (code 0x36), redirection number (code 0x0c) or redirection number restriction (code 0x40) the saved values are updated and **FEATURE_DIVERSION** indicated once the call is alerting/ringing.

When a REL is received, any existing diversion information is zeroed. If the REL contains redirection information (code 0x13) or redirection number (code 0x0c) the values are saved and **FEATURE_DIVERSION** indicated.

### 8.3.2 User to user information

The ISUP procedures for user to user information are defined in Q.737.1. There are three service levels. Service 1 for sending UUI in normal setup and cleardown messages, service 2 for sending USR messages during call setup, and service 3 for sending USR messages once the call is established. Services 2 and 3 must be negotiated before being used, service 2 must be requested in the IAM, service 3 can be requested in a later FAR message.

The driver will stop user to user information being sent when it is hasn't been correctly negotiated.

If **FEATURE_USER_USER** is specified on **call_feature_openout()**, or on **call_feature_send()** with **CONTROL_LAST_INFO** or **CONTROL_EXTRA_INFO** and while there is as IAM waiting to be sent, then the user to user information is added to the IAM message.

Responses to requests received on an IAM should be sent with **CONTROL_NEXT_CC_MESSAGE** so that the UUS parameter is added to the next backwards message (this need not be the first backwards message), if necessary use **call_progress()** to send an additional CPG.

Any FAR/FAA/FRJ negotiating UUS 3 is always sent immediately. As is any REL generated because the service was marked 'essential' and rejected.

If a request from the application contains actual user to user data then it is sent in a USR message if **CONTROL_DEFAULT**, **CONTROL_DEFERRED** or **CONTROL_DEFERRED_MESSAGE** is requested, otherwise it is queued and added to the next backwards message (ACM, CPG, ANM etc).

If a received message contains user to user information or indicators then **FEATURE_USER_USER** is indicated, if the message is an IAM then no ACM is sent.

Receipt of facility messages (FAR/FAA/FRJ) raise **EV_EXT_UUS_SERVICE_REQUEST**. Receipt of UUI data raises EV_EXT_UUI_PENDING if there is no other UUI data queued.

The first byte of the user to user information (the protocol discriminator) passed to/from the application in a separate field. A valid protocol discriminator with a zero length user information field is indicated by setting the first byte of the buffer to **UUI_NO_DATA_IN_BUFFER** (0xff)

When **FEATURE_USER_USER** is read, the state information and oldest user information buffer is written into the application's buffer. If the values haven't changed the **feature_type** is cleared.

### 8.3.3 Raw parameters and messages

The requirements of applications and networks are too diverse to make is sensible to add API calls and parameters for all the ISUP message parameters and procedures.

Setting **CNF_RAW_MSG** in the **cnf** field to **call_openin()** or **call_openout()** requests that a (reformatted) copy of every received ISUP message be passed to the application. They are read by calling **call_feature_details()** with **feature_type** set to **FEATURE_RAW_MSG**.

The message type code is written to **raw_msg.data[0]** with any parameters written to **raw_msg.data[1]** onwards encoded code+length+data. This means that the application doesn't have to know which parameters are fixed and mandatory variable.

**Note** **If multiple subsequent address messages (SAM) are queued, they may be merged into a single raw message.**

The raw message data is queued (and readable) at the start of the message processing. If the raw message queue was empty an **EV_EXT_RAW_MSG** event is raised before the first other call control event. If no call control event is raised than an **EV_EXT_RAW_MSG** event is always raised at the end of message processing.

**Note** **Since an EV_EXT_RAW_MSG event is not raised if there is an earlier unread raw message an application may see fewer than expected events if local processing is delayed. The application should ensure that all the raw messages are read**

The ISUP message data that generated a call control event can be identified by looking at the **raw_msg_seq** field of the raw data and event structures. Events that are not associated with received messages have **raw_msg_seq** set to zero, otherwise the value is incremented just before raising the first normal call control event after a raw message is queued (so any **EV_EXT_RAW_MSG** event has the same value as the raw message, and the other events a value one higher).

**Note** **It might have been better to increment the sequence before generating the raw message data, but the value has always been generated this way.**

If **call_feature_openout()** or **call_feature_send()** requests **FEATURE_RAW_MSG** with **CONTROL_DEFAULT** (or, unusually, **CONTROL_DEFERRED_MESSAGE**) a complete ISUP message is built and sent (deferred to be sent later); **raw_msg.data[]** should be formatted the same as for received raw messages above.

Sending a raw message will not change the call state, ISUP message types that would change the call state cannot be sent as raw messages.

**call_feature_openout()** will use the generated message instead of the IAM it would normally create. This can be used in order to make a continuity test call.

If **call_feature_openout()** or **call_feature_send()** requests **FEATURE_RAW_MSG** with any other **CONTROL_*xxx*** values then it defines a list of parameters that will be used to modify the next ISUP message sent by an API call (or a message deferred by an earlier **call_feature_openout()** or **call_feature_send()** call).

The parameters are passed in **raw_msg.data[0]** onwards encoded code+length+data.

Usually any existing parameter with the same code is replaced. If the parameter can be repeated (e.g.: generic number) then an additional copy is added. Specifying a length of zero will always delete any existing parameter.

**Note** **Any parameter not in the codec tables will be discarded when the message is finally encoded. If a fixed or mandatory variable parameter is absent the message itself is discarded.**

The helper functions defined in section 8.2 can be used to access the parameter data.

See section 8.5.3 and Appendix A for examples.

## 8.4 Transmission path switching

In addition to the call setup functions of the call control API, an ISUP application must also use the Aculab Switch API to set up the bearer (voice or data) path for each call, known as the "transmission path". The path in the direction from calling to called subscriber is known as the "forward" path, whilst the path from called to calling subscriber is known as the "backward" path. The forward and backward paths may be completed at the same time, or each may be connected at different times, during call establishment.

Careful consideration always needs to be given to the synchronisation between the Call Control API activity and switching of forward and backward transmission paths, and there may sometimes be slightly conflicting requirements. Network providers, for example, may impose restrictions to avoid "early" switching to avoid unauthorised data flow (or conversation) before charging begins. In the case of a voice call however, subscribers may feel entitled to expect that switching is already completed by the time the called subscriber first speaks, which may be just a fraction of a second after the call enters the "connected" state.

The actual rules governing transmission path switching are often a local matter subject to bilateral agreement between the Aculab customer and the network, but this section attempts to describe some of the issues that need to be addressed and some of the strategies that may be adopted.

### 8.4.1 Originating exchanges

The guidance given in ITU-T Q.764 is that the backward transmission path may be connected immediately after the Initial Address Message (IAM) has been sent, which corresponds in the Aculab API to a successful return from `call_openout()`. In some circumstances, such as if a continuity test is being performed, this may not be appropriate as it could expose the calling subscriber to unwanted test tones. Where the backward path cannot be connected immediately, it may be better to wait for the first event notification ("`EV_OUTGOING_PROCEEDING`") before connecting the voice paths (this is in line with the API guidance for other Aculab protocols). Specific applications may be constrained by other factors (such as network-specific rules and regulations) that require them to wait until later in the call setup sequence before connecting the backward paths. In this case extra care should be taken to cater for unexpected tones and announcements that should be played to the caller, as described in section 8.4.4.

The Q.764 guidance for switching of the forward path is that it should normally be completed on receipt of a Connect or Answer message, which corresponds to an `EV_CALL_CONNECTED` event. For 3.1KHz audio calls, Q.764 also allows the forward path to be completed at the same time as the backward path (immediately after `call_openout()`). This may, for example, be required for calls that might undergo interactive dialogue (e.g. DTMF or voice recognition) during call establishment.

### 8.4.2 Intermediate exchanges

The guidance in Q.764 is that the path in both directions can normally be connected immediately after calling `call_openout()`. As with originating exchanges however, there are exceptions such as continuity testing, where the same considerations should be applied.

### 8.4.3 Destination exchanges

The destination exchange is responsible for completing the connection path to the called party apparatus. In the case of a voice call, for example, this may be the connection to the subscriber line.

In most cases, the destination exchange can complete the path in both directions as soon as the decision to answer the call is taken. In ISUP protocol terms this corresponds to an Answer or Connect message being sent, while in terms of the Aculab API, it corresponds to calling `call_accept()`. In the case of a voice call, completing the voice path in both directions immediately before `call_accept()` is usually appropriate.

### 8.4.4 Tones and announcements - general

Availability of tones or announcements is conveyed in ISUP messages using an “in-band indicator”, which appears in various ISUP protocol parameters such as the “optional backward call indicators”. It is used during call setup to indicate that audible messages, such as ring tone, are being connected and that the backward transmission path should be completed to the calling subscriber. This applies to all tones and announcements. Further advice specific to ring tone is provided in section 8.4.5.

At the Aculab API, the in-band indicator appears in many of the call control structures allowing it to be set in messages that are sent. It is also available for reading from the `uniquex_isup` structure used by `call_details()`, which allows an application to detect that the remote exchange is applying tones or announcements. When making an outgoing call, if the backward path is not connected when the call commences, then `call_details()` should be called after all event notifications to see if the `in_band` field is set. If it is set, the backward voice path should be connected.

As described in section 2.4.3, a call in an Intelligent Network may require DTMF or voice interaction with the caller during the call setup phase in response to tones or announcements from the network. In that case, the in-band indicator would still be used, but it would be necessary to ensure the forward path is also connected. Such issues may need to be discussed with the network provider when an application is being developed.

### 8.4.5 Ring tone

For ISUP calls, ring tone is usually generated by the destination (called) exchange. This is preferable to ring tone generated by the originating exchange as it removes certain delays that would otherwise be required by subsequent voice path switching when the call is answered. This is clearly evident when making international voice calls to fixed-line phones when the ring tone heard by the caller is usually that of the remote country’s network.

In some situations, a call may originate in an ISUP network, but subsequently pass through a different network, such as a mobile network, in which case special arrangements may exist for ring tone. For example, the exchange that provides inter-working with the other network may provide ring tone.

Exchanges that provide ring tone should indicate it using the ISUP protocol in-band indicator as described in section 8.4.4, but many exchanges do not conform to this requirement and may fail to set the in-band indicator even though they provide ring tone. If in doubt when developing an application, it is best to ask the network provider to confirm which exchange should provide ring tone and/or whether the in-band indicator can be relied upon to detect it.

Applications using Aculab ISUP that generate ring tone, should always set the “`in_band`” field in the corresponding call control API structures.

## 8.5 ISUP Continuity test

The ISUP procedures allow bearer continuity tests to be done either during call establishment (requested in the IAM) or as a separate test (CCR). The CCR test is also done periodically after an initial failure. Since an ISUP call might pass through multiple ISUP exchanges, a continuity test on an IAM can be on the 'current' or 'previous' circuit. The procedures for these are similar, except that the voice path loopback isn't required if a previous circuit is being tested.

The basic procedure is:

a) An IAM is sent with 'continuity check required on this circuit' set in the 'Nature of connection indicators'. 2000±20Hz (2010±8Hz for ANSI) tone sent on the voice channel, and a tone detector placed on the receive voice channel.
b) On receipt of the IAM a loopback is applied to the voice channel.
c) If the reflected tone is received a COT message is sent indicating 'success' and the tone removed, on receipt of the COT the loop is removed and call processing continues.
d) If the reflected tone isn't received a COT message is sent indicating 'failure' and the tone removed, on receipt of the COT the loop is removed and the call discarded.
e) After a delay a CCR is sent to perform a retest, tone sent and detector enabled.
f) On receipt of the CCR a loopback is applied to the voice channel (and for ANSI an LPA sent).
g) If the reflected tone is received a REL is sent to release the circuit.
h) If the reflected tone is not received a COT message is sent indicating 'failure' is sent and the CCR procedure is repeated.

Refer to Q.764 section 2.1.8 and Q.724 section 7 for further details.
The receiving system runs appropriate timers and will reset the circuit if they expire.
The retest procedures are aborted if an IAM is received.
ANSI ISUP may require a 2-wire test be performed, in this case the loopback has to be replaced with a tone detector and a 1780±20Hz tone sent back while the 2008Hz tone is being received.

The Aculab ISUP driver code has support for inward continuity test requests, but as it is not capable of generating or detecting tones only partial support for outbound continuity tests. An application can use TiNG to generate and detect the tones for outbound tests.

### 8.5.1 Inward continuity check (IAM)

The behaviour of the driver on receipt of an IAM that requests continuity check depends on the **`apply_continuity_loop`** and **`continuity_defer_event`** ISUP configuration parameters.

If both parameters are set to **`yes`** (the default) then the driver will apply a loopback. The **`EV_INCOMING_CALL_DETECTED`** (and **`EV_RAW_MSG`**) call control events will not be generated until a COT is received indicating that the test has passed. If the continuity test fails the application will see an **`EV_IDLE`** event and should do another **`call_openin()`**.

With the default parameters inward continuity test should be transparent to the application.

If **`continuity_defer_event=no`** then the call control events are generated immediately. The application must look at the **`continuity_check_ind`** field of the call details and if it is non-zero wait for the **`EV_DETAILS`** event generated when COT is received (after which **`continuity_check_ind`** will be zero) before connecting the voice path or accepting the call. Any ACM will be buffered by the driver and sent when the continuity test completes.

If **`apply_continuity_loop=no`** as well then the driver will not apply the loopback, the application must apply a loop when an incoming call is received with **`continuity_check_ind`** is set to **`CCI_REQUIRED`** (value 1). This allows a 2-wire transponder be created.

At the end of the test the driver may remove any loopback it applied (based on the configured **`continuity_check_output_value`**). The default is A-law silence for ITU/China variants and µ-law silence for ANSI variants.

**Note** **The driver does not currently remove the loop if the test passes; this ensures that the driver doesn't set the output data after the application has connected the audio path.**

### 8.5.2 Inward continuity check test call (CCR)

The behaviour of the driver on receipt of a CCR depends on the **apply_continuity_loop**, **ccr_application** and **ccr_auto_lpa** ISUP configuration parameters.

By default the CCR test is handled within the ss7 driver.

If **ccr_application= yes** then a received CCR is passed to the application as an incoming call with **continuity_check_ind** set to **CCI_CCR_TEST** (value 4).

If additionally **apply_continuity_loop=no** the application can then add an appropriate loop - e.g.: a 2-wire transponder.

For ANSI networks an LPA is sent unless **ccr_auto_lpa=no**, in which case the application should create the loop and then use flexible ISUP to send an LPA.

At the end of the test (e.g.: COT + fail or REL received) the driver will use the configured **continuity_check_output_value** to remove any loop it applied, and an **EV_IDLE** sent to any application.

### 8.5.3 Outbound continuity test (IAM)

A continuity test can be requested by making an outward call with **continuity_check_ind** set to **CCI_REQUIRED** (test on this circuit) or **CCI_PREVIOUS** (test on previous circuit). In the former case the application is responsible for generating and checking the tone.

Once the tone has been verified (or an indication that the test on the previous circuit has been completed) the application must remove the test tone use flexible ISUP to send the COT message. The following C function could be used:

```
ACU_ERR
send_continuity(ACU_CALL_HANDLE handle, int passed)
{
    FEATURE_DETAIL_XPARMS fdxp;
    INIT_ACU_CL_STRUCT(&fdxp);

    fdxp.handle = handle;
    fdxp.feature_type = FEATURE_RAW_MSG;
    fdxp.message_control = CONTROL_DEFAULT;     /* Whole message specified */
    fdxp.feature.raw_msg.length = 4;

    fdxp.feature.raw_msg.data[0] = 0x05;        /* Continuity */
    fdxp.feature.raw_msg.data[1] = 0x10;        /* Continuity indicators */
    fdxp.feature.raw_msg.data[2] = 0x01;        /* Length */
    fdxp.feature.raw_msg.data[3] = passed;      /* 1 if passed, 0 if failed */

    return call_feature_send(&fdxp);
}
```

If the test is successful the call processing continues as normal.

If the test fails the application can either use the same call handle for the retest, or call **call_release()** and perform the retest on another call handle. Releasing the call handle after a continuity test has failed is a purely local action.

### 8.5.4 Outbound continuity check test call (CCR)

To send a CCR on a new call handle, use **call_openout()** with **continuity_check_ind** set to **CCI_CCR_TEST**, this will send a CCR instead of an IAM.

To send CCR on an existing call handle (after a previous continuity test has failed) use flexible ISUP. As **send_continuity()** above except set **raw_msg.length = 1** and **raw_msg.data[0] = 0x11**.

The application then needs to generate and test the tone (on ANSI networks it should wait for LPA first).

Test failure is indicated by sending COT + fail (as for continuity test on IAM above, in this case the driver will convert all COT to COT + fail). The application should then schedule another retest.

Test success is indicated by sending a REL - use **call_disconnect()**.

## 8.6 System performance considerations

Aculab ISUP allows customers to construct large-scale platforms with many hundreds or thousands of voice channels. Such applications need to be carefully structured to ensure that the processing power of the platform is sufficient to meet the demands of call traffic.

It is assumed that developers will already posses the necessary programming skills to implement efficient applications. This section provides further insight into some aspects of performance optimisation that are specific to the Aculab ISUP API.

### 8.6.1 Maximising throughput

The following guidelines may be of assistance in writing performance-critical applications:

- Avoid unnecessary API calls. For an incoming call, apart from handling event notifications, it is often sufficient simply to call `call_incoming_ringing()`, and then sometime later to call `call_accept()`. There are some circumstances where `call_proceeding()` or `call_progress()` may need to be called, but if they are not needed they should not be used.

- Avoid “thread-heavy” applications. With most platforms, multi-threaded applications become inefficient with large numbers of threads. If an ISUP application is handling tens or more of E1/T1 trunks, thread management within the OS can become a serious bottleneck if a separate thread is used for each timeslot. A better approach for a large-scale application might be to have a thread per E1/T1 rather than a thread per timeslot. Alternatively, a “worker thread” strategy may be possible where threads are dispatched to perform tasks as and when they arise, without permanently associating a task with any single timeslot or network port.

### 8.6.2 Avoiding missed calls

Missed calls can occur when a circuit (timeslot) is re-used for a call immediately after a previous call on the same circuit has been cleared down, but before the application has had time to open a new handle using `call_openin()`. Aculab ISUP offers various solutions to this problem.

- “Call queuing” - The driver can be configured to silently queue calls that repeat rapidly on a timeslot, to allow a short additional time (up to 900mS) for an incoming handle to be opened. If an incoming handle is opened within the queuing time, it is presented with the queued call. Otherwise the call is rejected when the queuing timer expires.

- “Timeslot -1” - The application can maintain a pool of handles, all using timeslot -1. The API treats -1 as “any timeslot”, so any handle in the pool can accept the next call regardless of which timeslot it concerns.

- “Early openin” - The application can open a new handle, using `call_openin()`, for the same timeslot before disconnecting (or releasing) the previous call on the same timeslot. This ensures there is a handle available for each call as soon as it arrives. After opening the new handle the application proceeds to disconnect and release the handle of the previous call, regardless of whether a new call has arrived.

The “call queuing” solution is very similar to the approach used in other Aculab protocols. It is configured using the “`isup_timer_openin`” configuration option as described in the ***SS7 Installation and administration guide***. The drawback of this approach is that ISUP protocols may set strict limits on the latency that is allowed when responding to a message. It is generally expected that an exchange should respond immediately (either pass on the call or reject it), but this approach introduces a delay that runs counter to the intentions of the protocol.

The “early openin” and “Timeslot –1” solutions overcome the drawback of delay introduced by call queuing, they each allow a well-written application to always react immediately to an incoming call. Note however, that not all Aculab protocols allow a new handle to be opened before closing the old one for the same timeslot, so application code that uses the “early openin” approach may not be portable to other protocols.

## 8.7 Using Aculab ISUP with national variants

Aculab SS7 has built-in “basic call setup” support for some of the common variants such as ITU-T, ANSI, and China, but there are many other national variants in use around the world. To allow other variants to be supported, or for further tuning of an existing variant, users are able to add or modify message and parameter definitions by defining coder/decoder (codec) extensions, which the SS7 driver uses when encoding and decoding network messages. Messages and parameters that have been defined with codec extensions can then be accessed at the API using the flexible ISUP API extensions, as described in section **Error! Reference source not found.**.

By selecting one of the basic pre-configured variants, modifying appropriate parameters such as timers and extending the message codec tables, it becomes possible to use Aculab SS7 ISUP in many parts of the world above and beyond those areas in which it has already been deployed. Full details of parameter and codec configuration are provided in the ***SS7 Installation and administration guide***.

# 9 Using Aculab SCCP

SCCP may be used to convey TCAP message traffic via the TCAP API or may be used directly through an SCCP API.

The APIs provide the necessary functions for sending and receiving SCCP address information and also receiving SCCP status information (See sections 9.1 and 9.2).

The SCCP must also be correctly configured with parameters that correspond to those of the peer SCCP nodes with which it converses. It may be configured to automatically perform global title address translation on messages sent and received. For a full description of how to configure Aculab SCCP, please refer to the ***SS7 Installation and administration guide***

## 9.1 SCCP Addressing

TCAP and SCCP API users need to provide SCCP addressing information when sending messages. The information may be provided in a user-provided text configuration file or via the API itself. SCCP address information from received messages can be obtained through the API. The relevant functions for configuring, sending and receiving SCCP addresses are documented in the TCAP API guide (see also section 10.2.5) and the SCCP API guide.

### 9.1.1 Routing on SSN

This is the simplest method of sending an SCCP message. The API user provides a remote point code and subsystem number for SCCP to deliver the message to. SCCP uses the remote point code as a destination point code for the underlying transport (MTP3/M3UA) and encodes the SSN into the called party address. The calling party address may be routed on SSN or routed on global title and is usually obtained by the text configuration file of the API.

For received messages routed on SSN, the SCCP will distribute the message to the appropriate application using the SSN in the called party address.

### 9.1.2 Routing on Global Title

There are two methods an SCCP user can send messages routed on global title. In the first method, the application provides a remote global title and a remote point code and optionally an SSN. In this case, SCCP has sufficient information to send the message and a global title translation is not required. The remote point code is used as a destination point code for the underlying transport and the global title is encoded into the called party address.

In the second method, the application provides a global title only. Since SCCP requires a destination to send the message to, global title translation will be attempted. If a rule is found that matches the supplied global title, SCCP will apply the translation and send the message to the destination specified by the rule.

To receive messages routed on global title, a global title translation table has to be written, specifying a [ROUTE_SSN] subsection. The destination should be set to the local point code and optionally, the SSN of the SCCP user that wishes to receive the message. If no SSN is specified in the rule, then SCCP will use the SSN from the called party address of the received message. The message will be delivered to the SCCP user with both the called and calling party addresses unmodified; that is as they were received from the network. Hence, if a called or calling party address does not contain a point code, then a point code will not be included as part of the global title. However, it is possible to obtain from the API the point codes from the MTP routing label.

## 9.2 SCCP status information

The Aculab TCAP/SCCP APIs include functions, which provide the user with SCCP status information about a remote system. In addition an event-driven mechanism, using the message interface, provides notification of changes in status. Such event reporting is disabled by default, so if the user wants to see these events then they must be enabled using the relevant functions described in the TCAP or SCCP API guides.

# 10 Using Aculab TCAP

The Aculab distributed TCAP API provides the means for an application to use the services of the Transaction Capabilities Application Part, as defined in ITU-T recommendations, currently Q.771-Q.775. The API also supports ANSI TCAP, which is described in ANSI T1.114.

Section 10.1 provides an overview of those aspects of TCAP protocol procedures and terminology that are relevant to the Aculab API. For a detailed and authoritative description of TCAP, refer to the aforementioned ITU and ANSI recommendations.

Section 10.2 provides specific guidance on how to use the Aculab API to implement a TCAP application. The API is described in detail in the TCAP API guide, to which you should refer for a full definition of the interface functions and how they should be used.

## 10.1 TCAP protocol and procedures

### 10.1.1 TCAP messages

TCAP applications communicate with one another using the SS7 SCCP network functions to exchange messages.

Each TCAP message contains a number of different so-called ‘portions’. All messages contain a ‘transaction portion’ that is provided in Aculab’s case by the TCAP library. The transaction portion includes the TCAP message type (known in ANSI TCAP as “package type”), and various transaction-specific protocol parameters such as the transaction identifiers described in section 10.1.6.

Some messages may also contain either or both of two optional portions, called the ‘component portion’ and the ‘dialogue portion’. The dialogue portion is mandatory in some messages and under some circumstances, but optional at other times. These portions are described in sections 10.1.2 and 10.1.3 respectively.

### 10.1.2 TCAP components and operations

TCAP ‘components’ are the means by which local and remote TCAP users request that one another perform operations and by which they reply to such operations.

There are four different component types, namely ‘invoke’, ‘return result’, ‘return error’, and ‘reject’. The ‘invoke’ component initiates a new operation, for which the TCAP user supplies an ‘invoke identifier’. The other component types are alternative responses to an earlier invoke, and contain the same invoke identifier. Each TCAP message may contain several different components in its component portion, allowing several operations to be invoked by the same message, or a mixture of replies and new invocations.

As an example of TCAP component use, a TCAP user may need to request that a remote database translates a given telephone number to some other telephone number. Such a translation is an ‘operation’. It is requested by sending an ‘invoke’ component that specifies the translation operation together with a parameter containing the number to be translated. In response, the remote database completes the operation by sending a ‘return result’ component together with a parameter containing the translated number.

### 10.1.3 TCAP dialogues and transactions

TCAP provides users with a means to establish a dialogue with some remote TCAP user elsewhere in the network. There are two types of dialogue, known as ‘structured’ and ‘unstructured’, as described later in this section. Within either type of dialogue, TCAP users are able to send and reply to components.

Dialogues, as perceived by the TCAP user, are implemented by TCAP as ‘transactions’. In order for a TCAP user to establish a dialogue, TCAP needs to establish a transaction. In many respects the terms ‘transaction’ and ‘dialogue’ can be used interchangeably.

TCAP messages may include an area called the ‘dialogue portion’. This allows the TCAP users to convey information to one another that identifies the purpose of the dialogue.

Examples are the inclusion of an 'application context name' which defines the protocol that is being used (e.g. GSM-MAP), and the protocol version. Other application-specific user information may also be included. The dialogue portion is used at the start of a dialogue to confirm that the two users have implemented the same (or compatible) versions of a protocol.

In some circumstances a dialogue portion is mandatory, but more often it is only included in the message if requested by the API user. When a dialogue portion is mandatory and is not provided by the user, the Aculab TCAP library will automatically include an appropriate portion.

### 10.1.4 Unstructured Dialogue

The unstructured dialogue has no specific beginning or ending. The two TCAP users send components to one another, requesting and/or responding to different operations in 'single shot' mode and expecting no replies.

### 10.1.5 Structured dialogue

A structured dialogue has an explicit beginning and either an explicit or implicit end.

To start a structured dialogue the user requests TCAP to establish a transaction, by sending a 'BEGIN' message which may contain operation invocation components. Thereafter, the two users continue the dialogue by exchanging TCAP 'CONTINUE' messages in the same transaction. The CONTINUE messages may contain components that reply to previous invocations, and/or that invoke new operations.

A structured dialogue may be explicitly ended with a TCAP END message or, in the case of exception conditions, a TCAP ABORT message. A dialogue may also be implicitly ended using a "pre-arranged end". If both users are able to recognise that a point has been reached where there are no more messages to be sent, the TCAP transaction can simply be discarded.

ANSI structured dialogue is similar to ITU in concept, but uses different message types and naming conventions.

### 10.1.6 Dialogue identifiers

The ITU TCAP recommendations refer to a 'dialogue identifier', which allows TCAP users to associate messages that belong to a single dialogue with one another. The dialogue identifier is only meaningful to the local TCAP user and it is not transmitted in TCAP messages, hence it is a rather abstract concept that can be implemented in different ways.

In practice, a typical application might want to associate the address of some block of memory with each dialogue, so the Aculab API implements the dialogue id as a (void *) 'userptr' that may be associated with a transaction. After the API user has associated a userptr with a transaction, the same userptr will be returned to the API user when subsequent messages are received within the transaction.

### 10.1.7 Transaction identifiers

Transaction identifiers are similar to dialogue identifiers (see section 10.1.6), but unlike dialogue identifiers, which only have local-user significance, the transaction identifiers are encoded within the TCAP messages sent to other applications. In Aculab's case, transaction identifiers are managed entirely within the TCAP library.

Although the API user may completely ignore transaction identifiers, the values can be accessed at the Aculab API. This can be useful when testing or debugging, as it allows diagnostic data captured from a line trace to be correlated with a specific dialogue as seen by the API user.

## 10.2 Writing Aculab TCAP applications

This section of the document provides some guidance on how to write TCAP applications using the Aculab API. It is intended simply as an introduction, and is not intended to be an exhaustive definition of the API.

### 10.2.1 Configuration parameters

Most TCAP parameters need to be configured on a per SSN basis. For all TCAP parameters, the API user has the choice of either specifying the name of a plain text configuration file for each SSN, or applying (or modifying) the configuration parameters using a programmatic interface at the API.

If TCAP configuration parameters are included in a text file, the name of the file is provided when the SCCP Service Access Point is created, as mentioned in section 10.2.3. Individual parameters can be set using `acu_tcap_set_cfg_int/str()`.

### 10.2.2 ASN.1 API parameters

The format of TCAP messages, and the different portions within the message, is defined in a language called Abstract Syntax Notation 1 (ASN.1). ASN.1 is a sophisticated, and sometimes complex notation but TCAP uses ASN.1 in one of its simplest forms known as Basic Encoding Rules (BER).

To a large extent, the Aculab TCAP library handles the ASN.1 encoding and decoding, simplifying things for the API user. Nevertheless, some parts of the component and dialogue portions need to be encoded by the user in ASN.1. An API user has a number of alternative ways of encoding/decoding the ASN.1 data, some of which are described in section 10.2.10.

The following API parameters convey information that may need to be obtained from an ASN.1 application protocol specification, such as MAP or INAP:

- Component parameter data in transmitted messages, using the ‘param’ and ‘param_len’ parameter for acu_tcap_msg_add_comp_invoke(), acu_tcap_msg_add_comp_result(), acu_tcap_msg_add_comp_error() and acu_tcap_msg_add_comp_reject().
- Component parameter data in received messages, from the tc_param and tc_param_len fields in an acu_tcap_component_t structure resolved by acu_tcap_msg_get_component().
- Dialogue user information in transmitted messages, using the ‘uinfo’ and ‘len’ parameters for acu_tcap_msg_add_dlg_userinfo().
- Dialogue user information in received messages, from the td_ui and td_ui_len fields in an acu_tcap_dialogue_t structure resolved by acu_tcap_msg_get_decode().
- Global operation identifiers in transmitted messages, using the op_code and op_code_len parameter for acu_tcap_msg_add_comp_invoke(), acu_tcap_msg_add_comp_result().
- Global operation identifiers in received messages, from the tc_op_code and tc_op_code_len fields in a component structure resolved by acu_tcap_msg_get_component().
- Application context name in transmitted messages, using the appl_ctx and appl_ctx_len parameters for acu_tcap_msg_add_dialogue(). Note however that the ANSI protocol variant also supports the use of numeric values, in which case the API user can simply pass it as an integer.
- Application context name in received messages, from the td_app_ctx and td_app_ctx_len fields in an acu_tcap_dialogue_t structure resolved by acu_tcap_msg_get_component(). Note however that the ANSI protocol variant also supports the use of numeric values, in which case it is presented in the td_app_ctx_val field.

### 10.2.3 SCCP Service Access Points

Before a dialogue can be started, the TCAP Application needs to perform an initialisation sequence that includes establishing a connection with the SCCP protocol layer that will convey messages to and from the remote application(s). Details of the connection, and associated SCCP addressing parameters, are maintained in an SCCP Service Access Point structure (SSAP).

SSAPs are created using `acu_tcap_ssap_create()`, and the connection with SCCP is requested using `acu_tcap_ssap_connect()`. Other `acu_tcap_ssap...()` functions are provided for modifying SSAP configuration, accessing addresses, and deleting SSAPs.

### 10.2.4 TCAP transaction structures

A TCAP transaction structure is needed in order to exchange messages with a remote TCAP user. These structures can be created on demand, or may be created automatically depending upon the type and direction of the dialogue. The transaction structure inherits the current values of its configurable parameters (including the SCCP address information) from the SSAP.
Transaction structures (`acu_tcap_trans_t`) can be created for outgoing dialogues using `acu_tcap_transaction_create()`. For incoming dialogues, they are created automatically when the first message is received. At the end of a transaction, they are be deleted using `acu_tcap_transaction_delete()`. Additional `acu_tcap_trans...()` functions allow configuration modifications to be made, and provide access to transaction addresses, transaction ids, etc.

### 10.2.5 Building messages

In order to send a TCAP message, the user must provide the address of a valid transaction structure which TCAP will use to derive appropriate protocol parameters and queue the message for transmission. The required sequence of API calls can vary, and the following is just one example:

- `acu_tcap_msg_alloc()` is used to allocate a buffer wherein the message will be built. As an alternative to this function, an existing message buffer that was previously used within the same transaction can be re-used.
- `acu_tcap_trans_get_remaddr()` may be used to obtain the address of the structure containing the address information. This is only necessary if the user wants to modify the SCCP addressing for the message.
- `acu_tcap_msg_init()` is used to initialise the message, and set the message type (BEGIN/CONTINUE/END etc.).
- `acu_tcap_msg_add_dialogue()` may be used of the user wants to add information to the dialogue portion.
- If components are to be included, pass them to the API using `acu_tcap_msg_add_comp_invoke()/_result()/_error()/_reject()` may be called repeatedly to add components to the message.

### 10.2.6 Sending messages

After a message has been built as described in section 10.2.5, it is sent to the remote TCAP user as follows:

- `acu_tcap_msg_send()` causes the message to be sent to a remote TCAP user over the SS7 network.

After sending the message the message buffer must either be freed, or re-used for sending another message within the same transaction:

- `acu_tcap_msg_free()` may optionally be used to free the message. Alternatively, follow the steps in section 10.2.5 to re-use the message buffer for another message.

### 10.2.7 Receiving messages

The API user may receive TCAP messages from a remote user, and/or may receive locally generated messages from the local SCCP, advising of MTP or SCCP status changes. Such messages may be received using one of the three functions listed below:

- `acu_tcap_ssap_msg_get()` can be used to receive any message that meets the SCCP address requirements that were provided when the SSAP was created. This function blocks until a message is available or a timeout expires.

- `acu_tcap_trans_msg_get()` can be used to receive only messages that belong to an existing transaction. This function blocks until a message is available or a timeout expires.

- `acu_tcap_event_msg_get()` may be used to receive messages from an event queue (see section 10.2.9), following an event indicating that a TCAP message is available. This function does not block, it returns immediately even if no message is available.

After receiving the message, the message buffer must either be freed, or re-used to send a message within the same transaction:

- `acu_tcap_msg_free()` may optionally be used to free the message. Alternatively, follow the steps in section 10.2.5 to re-use the message buffer for another message.

**Note** **Messages returned by these functions reference the 'ring buffer' used to receive data from the TCP connection to SCCP. Failure to free them will cause the receive side to block.**

### 10.2.8 Decoding messages

After a message has been received using one of the functions described in section 10.2.7, the information that it contains can be obtained using the following steps:

- `acu_tcap_msg_decode()` performs an initial message decode, identifying the message type (BEGIN/CONTINUE/END etc.), and identifying the transaction to which the message belongs. If the message is the first message for a new incoming transaction, the transaction buffer is automatically allocated and added to the `tm_trans` field of the message structure. If the message contains a dialogue portion, this function also resolves a pointer to a structure that describes it.

- If the message is an SCCP status report, then `acu_tcap_msg_get_sccp_status()` may be called to resolve a pointer to a structure that describes the status at the time the message was generated. A separate API funcion, `acu_tcap_get_sccp_status()`, may be called to resolve a similar structure that provides the current status, which may have changed during the time the message was queued.

- `acu_tcap_msg_has_components()` determines whether or not the message contains any components. This step is optional, and is not a prerequisite to receiving the individual components as described below.

- `acu_tcap_msg_get_component()` may be called to obtain a structure that describes an individual component from the message portion. The API user must then process the component after which, if further components may be present, the function may be called again.

### 10.2.9 Message event notification

It is often the case that an application needs to be able to wait for TCAP messages in the same thread as that used for waiting for events from other parts of the system, such as ISUP call control activity. In these circumstances the functions `acu_tcap_ssap_msg_get()` and `acu_tcap_trans_msg_get()` are unsuitable, as some other event could arrive while the

calling thread is blocked within the TCAP library.

The event mechanism allows an application thread to wait for both TCAP messages and other events in a single thread, using the platform's native event notification mechanism (`poll/select` for Unix-style systems, `WaitForMultipleObjects` for Windows). When the application receives an event notification that a TCAP message is available, it can receive the message using the non-blocking function `ss7_tcap_event_msg_get()`, process the message, and then continue to wait for further events.

To improve scalability, the application creates an event and then attaches transactions (or the ssap structure itself) to the event. Thus a single native event can be used for large numbers of transactions.

## 10.2.10 Encoding/Decoding ASN.1

As mentioned elsewhere, the API user will have to encode and decode ASN.1 data. The Aculab TCAP API gives the user freedom to choose from a number of ways of doing this, as described below.

### 10.2.10.1 Aculab ASN.1 encode/decode functions

The Aculab TCAP API includes interface functions that may be used for encoding and decoding ASN.1. The Aculab functions have been designed into the API in such a way as to optimise code paths, maximise performance, and have the benefit of full support from Aculab. In the absence of other constraints, Aculab would encourage customers to use these functions in preference to the alternative methods described in sections 10.2.10.2 and 10.2.10.3.

In order to use the Aculab ASN.1 functions, the user must refer to the ASN.1 specification of the protocol that he is implementing, and call the appropriate functions to encode/decode the data accordingly. Sometimes, this task is much simpler than it may appear. For example, the GSM MAP protocol specification is many hundreds of pages long, but a typical application will use only a tiny subset of that specification. Once that subset is identified, the amount of relevant ASN.1 is often reduced to a few pages.

A limitation of the Aculab ASN.1 decode functions is that they will simply attempt to decode whatever is requested, in the order they are called. It is the application's responsibility to ensure that all expected ASN.1 parameters are recovered, and to ensure there are no "extra" parameters beyond those which are permitted.

### 10.2.10.2 Third party ASN.1 'toolkits'

There are a number of commercially available ASN.1 toolkits, which usually include a so-called ASN.1 'compiler'. These are designed to take a textual ASN.1 specification as input, and generate 'C' or 'C++' source code, with interfaces to encode and decode the data according to the specification.

ASN.1 toolkits are probably the most commonly used options for software development using ASN.1. Their main advantage is that in theory the user is isolated from, and doesn't need to understand, the actual protocol specification.

In practice, ASN.1 compilers do suffer from some drawbacks. The generated 'C' or 'C++' interfaces can be hard to understand and difficult to use, and the generated code for encoding and decoding the ASN.1 is often inefficient. This can present a significant performance impact, and it can be difficult to contrive test strategies that prove the generated code actually works correctly under all conditions. Theses issues are further exacerbated by the fact that ASN.1 compilers will generally operate on an entire protocol specification, generating much more code than the application actually needs.

Another drawback is that the authors will have published the ASN.1 specification in human-readable form. The ASN.1 compilers, of course, require machine-readable input and it is quite common to have to manually edit ('tweak') the specification text in places to ensure 100% format compatibility with the compiler's expectations.

#### 10.2.10.3 Third party application-part APIs

It is possible to purchase application APIs for many TCAP application protocols such as INAP, MAP, CAMEL etc. These APIs provide a documented and supported 'C' or 'C++' interface that an application writer can use to encode and decode his ASN.1 data.

If using such an API, there should be no need to refer to the ASN.1 specification itself. A further advantage is that the API vendor may be able to offer assistance and consultancy with writing an application, thus reducing development timescales.

On the downside, third party APIs can be expensive to purchase or license, and there can be additional costs for support. Frequently, the APIs are generated using an ASN.1 compiler as described in section 10.2.10.2 and so they suffer from many of the same drawbacks, sometimes being over complex and inefficient.

# 11 Sample applications

This section describes a sample network and application that allow a “hands-on” approach to learning how to install and use Aculab SS7. If you just wish to install, configure, and test the sample network, then you can do so. You do not need to install and build the actual applications if you only intend to test the configuration.

The sample applications described in this section are intended to illustrate some of the first steps to get you started with Aculab SS7. As such, they have been deliberately simplified and do not necessarily handle all the circumstances and events that may arise in a real-world scenario.

To reduce complexity, and to avoid the need for additional hardware, these samples do not involve any “voice” traffic. The basic interactions with the Aculab Switch API are illustrated by data-filling the “voice” channel with a constant pattern. A real-world application would also have to deal with the required switching of the voice/bearer channels using the Aculab Switch API.

The TCAP portion of the sample applications illustrates the use of Aculab’s Distributed TCAP product by way of a small set of ITU INAP operations.

Source code and TCAP configuration files for the samples are available for download from the Aculab AIT server.

**Note** **Aculab may continue to improve and develop the sample applications after publication of this document. Therefore you may find that the version downloaded from the website differs in detail from that described here.**

## 11.1 What the samples do

The samples consist of three separate programs that, in turn, illustrate use of the ISUP API in isolation, use of the TCAP API in isolation and use of the two APIs together.

### 11.1.1 Call generator

This is a command-line tool for generating calls and uses only the ISUP API. This program acts as a simple ISUP local exchange with no network “intelligence”, and provides the input stimulus for the other sample programs. The timeslot, called number, and (optionally) the data fill pattern for the forward speech circuit, are command-line arguments.

The sequence of event notifications relating to the call, whether it is rejected or progresses to the “alerting” and “connected” phase, is provided as textual output. If the data pattern sampled from the backward speech circuit is not as expected, this is also reported.

### 11.1.2 “SSP” Call handler

This program simulates an ISUP exchange with intelligent switching capability in keeping with the SSP model described in section 2.4.1. It uses both the ISUP API and the TCAP API.

The program asks a remote instance of the SCP sample program whether each call should be allowed to proceed and accepts only those calls that the SCP permits. Calls not permitted by the SCP are rejected.

### 11.1.3 “SCP” Call validation

This program uses only the TCAP API, and illustrates an Intelligent Network control node in keeping with the SCP model described in section 2.4.2. It waits for INAP `initialDP` operations from the SSP sample program to validate a call’s destination address. The SCP responds to the SSP with `connect` or `releaseCall`, instructing it to either accept or reject the call respectively, depending upon the digits in the called party number.

The following two diagrams illustrate the message exchanges between the three sample programs:

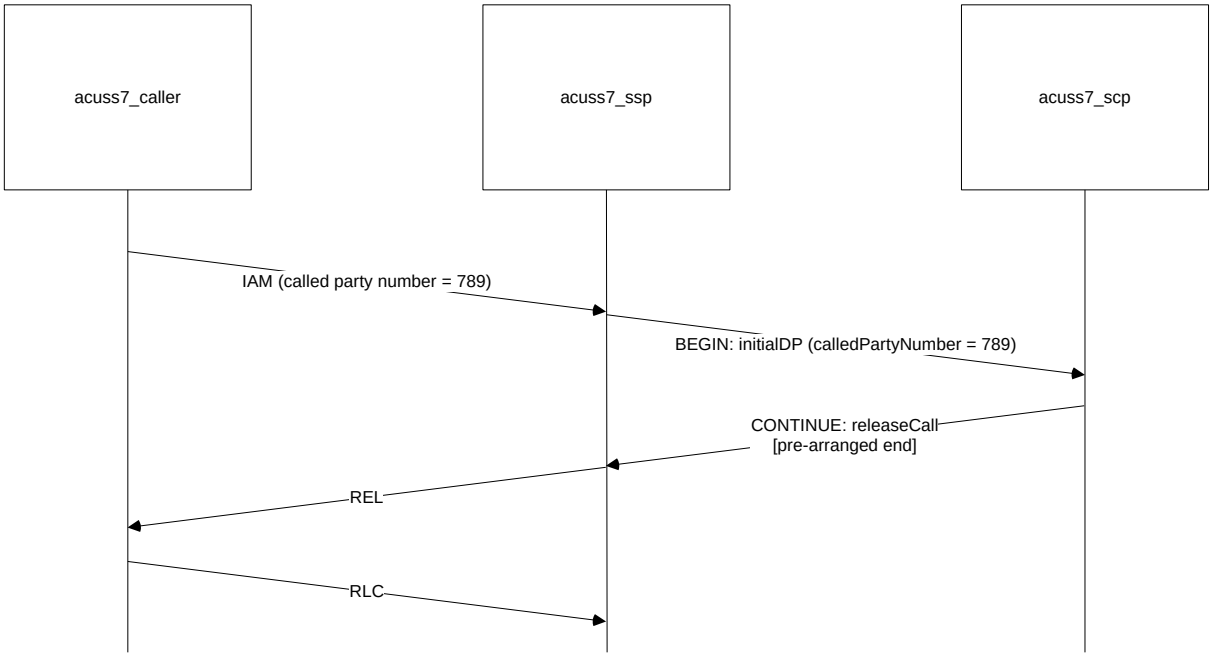


**Figure 12 - Unsuccessful call attempt**

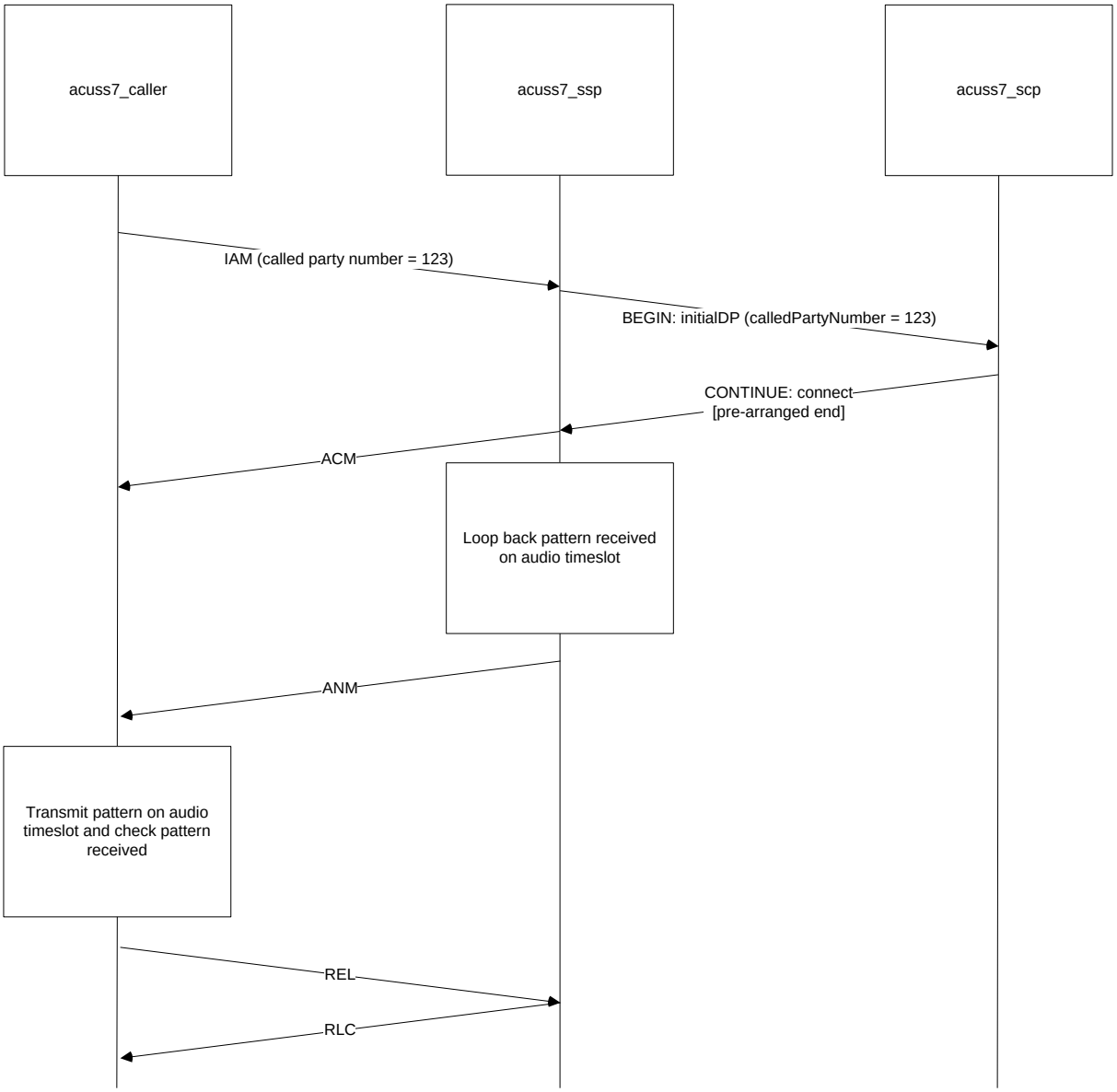


**Figure 13 - Successful call attempt**

## 11.2 The sample network

The sample applications are based upon a simple network consisting of three signalling points as illustrated in figure 12. The entire network is simulated using a 4 port Aculab card installed in a single chassis, as illustrated in figure 13.

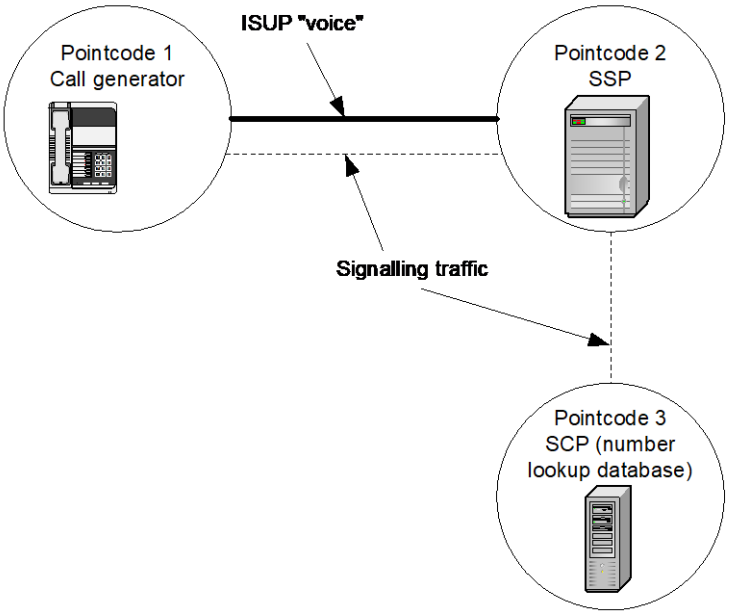


**Figure 14 - Network diagram for sample applications**

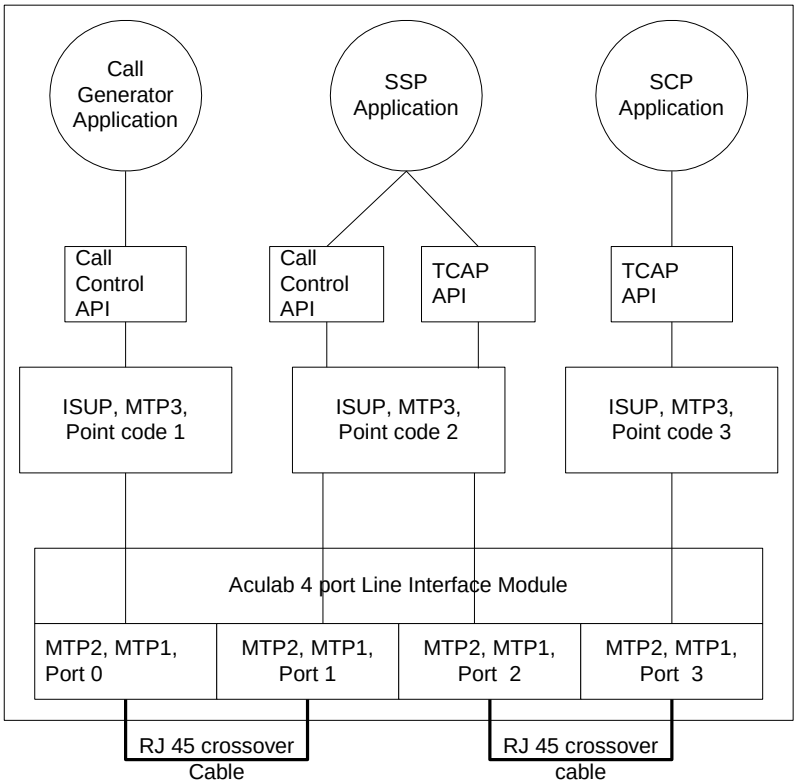


**Figure 15 - Single-chassis implementation of the sample network using a 4-port PM module and two crossover cables (simpler networks could be tested using just two ports and a single cable)**

## 11.3 Installation and configuration

### 11.3.1 Hardware and software installation

In order to use the sample applications, at least one Aculab digital access card must be installed in the test chassis. The sample assumes four network ports are present, although simpler networks could be simulated using just two network ports. The exact procedure depends upon which hardware modules are purchased; please refer to the corresponding card installation guide for detailed instructions. In addition, the software for SS7, Distributed TCAP and call control components must be installed using the Aculab Installer Tool, as described in the ***Aculab installation tool - FTP downloads utility guide***.

#### 11.3.1.1 Stack configuration file

A stack configuration file suitable for the sample applications can be found in the file `ss7.cfg` that accompanies the sample source code.
The stack must be started using "`ss7maint start`".

#### 11.3.1.2 Firmware configuration

The firmware configuration is provided when the firmware is downloaded to the card, which can be performed by several alternative methods. The Interactive Aculab Configuration Tool (ACT) or the `fwdspldr` command-line tool may be used, or it may be downloaded using the programmatic interface of the call control API. No matter which method you choose for firmware download, the parameters for the four-firmware ports should be similar to the following:

Port 0: `-cOPC1 -cDPC2 -cTS16 -cSLC0 -cCIC200`
Port 1: `-cOPC2 -cDPC1 -cTS16 -cSLC0 -cCIC200`
Port 2: `-cOPC2 -cDPC3 -cTS16 -cSLC0`
Port 3: `-cOPC3 -cDPC2 -cTS16 -cSLC0`

You will also need to configure a suitable clocking mode. This can be done using the ACT, or using the `swcmd` command-line utility. Clocking modes are described in more detail in the Aculab ***Switch API guide.***

### 11.3.2 Verifying network connectivity

Once you have completed the configuration, started the SS7 protocol stack, and all firmware downloads are complete, you can perform some checks to confirm network connectivity. This can be done before any attempts to build or run the sample code. The tool used is `ss7maint`, which is fully described in the Aculab ***SS7 Installation and administration guide***. Please refer to that document when performing the tasks below, as you may want to explore other useful options such as "`-v`" (verbose) that requests more detailed output, and "`-hi200`" which provides a continuous display updated every 200 milliseconds.

#### 11.3.2.1 MTP2 checks

Use "`ss7maint linkstatus -2`" to confirm the links are in service at level 2.

#### 11.3.2.2 MTP3 checks

Use "`ss7maint linkstatus -3`" to confirm the links are in service at level 3. Use "`ss7maint mtp3status`" with additional options to confirm specific accessibility and status of signalling routes and signalling points.

#### 11.3.2.3 ISUP checks

Use "`ss7maint isupstatus`" with various options to confirm correct ISUP circuit availability.

## 11.4 Running the samples

### 11.4.1 Compiling and linking

You will need to compile the source code and link it with the call control and/or Distributed TCAP libraries. The exact procedure for this depends upon which operating system you are using, as well as your own personal preferences.

### 11.4.2 Running the SSP

Ensure that the application's Distributed TCAP configuration file `acuss7_ssp.cfg` is in the current directory when the application is started. The application accepts no command line arguments.

While running, the application reports call events and the call handle for each event. The representation of the call handles is not significant and they only serve to distinguish calls in the output. E.g.:

```
$ acuss7_ssp
ssp: waiting for connection to SCCP.
ssp: Connected to SCCP.
ssp: INCOMING_CALL_DET 00A41801
ssp: calledPartyNumber: 84 00 87 f9
ssp: sending initialDP.
ssp: response for call 0040AE60
ssp: releaseCall.
ssp: IDLE 00A41801
ssp: INCOMING_CALL_DET 00A41C01
ssp: calledPartyNumber: 84 00 21 f3
ssp: sending initialDP.
ssp: response for call 0040AE74
ssp: connect.
ssp: CALL_CONNECTED 00A41C01
ssp: REMOTE_DISCONNECT 00A41C01
ssp: IDLE 00A41C01
```

### 11.4.3 Running the SCP

Ensure that the application's TCAP configuration file `acuss7_scp.cfg` is in the current directory when the application is started. The application accepts no command line arguments.

While running, the application reports each destination address it is requested to validate and whether that address has "passed" or "failed" the validation. E.g.:

```
$ acuss7_scp
scp: entering event loop.
scp: host a (127.0.0.1:8256) in service

scp: initialDP.
scp: serviceKey: 1234.
scp: calledPartyNumber: 84 00 87 f9
scp: fail for calledPartyNumber: 84 00 87 f9
scp: initialDP.
scp: serviceKey: 1234.
scp: calledPartyNumber: 84 00 21 f3
scp: pass for calledPartyNumber: 84 00 21 f3
```

### 11.4.4 Generating calls

When the application is started, the user supplies the timeslot on which the call is to be made and the destination address of the call. Optionally, a data pattern may be supplied which will be fed into the speech path instead of conventional voice traffic. If this argument is omitted, the "idle" pattern (silence) is used.

While running, the application reports events for the call as it progresses. E.g.:

```
$ acuss7_caller 1 789 0xa5
caller: WAIT_FOR_OUTGOING
caller: DETAILS
caller: OUTGOING_PROCEEDING
caller: REMOTE_DISCONNECT
caller: IDLE

$ acuss7_caller 1 123 0xa5
caller: WAIT_FOR_OUTGOING
caller: DETAILS
```

```
caller: OUTGOING_PROCEEDING
caller: CALL_CONNECTED
caller: IDLE
```

# Appendix A: Code examples

This section contains further examples of code extracts that are intended to supplement the information provided elsewhere in this document and in the API guides.

## A.1 Example 1 – Flexible ISUP parameter edit on a supported message type

An application can use Flexible ISUP to override parameters that are normally provided by the ISUP driver, or to add standard parameters that the driver does not normally include. In this example, the Backward Call Indicators parameter in an Address Complete Message (ACM) is overridden with an application-supplied value. Backward Call Indicators is a standard parameter in an ACM, so no codec extensions are necessary.

```
/* Flexible ISUP Example 1
 * Code fragment to demonstrate use of Flexible ISUP to edit an existing
 * parameter.  No codec extensions are needed to modify standard White
 * Book ISUP parameters. */
FEATURE_DETAIL_XPARMS fdxp;
PROCEEDING_XPARMS procxp;

INIT_ACU_CL_STRUCT(&fdxp);

/* Queue up a replacement Backward Call Indicators of 0x1016
 * i.e bits B, C, E and M are set (ref: Table21/Q.763). */
fdxp.handle = handle;
fdxp.feature_type = FEATURE_RAW_MSG;
fdxp.message_control = CONTROL_NEXT_CC_MESSAGE;
fdxp.feature.raw_msg.length = 4;
fdxp.feature.raw_msg.data[0] = 0x11; /* Backward Call Indicators */
fdxp.feature.raw_msg.data[1] = 0x02; /* Length in octets */
fdxp.feature.raw_msg.data[2] = 0x16; /* bits A-H */
fdxp.feature.raw_msg.data[3] = 0x10; /* bits I-P */

call_feature_send(&fdxp);

INIT_ACU_CL_STRUCT(&procxp);
procxp.handle = handle;

/* an ACM generated by call_proceeding() will contain the BCI value
 * 0x1016 */
call_proceeding(&procxp);
/* End Flexible ISUP Example 1 */
```

## A.2 Example 2 – Supporting a nationally significant message

Flexible ISUP can be used to implement support for nationally significant messages, such as the Charge Information (CRG) message. The format of a CRG message is not internationally agreed, so the ISUP driver must be configured to recognise a national CRG message. One possible coding of a CRG message is described by the following codec extension sample:

```
[ISUPCodec]
   name = national_charge
   # CRG
   [Msg]
        code = 0x31
        prm = 0xff, O, 3 # Charge band number
        prm = 0xfe, O, 3 # Number of charging units
        prm = 0x38, O, 3 # Message compatibility information
        prm = 0x39, O, 4 # Parameter compatibility information
   [EndMsg]
[EndISUPCodec]
```

The following application code illustrates methods for the transmission of the above CRG message:

```
/* Flexible ISUP Example 2
 * Code fragment to demonstrate use of Flexible ISUP.
 * Send a custom Charge Information message, containing a Charge Band Number. */
ACU_ERR
send_charge_band(ACU_CALL_HANDLE handle, ACUSS7_OCTET charge_band)
{
    FEATURE_DETAIL_XPARMS fdxp;
    INIT_ACU_CL_STRUCT(&fdxp);

    fdxp.handle = handle;
    fdxp.feature_type = FEATURE_RAW_MSG;
    fdxp.message_control = CONTROL_DEFAULT;     /* Whole message specified */
    fdxp.feature.raw_msg.length = 8;

    fdxp.feature.raw_msg.data[0] = 0x31;        /* Charge Information message */

    fdxp.feature.raw_msg.data[1] = 0x39;        /* Parameter Compatibility Info */
    fdxp.feature.raw_msg.data[2] = 0x02;        /* Length */
    fdxp.feature.raw_msg.data[3] = 0xff;        /* Charge Band Number */
    fdxp.feature.raw_msg.data[4] = 0x86;        /* Release Call / Send Notification */

    fdxp.feature.raw_msg.data[5] = 0xff;        /* Charge Band Number */
    fdxp.feature.raw_msg.data[6] = 0x01;        /* Length */
    fdxp.feature.raw_msg.data[7] = charge_band;

    return call_feature_send(&fdxp);
}

/* Send a custom Charge Information message, containing a Number Of Charging
 * Units. */
ACU_ERR
send_charging_units(ACU_CALL_HANDLE handle, ACUSS7_OCTET num_charge_units)
{
    FEATURE_DETAIL_XPARMS fdxp;
    INIT_ACU_CL_STRUCT(&fdxp);

    fdxp.handle = handle;
    fdxp.feature_type = FEATURE_RAW_MSG;
    fdxp.message_control = CONTROL_DEFAULT;     /* Whole message specified */
    fdxp.feature.raw_msg.length = 8;

    fdxp.feature.raw_msg.data[0] = 0x31;        /* Charge Information message */

    fdxp.feature.raw_msg.data[1] = 0x39;        /* Parameter Compatibility Info */
    fdxp.feature.raw_msg.data[2] = 0x02;        /* Length */
    fdxp.feature.raw_msg.data[3] = 0xfe;        /* Number Of Charging Units */
    fdxp.feature.raw_msg.data[4] = 0x86;        /* Release Call / Send Notification */

    fdxp.feature.raw_msg.data[5] = 0xfe;        /* Number Of Charging Units */
    fdxp.feature.raw_msg.data[6] = 0x01;        /* Length */
    fdxp.feature.raw_msg.data[7] = num_charge_units;

    return call_feature_send(&fdxp);
}
/* End Flexible ISUP Example 2 */
```

## A.3 Example 3 – Enabling, receiving, and handling of EV_EXT_RAW_MSG

Flexible ISUP provides the extended event `EV_EXT_RAW_MSG` to indicate to an application that a network message has arrived. The ISUP driver holds a queue of messages on a per-call basis. An `EV_EXT_RAW_MSG` event is only generated when a call's queue becomes non-empty, so it is very important that an application reads all available messages after receipt of `EV_EXT_RAW_MSG`. In this example, an application uses Flexible ISUP to receive CRG messages and perform some simple processing.

### A.3.1 Enabling EV_EXT_RAW_MSG

An application must request `EV_EXT_RAW_MSG` events by setting the `CNF_RAW_MSG` bit in the cnf word:

```
/* Flexible ISUP Example 3.1
 * Code fragment to demonstrate use of Flexible ISUP.
 * Enable receipt of raw messages */
OUT_XPARMS oxp;

if (strlen(dest) >= sizeof oxp.destination_addr) {
    printf("destination address \"%s\" too long\n", dest);
    return EXIT_FAILURE;
}

INIT_ACU_CL_STRUCT(&oxp);
oxp.net = opp.port_id;
oxp.ts = ts;
oxp.cnf = CNF_REM_DISC | CNF_RAW_MSG;
oxp.sending_complete = 1;
strcpy(oxp.destination_addr, dest);
res = call_openout(&oxp);
/* End Flexible ISUP Example 3.1 */
```

### A.3.2 Receiving EV_EXT_RAW_MSG and processing raw messages

This sample illustrates detection and collection of raw messages. The ISUP helper function `isup_get_parameter()` is used to locate a Number Of Charging Units parameter in a received CRG message.

```
/* Flexible ISUP Example 3.2
 * Code fragment to demonstrate use of Flexible ISUP.
 * Handle received EV_EXT_RAW_MSG events */
STATE_XPARMS sxp;
FEATURE_DETAIL_XPARMS fdxp;
ACU_ERR res;
ACU_UCHAR *prmp;

for (;;) {
    /* Collect an event ... */
    INIT_ACU_CL_STRUCT(&sxp);
    sxp.timeout = 30000;
    res = call_event(&sxp);
    if (res)
        bail_out("call_event", res);

    if (sxp.handle != handle)
        return EXIT_FAILURE;

    switch (sxp.state) {
        /* :
           : Handle other events here
           : */
    case EV_EXTENDED:
        switch (sxp.extended_state) {
        case EV_EXT_RAW_MSG:
            printf("EXT_RAW_MSG");
            /* loop to ensure all raw messages for this call are read from
             * the driver. The application won't receive a subsequent
             * EV_EXT_RAW_MSG event for this call otherwise. */
            for (;;) {
                INIT_ACU_CL_STRUCT(&fdxp);
```

```
                    fdxp.handle = handle;
                    fdxp.feature_type = FEATURE_RAW_MSG;
                    res = call_feature_details(&fdxp);
                    if (res)
                        bail_out("call_feature_details", res);

                    if (!(fdxp.feature_type & FEATURE_RAW_MSG))
                        break; /* call's raw message queue is now empty */

                    printf("RAW_MSG:0x%02x seq:%u\n", fdxp.feature.raw_msg.data[0],
                           (unsigned int)fdxp.feature.raw_msg.raw_msg_seq);

                    /* Only Charge Information messages (message type 0x31) will
                     * be scrutinised */
                    if (fdxp.feature.raw_msg.data[0] != 0x31)
                        continue;

                    /* Look for Number Of Charging Units parameter (0xfe) */
                    prmp = isup_get_parameter(&fdxp.feature.raw_msg, 0xfe, NULL);
                    /* prmp will point to Type octet, or NULL if not found.
                     * prmp[0] = Type
                     * prmp[1] = Length
                     * prmp[2] = First octet of parameter data */
                    if (prmp)
                        printf("Charge: %d units\n", prmp[2]);

                };
                break;

            default:
                printf("unexpected extended event: 0x%lX\n", sxp.extended_state);
                return EXIT_FAILURE;
            }
            break;

        default:
            printf("unexpected event: 0x%lX\n", sxp.state);
            return EXIT_FAILURE;
        }
    }
    /* End Flexible ISUP Example 3.2 */
```

## A.4 Example signalling trace

The following ISUP signalling trace was captured and formatted using the `ss7maint` tool, and shows the effects of some of the code examples in Appendix A. Note the overridden Backward Call Indicators value in the ACM and the presence of CRG messages, coded in accordance with the `national_charge` codec extension of example A.2.

```
****************************** Protocol decode ******************************
* Line 17
*
[13262 15:22:58.086] MTP2 2020-7070 slc 0 TX card 111(0) ts 16 55s335

   00:  c3 be 14 85  9e 1b f9 81  c8 00 01 00  00 00 0a 00
   10:  02 00 04 04  00 21 f3

BSN/BIB=67/1, FSN/FIB=62/1, Type=MSU, LI=20, Actual length=23
Network Indicator=2 (National), Service indicator=0x5, Spare/priority=0
Decode=ITU, ISUP

LABEL: OPC=2020, DPC=7070, SLS=8

CIC=200,Type=IAM (Initial Address)
Nature of connection indicators=0x00
Forward call indicators=0x0000
Calling party category=0x0a
Transmission medium requirement=0x00
Called party number:
   0x04, 0x00, "123f"
Pointer to optional part is zero

****************************** Protocol decode ******************************
* Line 90
*
[13262 15:22:58.119] MTP2 2020-7070 slc 0 RX card 111(0) ts 16 55s365

   00:  bf c5 0b 85  e4 87 e7 86  c8 00 06 16  10 00

BSN/BIB=63/1, FSN/FIB=69/1, Type=MSU, LI=11, Actual length=14
Network Indicator=2 (National), Service indicator=0x5, Spare/priority=0
Decode=ITU, ISUP

LABEL: OPC=7070, DPC=2020, SLS=8

CIC=200,Type=ACM (Address Complete)
Backwards call indicators=0x1016
Pointer to optional part is zero

****************************** Protocol decode ******************************
* Line 102
*
[13262 15:22:58.122] MTP2 2020-7070 slc 0 RX card 111(0) ts 16 55s370

   00:  bf c6 11 85  e4 87 e7 86  c8 00 31 01  ff 01 01 39
   10:  02 ff 86 00

BSN/BIB=63/1, FSN/FIB=70/1, Type=MSU, LI=17, Actual length=20
Network Indicator=2 (National), Service indicator=0x5, Spare/priority=0
Decode=ITU, ISUP

LABEL: OPC=7070, DPC=2020, SLS=8

CIC=200,Type=CRG (Charge)
DECODE WARNING: No message decode available.

****************************** Protocol decode ******************************
* Line 117
*
[13262 15:22:58.125] MTP2 2020-7070 slc 0 RX card 111(0) ts 16 55s375

   00:  bf c7 0e 85  e4 87 e7 86  c8 00 09 01  11 02 00 10
   10:  00

BSN/BIB=63/1, FSN/FIB=71/1, Type=MSU, LI=14, Actual length=17
Network Indicator=2 (National), Service indicator=0x5, Spare/priority=0
Decode=ITU, ISUP

LABEL: OPC=7070, DPC=2020, SLS=8
```

```
CIC=200,Type=ANM (Answer)
Optional parameter: 0x11 (Backward call inds):
    00:  00 10
Optional parameter: 0x00 (End of optional parameters):

****************************** Protocol decode ******************************
* Line 130
*
[13262 15:22:58.128] MTP2 2020-7070 slc 0 RX card 111(0) ts 16 55s375

   00:  bf c8 11 85  e4 87 e7 86  c8 00 31 01  fe 01 0a 39
   10:  02 fe 86 00

BSN/BIB=63/1, FSN/FIB=72/1, Type=MSU, LI=17, Actual length=20
Network Indicator=2 (National), Service indicator=0x5, Spare/priority=0
Decode=ITU, ISUP

LABEL: OPC=7070, DPC=2020, SLS=8

CIC=200,Type=CRG (Charge)
DECODE WARNING: No message decode available.

****************************** Protocol decode ******************************
* Line 145
*
[13262 15:23:00.090] MTP2 2020-7070 slc 0 TX card 111(0) ts 16 57s340

   00:  c8 c0 0d 85  9e 1b f9 81  c8 00 0c 02  00 02 82 90

BSN/BIB=72/1, FSN/FIB=64/1, Type=MSU, LI=13, Actual length=16
Network Indicator=2 (National), Service indicator=0x5, Spare/priority=0
Decode=ITU, ISUP

LABEL: OPC=2020, DPC=7070, SLS=8

CIC=200,Type=REL (Release)
Cause:
   00:  82 90
Pointer to optional part is zero

****************************** Protocol decode ******************************
* Line 173
*
[13262 15:23:00.101] MTP2 2020-7070 slc 0 RX card 111(0) ts 16 57s350

   00:  c0 c9 09 85  e4 87 e7 86  c8 00 10 00

BSN/BIB=64/1, FSN/FIB=73/1, Type=MSU, LI=9, Actual length=12
Network Indicator=2 (National), Service indicator=0x5, Spare/priority=0
Decode=ITU, ISUP

LABEL: OPC=7070, DPC=2020, SLS=8

CIC=200,Type=RLC (Release Complete)
Pointer to optional part is zero
```

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