aculab

ACULAB.COM

Aculab SS7

Signalling monitor user's guide and API reference

MAN1204 Revision 6.16.1

Document Revision

Rev	Date	By	Detail
6.7.0	05.10.06	WM	First issue with SS7 Version 6.7 software
6.7.1	16.10.06	DJL	Update following editorial reviews, typos only
6.7.2	14.01.07	WM	Security key configuration
6.8.3	16.03.07	DSL	Decode of configure_link responses
6.10.1	11.09.08	WM	Clarified description of line taps. Remove hyperlinks to cross-referenced documents.
6.10.3	30.10.08	DSL	General release copy
6.11.0	17.09.10	WM	Added MTP2 API, fonts updated
6.11.11	01.11.11	DSL	Refer to Prosody X V3 cards as well as PMXC modules.
6.12.2	28.06.13	DSL	Add `copy_msg()` and `free_msg()` functions, clarifications and corrections.
6.13.0	24.10.14	DSL	Minor corrections and editorial changes.
6.14.0	16.09.16	DSL	Align with current cards.
6.15.1	31.08.18	DSL	Add HSL, tracing and new PT monitor
6.16.0	16.02.22	DSL	Add Prosody X Evo information.
6.16.1	13.02.22	DSL	Update title page

CONTENTS

1 Introduction

This manual describes the Aculab SS7 signalling monitor, which allows applications to use an Aculab prosody card to ‘eavesdrop’ on SS7 signalling traffic between SS7 signalling points (SP). It includes a description of the monitor itself with a full API.

In order to use the signalling monitor, you will also need to install, configure, and use, the specification, and explains how it relates to other Aculab products and APIs.

In addition to the monitor API, a series of additional API calls are provided that allows the monitor library to activate full ‘active’ MTP2 functionality or to send and receive raw HDLC frames.

In order to use this software, you will also need to install, configure, and use, the underlying core Aculab telephony software and Aculab SS7 server products as described in the SS7 installation and administration guide, the SS7 developer’s guide, and in the card installation guide and related documentation for the ‘core’ V6 telephony software. You may therefore find it useful to have these manuals available while reading this one.

Applications that use the SS7 signalling monitor may also be required to monitor bearer traffic, such as subscriber conversations. For E1/T1 traffic, this can be achieved using the Aculab Prosody (speech) APIs, which can co-exist with the SS7 signalling monitor. Some general advice and guidance about monitoring of bearer traffic is included in this document, but the Prosody API is unaffected by the presence of the SS7 signalling monitor so reference should be made to the appropriate Prosody documentation if it is needed.

The API specification is supplemented, in section Appendix B: with a working example application. This demonstrates a simple monitor which is triggered by calls to the number starting `123`. Each call results in a call record being written to `stdout` identifying start, connection, and end times, along with details of called and calling numbers.

2 Product overview

2.1 Overall structure

Figure 1 illustrates, in simplified form, the major components of the signalling monitor and shows how it relates to the SS7 signalling software (ISUP & TCAP) APIs.

It can be seen from Figure 1 that the SS7 signalling software (SS7 Server) and the core Aculab telephony software always need to be installed when a monitoring application is in use, since they contain software components that are needed by both monitoring and signalling applications. The monitoring software is optional and only needs to be installed if a monitoring application is going to be run. The card hardware and software is the same, regardless of whether it is to be used for signalling, monitoring, or both.

Monitor applications will also need to use some of the functions of the Aculab resource manager API, the switch API and, in some cases, the Prosody API. These are not shown in the diagram, but would be installed along with the SS7 signalling software and core telephony software, so that they are available to monitoring applications when required.

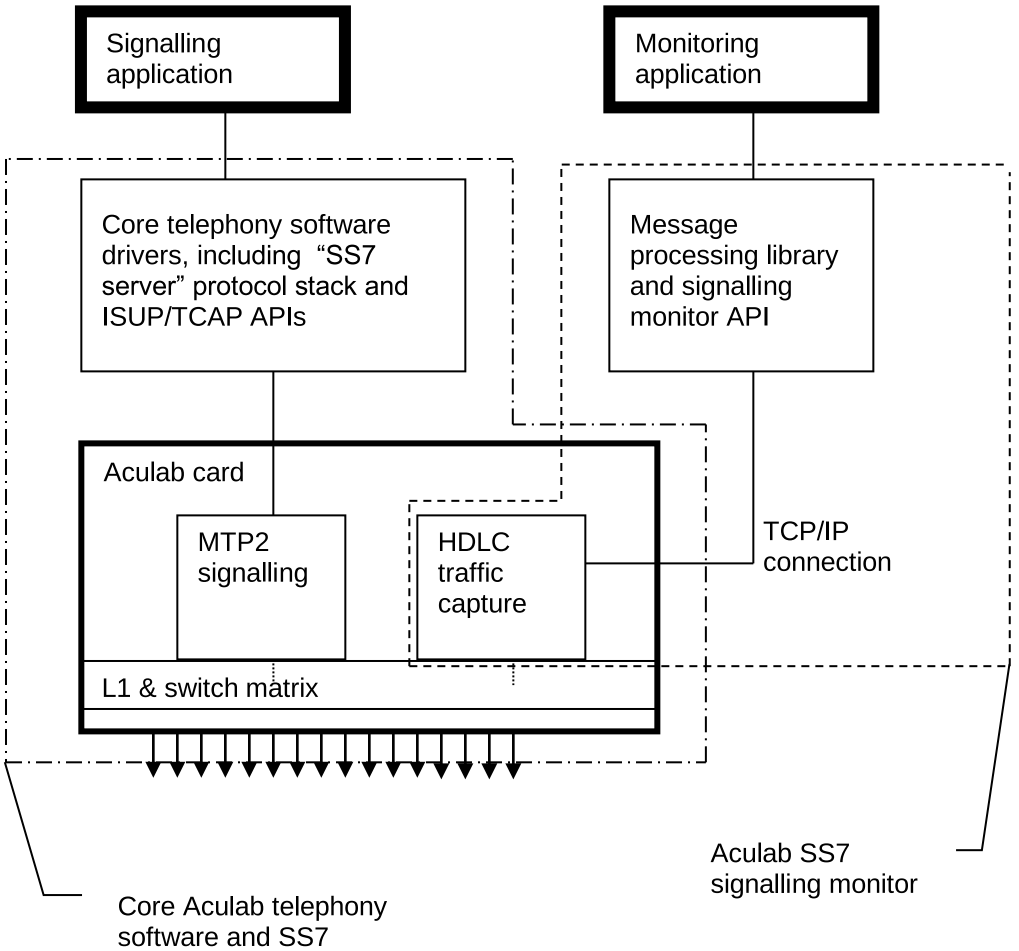

Figure 1: Major system components for monitoring and signalling

2.2 Signalling traffic capture

The SS7 MTP2 on Prosody cards runs on dedicated hardware that is connected to the card's TDM switch. Information is passed to/from MTP3 and monitor applications using TCP connections. The on-card receive processing has three basic stages:

1) HDLC framing
2) MTP2 state processing
3) TCP/IP message encoding

When configured as a monitor the MTP2 state processing is bypassed and received HDLC frames are passed directly to the application.

An idle SS7 MTP2 link is filled with continuous short HDLC frames (LSSU or FISU). Since these are not normally of interest to a monitoring application, short receive frames are normally filtered out. The application can request that a single shorter frame be reported: e.g. to request that the first LSSU be reported (typically indicating link failure) or to request that a FISU be reported to verify the link is still active.

The HDLC transmit is always active and will idle sending hdlc flags on monitor links. The transmit data is not normally connected to an external timeslot unless the 'monitor' is being used as a general hdlc controller.

2.3 Message processing library and API

The SS7 signalling monitor API allows an application to establish a TCP/IP connection to the card responsible for traffic capture, and then to retrieve monitored traffic by reading data from that connection.

The data on the TCP/IP connection between the card and application is encapsulated in a proprietary and unpublished Aculab protocol (based heavily on SIGTRAN M2UA). The monitor API library provides the functions to connect to the card and to encode and decode the TCP/IP stream.

In addition to providing an interface for applications to access the raw HDLC data, the monitor library can be configured to intercept the data with various filtering options, for example, to suppress messages for certain SS7 user parts. It may also be used to perform either or both an MTP3 and an ISUP message decode. If the ISUP decode is in use, the library maintains context information which the application may (but does not have to) use to keep track of individual ISUP telephone calls.

The monitor could be used to receive (and can send) HDLC frames for any protocol provided that they are not longer than SS7 frames and use CRC16.

Full details of the API are provided later in this document (see sections 4 to 9).

2.4 Traffic capture modes

To capture the signalling traffic mtp2/hdlc engines (with internal TDM timeslots to the TDM switch) are allocated and assigned for monitoring. These are then connected, using the TDM switch matrix, to traffic sources containing the actual SS7 signalling. Two mtp2/hdlc blocks are required for each SS7 signalling link being monitored, each monitoring data in one direction only, i.e. `tx` or `rx`.

It is possible to configure three alternative modes of capture traffic, as described below. See also section Appendix A:, which contains further details of these three modes and provides example code snippets showing how to configure and initialise each mode.

Passive interception

In this mode, external line taps (not supplied by Aculab) are used to extract the traffic captured from E1/T1 trunks containing SS7 signalling links. The line taps may be truly passive, or they may be powered, as described below.

Purely passive taps typically use resistive-coupling or transformer-coupling, and present a weak signal to the monitor, at the expense of a small attenuation of the signal being monitored. The monitor has to be adjusted for the weak signal, please refer to the `-cRXMON` firmware configuration parameter in the parameter in the SS7 installation and administration guide.

Powered taps require an external power source as they provide amplification. They present a high impedance to the circuit being monitored, with very little attenuation, but provide a regenerated signal at full strength for the monitor. They tend to be much more costly that purely passive taps.

Owing to the cost advantage, un-powered passive are preferred by most users. Your Aculab account manager may be able to advise on suppliers.

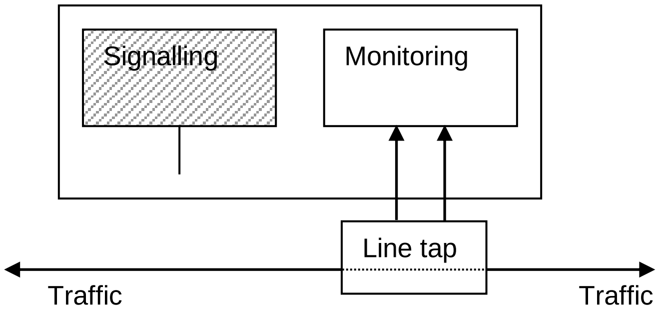

Figure 2: passive interception

Active interception (pass-through)

In this mode the traffic is captured from external E1/T1s containing signalling links by diverting them through a pair of TDM network ports that are connected in a bi-directional 'pass-through' mode to one another. Compared with the passive intercept configuration this has the drawback that the Aculab card becomes an active participant, and hence a potential point of failure in the physical (Layer 1) connection between the monitored SPs. It does however offer cost savings, as no external line taps are required.

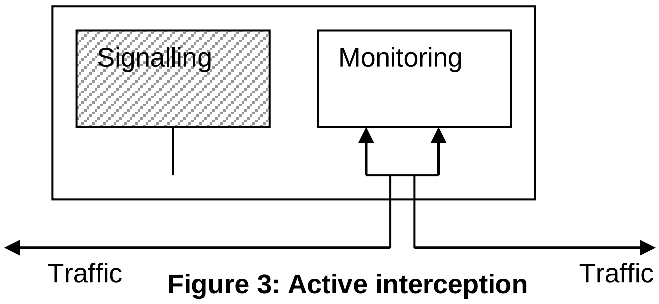

Figure 3: Active interception

Local traffic monitor

In this mode, as well as running a monitoring application, some other (or the same) application is performing ISUP or TCAP signalling to a remote SP. The monitoring application is processing the data to/from the signalling application's own signalling link(s).

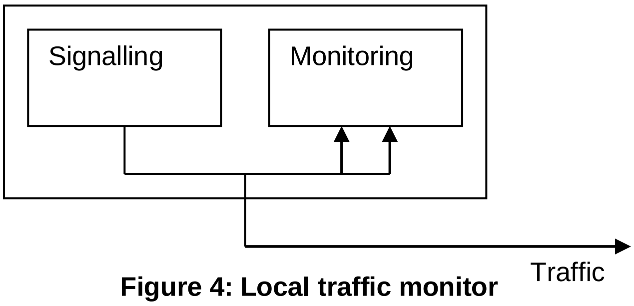

Figure 4: Local traffic monitor

All of the modes described above can coexist. A single card can be performing a combination of signalling, local monitoring, and active & passive interception concurrently on different network ports.

Note **For all monitor modes, SS7 signalling messages between any two signalling points may take various routes. These routes may vary from one message to another, and may be different for each direction of traffic. To capture all traffic, customers deploying the signalling monitor need to identify all potential traffic routes and ensure that all relevant signalling links are monitored.**

2.5 Bearer traffic capture

Bearer traffic can be captured using the Aculab switch and Prosody APIs to access data on the bearer timeslots in the same way a signalling application would. Refer to the Aculab switch and Prosody API guides for details.

The Prosody T monitor (on Prosody X Evo) supports direct monitoring of bearer data. See section 2.6.3.

In order for a monitoring application to access bearer traffic based on the monitor API's ISUP message decoder; it will need to translate the ISUP circuit identification codes (CICs) into actual bearer channels (E1/T1 ports and timeslots). The mapping between ISUP CICs and physical bearer channels cannot be predicted, it is a bilateral agreement between the two ISUP switches. It will usually be necessary to request the CIC mapping details from the administrators of one (not both) of the switches that are being monitored.

Note **SS7 ISUP bearer traffic may take a different route, or may use a different physical medium, compared with the signalling traffic. So even when the Aculab signalling monitor is successfully capturing the signalling traffic, a different technology, and/or another monitoring location, may sometimes need to be deployed for bearer traffic capture.**

2.6 Supported cards and monitoring capacity

2.6.1 Signalling links

The number of signalling links that can be monitored depends on the number of timeslots of signalling data the card/chassis being used can process each TDM frame.

Card/Chassis Type	Timeslots	Note	
Prosody 1U enterprise	128	1	
Prosody X rev 3 card	64		
Prosody T card	64		
4 or 8 port HA chassis	64		Contains a Prosody X rev 3 card
16 port HA chassis	64+64	2	Contains 2 Prosody X rev 3 cards
4 or 8 port PX Evo chassis	64		Contains a Prosody T card
16 port PX Evo chassis	128	3	Contains 2 Prosody T cards
Prosody X Evo chassis	all	4	Using the Prosody T monitor

Notes:

1) At high message rates higher level processing may constrain throughput.
2) 64 on ports 0-7 and 64 on ports 8-15.
3) Any split between all 16 ports (transparent use of internal H.100 bus).
4) This uses the PX Evo chassis x86 motherboard cpu instead of on-card processors.

The number of signalling links that can be monitored is half the above because both the transmit and receive sides need to be processed.

MTP2 signalling and monitoring can be done at the same time; the limit applies to the combined number of timeslots.

The application must also be able to process all the monitored data in real time. If the application fails to keep up with the received data then receive data is silently discarded.

2.6.2 High speed signalling links

High speed signalling links (HSL) (see Q.703 Annex A) use unchannelised E1 (or T1) to get data rates of 2.0 (or 1.5) Mbit/sec.

The SS7 monitor supports HSL (not on Prosody 1U enterprise). Each timeslot of signalling counts towards the card's limit on the number of links. So a Prosody X rev 3 card can monitor the transmit and receive sides of a single HSL that uses all of an E1.

The monitor can decode messages that use extended sequence numbers.

Note **To avoid mis-sequenced monitored messages, the two data taps from an external trunk should be connected to the same card.**

Note **On a Prosody X rev 3 card the receive data must come indirectly from the H.100 bus not directly from the external trunk stream. This is handled by the library if acu_ss7mon_set_sw() is used to set the switch matrix connections.**

2.6.3 Prosody T monitor

On Prosody X Evo systems (containing Prosody T cards) an alternate monitor is available. Rather than use the card's MTP2 engines it processes the data using the x86 CPU on the Prosody X Evo chassis motherboard.

The faster cpu and more memory allow all the external timeslots be monitored (eg 8 HSL on a 16 port card).

Additionally raw TDM u-law/A-law data can be received through the monitor interface (as well as through TiNG).

The interface for monitoring traffic is the same as the other monitors. The only difference is setting up the switch matrix to link the external timeslot to the monitor software. The monitor is selected by configuring `ACU_SS7MON_CFG_PT_MONITOR`.

Internally the monitor uses the receive data path used by Prosody S. Prosody S can be used at the same time, but care must be taken to ensure the same timeslots are not being used by both sub-systems

This monitor has a higher latency than the normal monitor (around 20ms) and cannot send frames.

2.6.4 E1/T1 network ports

Each external E1/T1 trunk containing one or more signalling links that needs to be monitored, consume two of the card's network ports. The first port monitors data travelling in one direction, the second monitors traffic in the opposite direction. The remaining ports on the card can be used for signalling and/or or bearer, traffic. This yields the formula

```
Total network ports in use = (number of ports used for signalling or bearers
                                          + (2 x number of E1/T1s being monitored)).
```

The number of available network ports differs depending on the card version.

3 Application structure

3.1 Threading model

The API is not designed to be thread-safe; moreover, it explicitly mandates a single-threaded interface between the library and application. This avoids some overheads of data copying by allowing the application to have direct access to the library's internal data buffers. Appropriate 'rules' define the various times, based on sequences of API calls, that such accesses are allowed.

3.2 Libraries and header files

The Aculab SS7 signalling monitor consists of a single header file named `ss7monitor.h`, and a single library file as follows:

- For Linux, the library is a shared object named `libacu_ss7monitor.so`.
- For Windows the library is a dynamically linked library named `acuss7_monitor.dll`.

As explained in section 2.5, applications will also need to make use of the Aculab resource manager and switch APIs for establishing the TDM switch matrix connections. Some applications may also need to use the Aculab Prosody and/or call control APIs, in which case additional header files and libraries will need to be used. Refer to the product documentation for these additional APIs if they are required.

Note **For windows, the file** `ss7monitor.h` **contains** `#include <winsock2.h>`**. This can lead to namespace collisions with the older windows file** `winsock.h`**, which may be included in other header files such as** `windows.h`**. One way of resolving this problem is to ensure that the macro** `_WINSOCKAPI_` **is defined either in the application source code prior to any** `#include` **statements, or as a compiler directive where the macro instructs the pre-processor to exclude the contents of** `winsock.h`

4 API description

This section provides a brief description of the major features of the API. For a detailed API reference guide, refer to section 5.

4.1 Endpoints

An 'endpoint' is the primary means by which an application identifies itself to the monitor. Any application using the monitor needs to create at least one monitoring 'endpoint'. Each endpoint can only access a single card, but can monitor many different signalling links.

API functions are provided for creating, deleting, and configuring endpoints:

```
acu_ss7mon_create_endpoint()
acu_ss7mon_delete_endpoint()
acu_ss7mon_configure_endpoint()
```

4.2 Tracing

The monitor library can write trace information that includes the messages sent to and received from the HDLC/MTP2 engines to a local file. Tracing can be enabled by calling `acu_ss7mon_configure_endpoint(ACU_SS7MON_CFG_TRACE_FILENAME, "filename")`.

The trace file can be decoded by ss7maint decode.

4.3 TCP/IP connections

A TCP/IP connection is required between the monitor library endpoint and the card. Either of these API functions will setup the necessary connection:

```
acu_ss7mon_connect().
acu_ss7mon_connect_card_id().
```

The parameters required to establish the connection are usually obtained from the Aculab resource manager. The `acu_ss7mon_connect_card_id()` function requests and saves the required information before making the TCP/IP connection.

4.4 Event notification

The API includes an event-driven interface `acu_ss7mon_get_msg()` that allows an application endpoint to read data from the TCP/IP connection, blocking with a timeout in the monitor library until data is available. A function is also available for the application to obtain the native OS identifier for the TCP/IP socket, `acu_ss7mon_getsocket()`. This latter function is useful for event-driven applications that may need to wait for input from other sources as well data from the line interface card, especially given the single-threaded API constraints.

4.5 Link monitoring

In order to monitor a signalling link, the application endpoint requests the card software to allocate an HDLC/MTP2 engine. API functions are provided to start and stop monitoring, or to modify configuration, of individual links:

```
acu_ss7mon_monitor_link()
acu_ss7mon_unmonitor_link()
acu_ss7mon_configure_link()
```

The response to `acu_ss7mon_monitor_link()` contains an identifier (a small integer) that is used to reference the link in subsequent requests and indications.

4.6 Connecting TDM data to the monitor

Having allocated a monitor the application then needs to establish a TDM switch connection to the data being monitored. This is done using the Aculab switch API functions.

The application can either use the timeslot(s) allocated by the HDLC/MTP2 engine or direct the HDLC/MTP2 engine to use specific timeslot(s) on its stream to the card's TDM switch.

If the application used `acu_ss7mon_connect_card_id()` to establish the TCP/IP connection the acu_ss7mon_set_sw() function can be used to connect the allocated timeslot(s) to the external stream.

4.7 Receiving monitored data

An API function, `acu_ss7mon_get_msg()` is provided that returns a pointer to a data buffer if any relevant data is available, as well as any other information such as ISUP call identifiers (see below).

The buffers returned by `acu_ss7mon_get_msg()` are pointers to memory allocated within the library. The library only guarantees that the data remains valid until the next API call to `acu_ss7mon_get_msg()`, at which point the buffer may be freed or reused by the library. If the application needs to keep a copy of the buffer contents, it must copy the actual data rather than keep a copy of the pointer. The API has been designed however in such a way that frequent copying should not be needed.

If an application does need to defer message processing it can use `acu_ss7mon_copy_msg()` and `acu_ss7mon_free_msg()` to copy the message information to malloced memory.

Additional decoder API functions are provided that set various filtering options on received data, or perform additional decoding of messages that have been filtered by the application.

4.8 Automatic ISUP decode

The application may request the library to automatically decode ISUP messages, and to provide automatic tracking of individual ISUP calls. When used in this mode, the message buffers returned by `acu_ss7mon_get_msg()` will also contain a pointer to an ISUP-specific 'information buffer' containing ISUP message parameters, and a pointer to a 'call detail buffer' in which a summary of an ongoing call, including it's current state, and certain parameters such as called and calling numbers, is accumulated.

As with the message buffers themselves, the ISUP information and details buffers are allocated within the library, which may free or reuse them as soon as the application makes another API call. If the application needs to keep a copy of the buffer contents, it must copy the actual data rather than keep a copy of the pointer. The call details buffers should not need to be copied often however, as its contents are cumulative and a fresh pointer to it is provided with each ISUP message for a given ISUP call.

4.9 Additional and alternative user part decoders

Application writers can add further user part decoders using `acu_ss7mon_set_upart_decode()`. Such decoders could be written for protocols, such as TUP or DUP, which are not provided by Aculab. User-written decoders can be fully integrated into the API and invoked by the same mechanisms as for library-provided decoders.

Although the source code for the library's internal ISUP code is not published, it is treated just like a user-written decoder, and is enabled in the same way as a user-written decoder. Therefore, an application could be written with its own decoder for ISUP, in preference to using the ISUP decoder by Aculab, for example, if required for a national variant that differed significantly from the ITU-T recommendations.

4.10 Active MTP2 interface

The monitor uses Aculab's normal MTP2 implementation, but configures it to skip the protocol processing and pass received data directly to the user. To access the MTP2 protocol itself it is only necessary to not set the 'monitor flag'.

Both the basic and preventive cycle error correction methods can be used.

There are some additional MTP2-specific functions that are described in section 5.5.

4.10.1 Coding guidelines for MTP2 users

It has already been explained, in section 3.1, that the entire monitor API is non re-entrant. The same applies to the MTP2 API extensions, and so a single-threaded application interface is required. Whilst it may be possible to write an entire monitor application that runs as a single thread, that approach would be much more difficult for an MTP2 user and so MTP2 users may prefer to use a multithreaded application with mutex-protection, provided by the application, around all calls into this API.

If mutex protection is used, then the user clearly will not want to call `acu_ss7mon_get_msg()` with a timeout as that would cause the calling thread to block with a mutex held. MTP2 users will therefore probably make use of `acu_ss7mon_getsocket()`, which provides the native OS identifier for the TCP socket. That allows the application to contrive to only call `acu_ss7mon_get_msg()` when there is some reason to expect that a message is available and so it can be called with a timeout of zero.

The library communicates with the MTP2 software via a TCP connection. When an application calls the function `acu_ss7mon_get_msg()`, even if there has been some indication that a message is available, the value of `ACU_SS7MON_MSG_NO_DATA` may be returned. This indicates that the monitor library has processed some data received from the TCP connection internally, but that the data was handled within the library, and that the data was of no significance to the API user. The application must ignore such events and continue to process message indications.

Moreover it is important, after there has been an indication from the OS that data may be available, that the user always calls `acu_ss7mon_get_msg()` and repeatedly calls it until it returns `ACU_SS7MON_MSG_NO_DATA`. That is because the library uses this function internally to send certain acknowledgement and management messages to the PMX card and, if the user does not call `acu_ss7mon_get_msg()`, these messages would not get sent.

4.10.2 MTP2 link status and statistics

The MTP2 status and statistics can be displayed by requesting ss7maint directly connect to the card instead by using:

```
ss7maint linkstatus -2 [-vC] [-h -i interval] [-S timeslot] -B board_ip
    -K board_key
```

The values can be obtained by the application using **`acu_ss7mon_ioctl_req()`** see section 5.2.5

4.10.3 MTP2 tracing

The MTP2 code generates trace messages that normally get written to the ss7 driver trace (read by **`ss7maint trace`**, and decoded by **`ss7maint decode`**). These are suppressed for monitor endpoints (they replicate the receive data), but can be useful for active MTP2 users.

The trace messages can be enabled and disabled using the **`SS7_IOC_LINK_TRC_FLTR`** ioctl command, see section 5.2.5.

The trace messages have type **`ACU_SS7MON_TRACE`** and contain binary data in an undocumented format. They can be converted to a printable form using the functions in the **`libacu_ss7decode.so`**/**`acuss7_decode.dll`** library **`#include <ss7_lib_decode.h>`** to get the definitions.

In order to decode the trace messages the library needs a short header (containing the message type and length) at the start of the trace data, set **`ACU_SS7MON_MEF_RAW_TRACE`** so that this header isn't stripped.

The code fragment below shows how the decode functions are used.

```
struct su_info *su = ss7_decode_init_su_info(NULL); // calls malloc()
...
switch (msg->mm-msg_type) {
...
case ACU_SS7MON_MSG_TRACE:
     ss7_decode_su(su, SS7_TRACE_FMT_SIGTRAN | SS7_TRACE_F_DECODE_TRACE,
              msg->mm_buffer, msg->mm_buflen);
     printf("%s", ss7_decode_get_output(su));
     break;
...
ss7_decode_free_su_info(su); // calls free()
```

The same structure can be used to decode multiple messages without being reinitialised.

For ANSI decode call **ss7_decode_set_variant(su, "ansi")** after allocating the structure.

Although shown above in the message processing loop, the trace decode isn't written to be particularly fast and may affect the ability to keep up with received data especially if a large number of signalling links are active. It is better to save the raw data and decode if needed.

4.10.4 MTP2 congestion (sending or receiving SIB)

There are no specific API functions or events relating to MTP2 congestion control, but there is some interaction with the API as follows:

Local congestion (sending SIB) occurs when the number of data buffers queued by MTP2 for MTP3 becomes excessive. This situation can be caused by failure of the API user to consume the buffers fast enough, by failing to call `acu_ss7mon_get_msg()` when Rx data is queued.

When remote congestion is encountered by MTP2 (SIB received), MTP2 may be unable to consume Tx buffers that are passed from the API user. If that happens, the Tx flow control mechanism may be encountered, such that the function `acu_ss7mtp2_data_req()` returns the value `ACU_SS7MTP2_WRN_TX_FLOW`. Please see section 5.5.4 for further detail.

5 API function reference

5.1 Endpoint management functions

5.1.1 acu_ss7mon_lib_version()

This function obtains the version string of the monitor library.

Synopsis

```
const char *acu_ss7mon_lib_version(void);
```

Return value

A C string identifying the version of monitor library.

5.1.2 acu_ss7mon_create_endpoint()

Allocates and initialises an endpoint data area.

Synopsis

```
acu_ss7mon_ep_t *acu_ss7mon_create_endpoint(void);
```

Each endpoint allows an application to connect to a single card. To monitor traffic from multiple cards, separate endpoints must be created for each card. An application can connect multiple endpoints to the same card.

Any number of signalling links can be monitored using a single endpoint.

Messages from all the links being monitored on a single endpoint are kept in the order in which they are received. Monitoring the transmit and receive sides of a signalling link through the same endpoint will significantly reduce problems with message sequencing.

Note **It is almost certainly best to use as few endpoints as possible.**

5.1.3 acu_ss7mon_delete_endpoint()

Frees all resources (closing any TCP/IP connection) associated with the specified endpoint.

Synopsis

```
void acu_ss7mon_delete_endpoint(acu_ss7mon_ep_t *ep);
```

Parameters

ep
The address of an endpoint created using `acu_ss7mon_create_endpoint()`.

5.1.4 acu_ss7mon_configure_endpoint()

Modifies the characteristics of an endpoint data area.

Synopsis

```
int acu_ss7mon_configure_endpoint(acu_ss7mon_ep_t *ep,
          acu_ss7mon_cfg_param_t param, ...);
```

Parameters

ep
The address of an endpoint created using `acu_ss7mon_create_endpoint()`.

param, ...
Identifies the parameter to be modified and its associated integer or string value; one of:

`ACU_SS7MON_CFG_L2_MODE, l2_modes`

Defines the mode for the interface. Set to a bitwise 'or' of the following flags:

`ACU_SS7MON_L2_MODE_RAW`	Normal monitor mode (default for monitor).
`ACU_SS7MON_L2_MODE_APP_EST_REQ`	Suppress the call to `acu_ss7mtp2_est_req()` on monitor links so that the application can supply link-dependant parameters..
`ACU_SS7MON_L2_MODE_MTP2`	Active MTP2 link (value is zero).
`ACU_SS7MON_L2_MODE_T1`	Active MTP2 link using the T1 defaults for the MTP2 parameters.
`ACU_SS7MON_L2_MODE_PCR`	MTP2 PCR link.

`ACU_SS7MON_CFG_DECODE_FLAGS, flags`

Modifies the way the library decodes received messages. It must be followed by a bitwise 'or' of the following flags:

`ACU_SS7MON_MEF_DECODE_MTP3`	Call `acu_ss7mon_decode_mtp3()` on all received data messages.
`ACU_SS7MON_MEF_DECODE_USERPART`	Call `acu_ss7mon_decode_upart()` on all received level 3 messages.
`ACU_SS7MON_MEF_EXTENDED_SEQ`	Use the MTP2 extended sequence format from Figure A.1/Q.703. The high BSN and FSN bits are discarded.
`ACU_SS7MON_MEF_RAW_TRACE`	Leave the 8 byte header on `ACU_SS7MON_MSG_TRACE` indications.

By default, the library will not perform any MTP3 or user part decoding.

`ACU_SS7MON_CFG_TRACE_FILENAME, filename`

Creates a trace file with the specified name.
The file is formatted so that it can be processed by `ss7maint decode`.
This parameter can be set from an environment variable with the same name.
For security reasons the filename specified by the environment variable can only contain alphanumerics, `'.'`, `'-'` and `'_'`.

`ACU_SS7MON_CFG_TRACE_MAX_FILE_SIZE, filesize`

The maximum size of the trace file (in bytes) before a new file with suffix `.n` is opened.
Default 1048576 (1MB), if set to zero log file rotation is suppressed.

`ACU_SS7MON_CFG_TRACE_FILE_KEPT, count`

The number of old trace files that are kept.
Default 10.

`ACU_SS7MON_CFG_TRACE_LEVEL, level`

Trace verbosity (higher values are more verbose).
The default 5 suppressed traces of monitored data. Set to 6 to include these.
This parameter can be set from an environment variable with the same name

`ACU_SS7MON_CFG_CARD_KEY, security_key`

Sets the value of the security key that will be used when connecting to the card. It must be followed by a pointer to a null-terminated character string containing the actual key. If the value of the key is not known, it can be obtained using the Aculab Resource Manager API.

`ACU_SS7MON_CFG_PT_MONITOR, pt_monitor_parameters`

Use the alternate monitor for Prosody T cards (see section 2.6.3).
The following parameters are supported:

`-t timeslot`	Lowest 'Prosody S' timeslot to use. 128+ are on the second stream, 256+ on the second card (if present).

To enable the pt_monitor with no additional parameters configure the empty string `""`.

`ACU_SS7MON_CFG_RXBUF_SIZE, buffer_size`

Changes the size of the receive buffer area within the monitor library. It must be followed by an integer containing the size in bytes. If this parameter is not specified it defaults to 65536, which should suffice for most applications.
Receive data is also buffered in the socket receive buffer and the TCP/IP connection.

`ACU_SS7MON_CFG_HIGH_SPEED, timeslot_count`

Sets the number of timeslots for high speed signalling links. This would normally be either 30 or 31 depending on whether timeslot 16 is used or not.

`ACU_SS7MON_CFG_HSL_TS_IGNORE, timeslot_bitmask`

Sets a bit-pattern of source timeslots to ignore on high speed links.
In particular setting it to 0x10000 causes timeslot 16 be ignored.

`ACU_SS7MON_CFG_H100_STREAM, tdm_stream_number`

Set the H.100 stream number for retiming port 0 receive data of high speed signalling links on Prosody X rev 3 cards. Ports 1 up use sequential H.100 streams.
The timeslots used match the port timeslot.
The default is 0.

`ACU_SS7MON_CFG_L2_CONFIG, l2_config_string`

Provides a default configuration string used when links are created on this endpoint. `acu_ss7mon_configure_link()` can be used to change some parameters on an active link. Refer to section 6 (MTP2 configuration parameters) for a description of supported strings.

The function returns zero to indicate success, or a negative value if an error occurred as listed in section 5.6.

5.1.5 acu_ss7mon_connect()

Establishes a TCP/IP connection to the card/chassis.

Synopsis

```
int acu_ss7mon_connect(acu_ss7mon_ep_t *ep, const char *ip_address);
```

Parameters

ep
The address of an endpoint created using `acu_ss7mon_create_endpoint()`.

ip_address
The card's IP address, either as a numeric address (e.g. "192.168.42.42"), or a name that can be resolved to such an address. Both IPv4 and IPv6 addresses are supported. If `getaddrinfo()` returns multiple addresses they are tried in turn until one succeeds.

The address can be obtained from the resource manager's `acu_get_card_info()` function.

The card security key must have been configured, as described in section 5.1.4.

This function should only be called once for each endpoint, regardless of how many links will be monitored. The function returns zero to indicate success, or a negative value if an error occurred as listed in section 5.6.

Note **This function blocks while establishing the connection.**

5.1.6 acu_ss7mon_connect_card_id()

Establishes a TCP/IP connection to the card/chassis using information obtained from the Aculab resource manager.

Synopsis

```
int acu_ss7mon_connect_card_id(acu_ss7mon_ep_t *ep, ACU_CARD_ID *card_id);
```

Parameters

ep
The address of an endpoint created using `acu_ss7mon_create_endpoint()`.

card_id
Card identifier returned by `acu_open_card()`.

This is an inline function that calls the Aculab resource manager `acu_get_card_info()` function then uses the `card_key`, `card_serial_no` and `ip_address` fields to configure the endpoint and then calls `acu_ss7mon_connect()`.

The `card_id` is also saved and is used by `acu_ss7mon_set_sw()` to make the required TDM switch matrix connections.

5.1.7 acu_ss7mon_get_socket()

Returns the native identifier for the socket used for the connection to the card.

Synopsis

```
socket_t acu_ss7mon_get_socket(acu_ss7mon_ep_t *ep);
```

This allows the application to use OS calls (e.g. `select()`, `poll()`, and `WSAEventSelect()`) to wait for library events and indications from outside of the library at the same time.

Note **Once the OS has indicated that the socket is readable, the application must call `acu_ss7mon_get_msg()` in a loop until it indicates that no more indications are available.**

Parameters

ep
The address of an endpoint created using `acu_ss7mon_create_endpoint()`.

5.1.8 acu_ss7mon_get_socket_error()

Returns the last native error code from a socket function.

Synopsis

```
int acu_ss7mon_get_socket_error(acu_ss7mon_ep_t *ep);
```

Such error codes are operating system specific, so reference needs to be made to the appropriate OS documentation.

Parameters

ep
The address of an endpoint created using `acu_ss7mon_create_endpoint()`.

The function returns zero to indicate success, or a negative value if an error occurred as listed in section 5.6.

5.2 Link monitoring functions

5.2.1 acu_ss7mon_monitor_link()

Requests that the card allocate an HDLC/MTP2 engine and corresponding timeslot for monitoring or data transfer.

Synopsis

```
int acu_ss7mon_monitor_link(acu_ss7mon_ep_t *ep, unsigned int link_id,
          unsigned int stream, unsigned int timeslot)
```

Parameters

ep
The address of an endpoint created using `acu_ss7mon_create_endpoint()`.

link_id
Is an application-provided token that is not used by the library. It may take any value, and will be returned to the application in the `mm_link_id` of every message indication for this link.

stream and ***timeslot***
These are nominally the switch matrix stream and timeslot for the link being monitored. However the monitor library doesn't configure the switch matrix and these fields may be set to any value. The values may appear in diagnostic/debug output from other tools.

The request is processed asynchronously by the card. Successful return from this function indicates that the request has been sent. If the request is accepted, the card will respond with a message with the `mm_type` field set to either `SS7MON_MSG_MONITOR_ACK` or `SS7MON_MSG_MONITOR_FAIL`, to indicate success or failure respectively.

The `SS7MON_MSG_MONITOR_ACK` indication contains the timeslot number needed to make the TDM switch connection between the hdlc block and the E1/T1 TDM timeslot. The easiest way to do this is to call `acu_ss7mon_set_sw()`; see 5.2.2.

If you are monitoring a local signalling link the source stream for the transmit traffic can be found by using the switch driver API call `sw_query_output()` on the external timeslot.

Once the `SS7MON_MSG_MONITOR_ACK` indication is received the monitoring must be started by calling `acu_ss7mtp2_set_req()`. For monitor links this is normally done automatically passing the configuration string specified by `acu_ss7mon_configure_endpoint(ep, ACU_SS7MON_CFG_L2_CONFIG, config_string)`.

The function returns zero to indicate the request was accepted, or a negative value if an error occurred as listed in section 5.6.

5.2.2 acu_ss7mon_set_sw()

Connects the specified TDM timeslot (normally on an E1/T1 trunk) to the monitor.

Synopsis

```
int acu_ss7mon_set_sw(acu_ss7mon_msg_t *monitor_ack_msg, unsigned int stream,
          unsigned int timeslot)
```

Parameters

monitor_ack_msg
The address of the received message that indicates the monitor link request was accepted. The endpoint structure and stream/timeslot of the monitor are taken from the message.

stream and ***timeslot***
Switch API stream and timeslot for the source data to be monitored.
These do not need to match the values passed to `acu_ss7mon_montor_link()`.
Remember to add the offset (32) for E1/T1 trunks.

The receive data of HSL on PxV3 cards is retimed using an H.100 stream (use `ACU_SS7MON_CFG_H100_STREAM` to specify the stream).

The monitor's stream and timeslot are taken from the msg's `mm_status` field.
If that is zero the `mm_interface_id` is used instead.
If the value is greater than 256 then it is encoded `stream << 8 | timeslot`.
If less than 256 it is a timeslot offset into streams 48 upwards (each having 32 timeslots).

The function is implemented in an inline wrapper that passes the monitor library the address of `sw_set_output()` so that the monitor library doesn't have to be linked against the switch library. The return value is always from `sw_set_output()`.

Note **This function can only be used if acu_ss7mon_connect_card_id() was used and the application has called acu_open_switch() for the card.**

5.2.3 acu_ss7mon_unmonitor_link()

Requests that the card stop monitoring a link.

Synopsis

```
int acu_ss7mon_unmonitor_link(acu_ss7mon_ep_t *ep,
          unsigned int interface_id);
```

Link monitoring stops automatically if the TCP/IP connection is broken, so it is not essential to call this before disconnecting the connection.

Parameters

ep
The address of an endpoint created using `acu_ss7mon_create_endpoint()`.

interface_id
The value from `mm_interface_id` in a received message of type `ACU_SS7MON_MONITOR_ACK` (See section 5.2.6).

The function returns zero to indicate success, or a negative value if an error occurred as listed in section 5.6.

5.2.4 acu_ss7mon_configure_link()

Allows a configuration string to be passes to the card to dynamically modify the characteristics of a monitored link.

Synopsis

```
int acu_ss7mon_configure_link(acu_ss7mon_ep_t *ep, unsigned int interface_id,
          const char *cfg_string);
```

Parameters

ep
The address of an endpoint created using `acu_ss7mon_create_endpoint()`.

interface_id
The value from `mm_interface_id` in a received message of type `ACU_SS7MON_MSG_MONITOR_ACK` (See section 5.2.6).

cfg_string
A `NULL` terminated string containing one or more space-separated parameter strings.

The full list of parameters is given in section 6 (MTP2 configuration parameters). However many are not appropriate once a link is active.

For a monitor link the following parameters can be changed at anytime:

`raw_rx_min=nn`
Specifies the shortest HDLC frame (‘`nn`’) that will be passed to the application, frames shorter than this are filtered out. This improves performance by reducing processing overheads for the continuously repeated MTP2 LSSU and FISU traffic. The default is 6.

`raw_rx_min_short=nn`
Specifies a once-only frame length (less than `raw_rx_min`). After this is called, the next frame with (`size >= raw_rx_min_short && size < raw_rx_min`) will be returned. Subsequent short frames will be suppressed unless the function is repeated. This can be used to identify the current state of an MTP2 link.

For an active MTP2 link any of the protocol parameters and timers can by changed.

The function returns zero to indicate success, or a negative value if an error occurred as listed in section 5.6.

5.2.5 acu_ss7mon_ioctl_req()

Send miscellaneous requests to the card.

Synopsis

```
#include <ss7monitor_l2stats.h>
int acu_ss7mon_ioctl_req(acu_ss7mon_ep_t *ep, unsigned int interface_id,
          unsigned int ioc_cmd, const void *buf, unsigned int buflen);
```

Parameters

ep
The address of an endpoint created using `acu_ss7mon_create_endpoint()`.

interface_id
The value from `mm_interface_id` in a received message of type `ACU_SS7MON_MSG_MONITOR_ACK` (See section 5.2.6).

ioc_cmd
The command code, one of:

`SS7_IOC_LINK_MTP2_STATS`
Requests the statistics values that are obtainable through the maintenance API for signalling links being used by the ss7 driver, and are displayed by **`ss7maint linkstatus -2vC`**.
buf should be **NULL**.

`SS7_IOC_LINK_MTP2`
Requests the **`ss7_ioc_link_mtp2`** structure that contains the internal state of MTP2. This is the data displayed by **`ss7maint linkstatus -2vv`**.
buf should be **NULL**.

`SS7_IOC_LINK_TRC_FLTR`
This allows the application control the events which generate **`ACU_SS7MON_MSG_TRACE`** indications. The flags for receive and transmit FISU, LSSU, in sequence MSU, out of sequence MSU and state engine messages can be individually set or cleared.
These are the flags modified by **`ss7maint prottrace`**.
By default all are disabled on monitor links and enabled on other links, except that out of sequence MSU are disabled for PCR links.
Duplicate FISU and LSSU are always suppressed.
buf should point to an **`ss7_ioc_link_trc_fltr_t`** structure defining the flags to change.

buf and ***buflen***
The address and length of any buffer associated with the request.

The function returns zero to indicate the request has been sent to MTP2, or a negative value if an error occurred as listed in section 5.6.

The response to the request itself will be a message of type **`ACU_SS7MON_MSG_IOC_RESPONSE(`*`ioc_cmd`*`)`**.

5.2.6 acu_ss7mon_get_msg()

Reads a message from the TCP/IP connection to the card.

Synopsis

```
int acu_ss7mon_get_msg(acu_ss7mon_ep_t *ep, acu_ss7mon_msg_t **msg,
          int tmo_ms);
```

The messages read by this function may contain asynchronous responses, for example following `acu_ss7mon_monitor_link()`, or they may provide pointers to buffers containing data captured from a signalling link.

Note **The msg information only remains valid until the next call to `acu_ss7mon_get_msg()` or `acu_ss7mon_isup_reject()` for the same endpoint.**

Parameters

ep
The address of an endpoint created using `acu_ss7mon_create_endpoint()`.

msg
The address of a message pointer.

tmo_ms
Specifies the number of milliseconds to wait for data, zero means don't wait, negative values wait forever.

The function returns zero to indicate success, or a negative value if an error occurred as listed in section 5.6. Errors should be treated as fatal and the endpoint deleted.

Upon successful return the item pointed at by `msg` will have been written with a pointer to an `acu_ss7mon_msg_t` structure which has the following fields:

mm_ep
Always the same as `ep`.

mm_link_id and ***mm_interface_id***
These identify the signalling link that the message is associated with.
mm_link_id is the **link_id** value the application passed to **acu_ss7mon_monitor_link()**.
mm_interface_id is the monitor's reference and is passed as the **interface_id** parameter on further api calls for the link.
They are both set to `~0u` if the message isn't associated with a link.

mm_status
An integer status value associated with the message.

mm_buffer and ***mm_buflen***
Address and length of any data buffer associated with the message.

mm_msg_type
Identifies the type of the message, and will be one of:

`ACU_SS7MON_MSG_NO_DATA`
A timeout occurred waiting for a message, or a received message (TCP connection data) was processed within the library.

`ACU_SS7MON_MSG_TRACE`
The message contains data that the SS7 driver would write to its trace.
These are almost all suppressed for monitor links and should be ignored.
Applications using active MTP2 links may want to save the data for later decode see section 4.10.3.

`ACU_SS7MON_MSG_MONITOR_ACK`
Indicates that the card has successfully processed an earlier call to `acu_ss7mon_monitor_link()`. The application should save the `mm_interface_id` value. `mm_buffer` contains the version string of the MTP2 software.

`ACU_SS7MON_MSG_MONITOR_FAIL`
Indicates that the card encountered an error while processing an earlier call to `acu_ss7mon_monitor_link()` `mm_status` contains the failure reason, `mm_buffer` contains an error message.

`ACU_SS7MON_MSG_UNMONITOR_ACK`
Indicates that the card has processed an earlier call to `acu_ss7mon_unmonitor_link()` `mm_status` indicates whether it was successful.

`ACU_SS7MON_MSG_IOC_RESPONSE(`*`ioc_cmd`*`)`
The messages contains the data requested by an **acu_ss7mon_ioctl_req()** call.

`ACU_SS7MON_MSG_L1DATA`
This indicates that an HDLC frame (longer than `raw_rx_min` bytes) has been received, for which no further processing was identified. `mm_buffer` and `mm_buflen` will describe a complete HDLC message, from the first byte after the opening flag to (but not including) the HDLC checksum.

`ACU_SS7MON_MSG_UNKNOWN`
This message contains captured data that the library was unable to parse. The `mm_buffer` and `mm_buflen` will be as for L1 data.

`ACU_SS7MON_MSG_L3DATA`
Indicates that the message has been processed by the MTP3 message decoder. Refer to sections 5.3.2 and 7 for further details.

`ACU_SS7MON_MSG_ISUPDATA`
Indicates that the message has been processed by the library's internal ISUP message decoder. `mm_buffer` and `mm_buflen` will describe an ISUP message, from the ISUP 'CIC' field to (but not including) the HDLC checksum. Refer to section 8 for further details.

`ACU_SS7MON_MSG_AUDIODATA`
Indicates that the message contains raw A-law/u-law data from a link with `monitor_audio` set to a non-zero value using the pt monitor.

`ACU_SS7MON_MSG_L2DATA_BAD`
Indicates that an HDLC frame has been received, for which the length field in the message (as defined in ITU-T Q.703) is inconsistent with the amount of data. `mm_buffer` and `mm_buflen` will be as for L1 data.

`ACU_SS7MON_MSG_CFG_LINK_ACK`
Indicates that a previous `acu_ss7mon_configure_link()` request succeeded.

`ACU_SS7MON_MSG_CFG_LINK_FAIL`
Indicates that a previous `acu_ss7mon_configure_link()` request failed, `mm_buffer` points to a NUL terminated character string containing the error message.

When using the MTP2 API the additional messages types below will also be returned:

`ACU_SS7MTP2_MSG_EST_CONF`
Indicates that a signalling link has been activated, following an earlier call to `cu_ss7mtp2_est_req()`.

`ACU_SS7MTP2_MSG_L2DATA`
As for the monitor API, this indicates receipt of L2 data. The data, consisting of (inclusively) all MSU content from SIO to end of message, will be described by the message fields `mm_buffer` and `mm_bufflen`. `acu_ss7mon_decode_mtp3()` may be used to decode the MTP3 routing label.

`ACU_SS7MTP2_MSG_REL_IND`
Indicates that a signalling link has failed, or that an attempt to activate one has failed.

`ACU_SS7MTP2_MSG_REL_CONF`
Indicates that a signalling link has been deactivated following a call to `acu_ss7mtp2_rel_req()`.

`ACU_SS7MTP2_MSG_TX_FLOW_ON`
Indicates that a Tx flow control restriction has cleared and the user may resume data transmission using `acu_ss7mtp2_data_req()`. The flow control would have been indicated earlier in the return value from the function `acu_ss7mtp2_data_req()`.

`ACU_SS7MTP2_MSG_RTRV_BSNT`
Indicates the MTP2 has responded to a request using `acu_ss7mtp2_rtrv_req()`, to retrieve the BSNT value as required for MTP3 changeover procedures. The retrieved BSNT is contained in the `mm_status` field of the message.

`ACU_SS7MTP2_MSG_RTRV_DATA`
Indicates that MTP2 is providing a retrieved data buffer, following a request using `acu_ss7mtp2_rtrv_req()`.

`ACU_SS7MTP2_MSG_RTRV_COMP`
Indicates that MTP2 has completed retrieval of all data buffers, following a request using `acu_ss7mtp2_rtrv_req()`.

`ACU_SS7MTP2_MSG_CANT_RTRV`
Indicates that MTP2 has been unable to provide a response, following a request using `acu_ss7mtp2_rtrv_req()`.

`ACU_SS7MTP2_MSG_RPO_ENTER, ACU_SS7MTP2_MSG_RPO_EXIT`
Indicates that a signalling link has entered or exited a Remote Processor Outage condition.

5.2.7 acu_ss7mon_copy_msg()

This function copies a message into malloced memory.

Synopsis

```
acu_ss7mon_msg_t *acu_ss7mon_copy_msg(acu_ss7mon_nsg_t *msg);
```

Parameters

msg
The address of a message to copy.

Return value

A deep copy of the supplied message. `NULL` if `malloc()` failed.

Note Any pointers that the application or any userpart decoders have into the message are not updated.

5.2.8 acu_ss7mon_free_msg()

This function frees a message allocated by `acu_ss7mon_copy_msg()`.

Synopsis

```
void acu_ss7mon_free_msg(acu_ss7mon_nsg_t *msg);
```

Parameters

msg
The address of a message to free.

Note On Windows systems it is important to use this function not free().

5.2.9 acu_ss7mon_interface_id_to_stream(), acu_ss7mon_interface_id_to_ts()

These functions are implemented as macros, and allow the application to discover the stream and timeslot that the card has assigned for the monitoring of a signalling link.

Synopsis

```
int acu_ss7mon_interface_id_to_stream(unsigned int interface_id);
int acu_ss7mon_interface_id_to_ts(unsigned int interface_id);
```

Parameters

interface_id
The value from `mm_interface_id` in a received message of type `ACU_SS7MON_MSG_MONITOR_ACK` (See section 5.2.6).

Note It is easier to use acu_ss7mon_set_sw() to set the switch connection (See section 5.2.2).

5.2.10 acu_ss7mon_get_dsp_version()

This function obtains the version string of the MTP2 software.

Synopsis

```
const char *acu_ss7mon_get_dsp_version(acu_ss7mon_ep_t *ep);
```

Parameters

ep
The address of an endpoint created using `acu_ss7mon_create_endpoint()`.

Return value

A C string identifying the version of the MTP2 software running on the card. Will be `NULL` if the endpoint isn't connected to a card.

5.3 Message decoder functions

5.3.1 acu_ss7mon_set_pointcode_size()

Sets the point code size for one, or all, network indicators.

Synopsis

```
int acu_ss7mon_set_pointcode_size(acu_ss7mon_ep_t *ep, unsigned int ni,
          unsigned int pc_size);
```

Parameters

ep
The address of an endpoint created using `acu_ss7mon_create_endpoint()`.

ni
The value of the network indicator. A value of ~0u means 'all network indicators'.

pc_size
Must be 14, 16, or 24 or 0. A value of '0' indicates that the point code size is not known, which will disable MTP3 and user part decodes. The default value, that applies if this function is never called, is 0.

The function returns zero to indicate success, or a negative value if an error occurred as listed in section 5.6.

5.3.2 acu_ss7mon_decode_mtp3()

This function can be called to explicitly request an MTP3 decode for a message.

Synopsis

```
int acu_ss7mon_decode_mtp3(acu_ss7mon_msg_t *msg);
```

Parameters

msg
Must point to a message buffer that meets the following criteria:

- It must be the most recent message buffer to have been returned by `acu_ss7mon_get_msg()`.
- The message buffer must have `mm_type` set to either `ACU_SS7MON_L1DATA` or `ACU_SS7MON_L2DATA`.

If the `ACU_SS7MON_MEF_DECODE_MTP3` flag is set `acu_ss7mon_get_msg()` calls this function before returning with `mm_type` set to `ACU_SS7MON_L1DATA` or `ACU_SS7MON_L2DATA`.

The decode returns zero to indicate success, or a negative value if an error occurred as listed in section 5.6. Unexpected message data generates success with `mm_type` set appropriately, not failure.

The fields of `msg` are updated as follows:

mm_bsn_bib, ***mm_fsn_fib*** and ***mm_len_pri***
Set from the first few bytes of the MTP2 header (unless `mm_type` was `ACU_SS7MON_L2DATA`).
If `ACU_SS7MON_MEF_EXTENDED_SEQ` is set the high bits of the sequence numbers are discarded.
If the MTP2 length field is inconsistent with that of the message itself `mm_type` is set to `ACU_SS7MON_MSG_L2DATA_BAD` and success is returned.
If extended sequence numbers are used these values are generated excluding the high bits.

mm_sio, ***mm_si*** and ***mm_ni***
Set from the SIO byte.

If the pointcode size for the si and ni has not been set, or the message is too short to contain the MTP3 routing label, success is returned without any further changes (`mm_type` is unchanged).

mm_dpc, ***mm_opc*** and ***mm_sls***
Set from the MTP3 routing label.

mm_h1h0
Set from the message if si is 0, 1 or 2, otherwise set to 0.

mm_buffer and ***mm_buflen***
Adjusted to exclude the MTP2 header and MTP3 routing label.

mm_msg_type
Set to `ACU_SS7MON_L3DATA`.

If `ACU_SS7MON_MEF_DECODE_USERPART` is set, the relevant user-part decode function is then called.

5.3.3 acu_ss7mon_set_upart_decode

Specifies a user part decoder function that the library will invoke for messages that meet defined criteria.

Synopsis

```
int acu_ss7mon_set_upart_decode(acu_ss7mon_ep_t *ep, unsigned int ni,
          unsigned int si, unsigned int pc_1, unsigned int pc_2,
          int (*decode_fn)(acu_ss7mon_msg_t *, acu_ss7mon_ui_t *),
          const void *cfg);
```

This function specifies a user part decoder function that the library will invoke for messages that meet the defined criteria, when returning a message to the user after `acu_ss7mon_get_msg()` is called. It is also used to enable the library's internal ISUP message decoder (see section 8).

Parameters

ep
The address of an endpoint created using `acu_ss7mon_create_endpoint()`.

ni
A network indicator value for which the function will be invoked. A value of ~0 means 'all network indicators'.

si
A service indicator value for which the function will be invoked.

pc_1 and ***pc_2***
Define the signalling relation for which the function will be invoked. They can be given in any order, and either or both may be set to ~0, meaning 'all pointcodes'.

decode_fn and ***cfg***
A pointer to a function that will be called to decode a message, and a user-defined parameter for that function. This function is passed an `acu_ss7mon_msg_t *` parameter, and an `ss7mon_ui_t *` parameter. The first points to the message for processing, and the second points to structure containing the following two fields:

> `ui_config`
> The `cfg` value that was provided to `acu_ss7mon_set_upart_decode()`.
>
> `ui_state`
> A pointer that will be the same for all messages for an individual relation (pair of pointcodes). It will initially be `NULL`, but the application can change this, on a per-relation basis, at any time. After changing it, the new value will be reflected on subsequent messages for the same signalling relation.

The decode function will be called with a `NULL` message address as a request to delete any saved resources if/when the endpoint is freed.

The decode function must return zero to indicate success or a negative value, as listed in section 5.6, if an error occurred. Any error response may be passed to the caller of `acu_ss7mon_get_msg()` who will consider it as a fatal error.

A return value of zero indicates success, a negative value indicates an error as listed in section 5.6.

5.3.4 acu_ss7mon_decode_upart()

Allows application-specific filtering to be applied to messages before requesting a full decode.

Synopsis

```
int acu_ss7mon_decode_upart(acu_ss7mon_msg_t *msg);
```

This function can be called manually to request a user part decode, and causes the decode function provided by `acu_ss7mon_set_upart_decode()` to be invoked for the message. It allows application-specific filtering to be applied to messages before requesting a full decode, and allows the application to request decoding of messages that have not been recognised for the user part by the library.

Parameters

msg

Must point to a message buffer that meets the following criteria:

- It must be the most recent message buffer to have been returned by `acu_ss7mon_get_msg()`.
- It must already have had an MTP3 decode performed, so that the fields in that structure are set as described in section 6.

If the `ACU_SS7MON_MEF_DECODE_USERPART` flag was set when configuring the endpoint, then the user part decoder function will be called by `acu_ss7mon_decode_mtp3()`.

A return value of zero indicates success, a negative value indicates an error as listed in section 5.6.

5.4 ISUP decoder functions

Refer to section 8 for information on how to use the ISUP decode functions.

5.4.1 acu_ss7mon_isup_accept()

Requests the library to commence tracking for an ISUP call.

Synopsis

```
int acu_ss7mon_isup_accept(acu_ss7mon_isup_lib_ref_t,
          acu_ss7mon_issup_user_ref_t);
```

This function may be called after the library's internal ISUP decoder has indicated a new ISUP call has started. It allows the application to request the library to commence tracking for the call. The current message being handled by the application (the last one returned by `acu_ss7mon_get_msg()`) must be an ISUP Begin call message.

Parameters

lib_ref
Must be the same value that the library indicated in the current message.

user_ref
May be any non-zero value. It will be passed out from the library on all subsequent messages that relate to the same the same call.

If the application wants to be notified of subsequent messages for the call, it must call this function before making another call to `acu_ss7mon_get_msg()`. Otherwise, the library assumes the application has no interest in the call.

A return value of zero indicates success, a negative value indicates an error as listed in section 5.6.

5.4.2 acu_ss7mon_isup_reject()

Tells the library to stop tracking an ISUP call.

Synopsis

```
int acu_ss7mon_isup_reject(acu_ss7mon_isup_lib_ref_t);
```

This function may be called at any time following an earlier call to `acu_ss7mon_isup_accept()`, until a message from the library indicating the ISUP call has ended. It allows the application to inform the library that it has no further interest in a ISUP call, so the library can free resources and will generate no more indications.

When this function is called, the library may free the data buffers associated with the message most recently returned by `acu_ss7mon_get_msg()`, so the use must not dereference these resources again.

The library automatically tidies up at the end of each call after processing the ISUP Release Complete Message. This function only needs to be used when the application initially accepts the call but subsequently, and before the call has ended, has a change of mind.

A return value of zero indicates success, a negative value indicates an error as listed in section 5.6.

5.4.3 acu_ss7mon_isup_locate_parameter()

Searches an ISUP message for a specific parameter.

Synopsis

```
int acu_ss7mon_isup_locate_parameter(acu_ss7mon_msg_t *msg, int param_name,
          unsigned char **param_data, int *param_length)
```

This function searches an ISUP message for a specific parameter, returning details (address and length) of the first, if any, occurrence of the parameter in the message.

Parameters

msg
Must be the most recent message returned by `acu_ss7mon_get_msg()`, and the `mm_msg_type` field must have been set the library's internal ISUP decoder to `SS7MON_MSG_ISUPDATA`.

param_name
Identifies the parameter, as per table 5/Q.763. This must be in the range 1-255.

param_data
Is filled in by the library with the address of the start of parameter content.

param_length
Is filled in by the library with the length of the parameter content.

A return value of zero indicates success, a negative value indicates an error as listed in section 5.6.

5.4.4 acu_ss7mon_isup_find_next_parameter()

Scans through an ISUP message returning each parameter in turn.

Synopsis

```
int acu_ss7mon_isup_find_next_parameter(acu_ss7mon_msg_t *msg, int param_itr,
          char *param_name, unsigned char **param_data, int *param_length)
```

This function returns the parameters in an ISUP message one by one. The fixed and mandatory variable parameters are returned first, followed by the optional ones.

Parameters

msg
Must be the most recent message returned by `acu_ss7mon_get_msg()`, and the `mm_msg_type` field must have been set the library's internal ISUP decoder to `SS7MON_MSG_ISUPDATA`.

param_itr
Iterator value. This should be set to zero to obtain the first parameter, or to the return value from an earlier call to get the following parameter.

param_name
Is filled by the library with the parameter name as per table 5/Q.763.

param_data
Is filled in by the library with the address of the start of parameter content.

param_length
Is filled in by the library with the length of the parameter content.

Return value

If a parameter is found a positive number that should be passed back as the `param_itr` value to obtain the next parameter.

If there are no more parameters, or there is a format error `ACU_SS7MON_ERROR_PRM_NOT_FOUND`.

5.5 MTP2-specific API functions

5.5.1 acu_ss7mtp2_est_req()

Requests the library to establish a signalling link.

Synopsis

```
int acu_ss7mtp2_est_req(acu_ss7mon_ep_t *ep, unsigned int interface_id,
          const char *cfg_string);
```

This function requests MTP2 to initiate link activation.

Parameters

ep
The address of an endpoint created using `acu_ss7mon_create_endpoint()`.

interface_id
The value from `mm_interface_id` in a received message of type `ACU_SS7MON_MONITOR_ACK` (See section 5.2.6).

cfg_string
A NUL terminated string containing one or more space-separated parameter strings. See section 6 (MTP2 configuration parameters) for a list of the valid parameters.

If `NULL` the string configured by `ACU_SS7MON_CFG_L2_CONFIG` is used.

All the configuration parameters are set back to their defaults before establishing the link, so should be specified every time this function is called.

If the link can be successfully established, the event will be indicated in a message of type `ACU_SS7MTP2_MSG_EST_CONF`. If the link fails to establish, a message of type `ACU_SS7MTP2_MSG_REL_IND` will be signalled.

If the link fails to establish (or fails at a later stage) then it can be re-established by calling this function again, if the link is actually up an `ACU_SS7MTP2_MSG_EST_CONF` response is returned.

The function returns zero to indicate success, or a negative value if an error occurred as listed in section 5.6.

Note **If ACU_SS7MON_L2_MODE_APP_EST_REQ is set this must be called on monitor links enabling per-link configuration strings.**

5.5.2 acu_ss7mtp2_rel_req()

Requests the library to deactivate a signalling link.

Synopsis

```
int acu_ss7mtp2_rel_req(acu_ss7mon_ep_t *ep, unsigned int interface_id)
```

Parameters

ep
The address of an endpoint created using `acu_ss7mon_create_endpoint()`.

interface_id
The value from `mm_interface_id` in a received message of type `ACU_SS7MON_MONITOR_ACK` (See section 5.2.6).

The function returns zero to indicate success, or a negative value if an error occurred as listed in section 5.6.

5.5.3 acu_ss7mtp2_state_req()

Requests the library to set or clear processor outage or emergency proving states.

Synopsis

```
int acu_ss7mtp2_state_req(acu_ss7mon_ep_t *ep, unsigned int interface_id,
          int state)
```

Parameters

ep
The address of an endpoint created using `acu_ss7mon_create_endpoint()`.

interface_id
The value from `mm_interface_id` in a received message of type `ACU_SS7MON_MONITOR_ACK` (See section 5.2.6).

state
Must be set to one of:

`ACU_SS7MTP2_REQ_LPO_SET, ACU_SS7MTP2_REQ_LPO_CLEAR`
To set or clear local processor outage.

`ACU_SS7MTP2_REQ_EMER_SET, ACU_SS7MTP2_REQ_EMER_CLEAR`
To set or clear emergency proving.

If emergency proving is requested while proving is in progress then MTP2 will change to doing emergency proving (ie sending SIE). Clearing the flag will not affect active proving.

The function returns zero to indicate success, or a negative value if an error occurred as listed in section 5.6.

5.5.4 acu_ss7mtp2_data_req()

Requests the library to transmit MTP3 data over a signalling link

Synopsis

```
int acu_ss7mtp2_data_req(acu_ss7mon_ep_t *ep, unsigned int interface_id,
          int length, const unsigned char *address)
```

Parameters

ep
The address of an endpoint created using `acu_ss7mon_create_endpoint()`.

interface_id
The value from `mm_interface_id` in a received message of type `ACU_SS7MON_MONITOR_ACK` (See section 5.2.6).

length
The length of the data to be sent.

address
The memory address of the data to be sent. The data must contain all MSU data from (inclusively) Service Information Octet to end of message.

The function may return a value of `ACU_SS7MTP2_WRN_TX_FLOW_OFF` to indicate that the data could not be sent owing to flow control. In that case, when the flow control clears, it will be indicated by a message of type `ACU_SS7MTP2_MSG_TX_FLOW_ON`, as described in section 5.2.6, after which the application must send the data again. Flow control conditions can be transitory (just waiting for a software ACK from MTP2); it should not be assumed that there is a significant amount of data queued with MTP2 unless the condition persists.

Other than flow control conditions, the function returns zero to indicate success, or a negative value if an error occurred as listed in section 5.6.

Note **This function can also be used in 'monitor mode' to send a complete hdlc frame (CRC16 added).**
The function returns zero to indicate success, or a negative value if an error occurred as listed in section 5.6.

5.5.5 acu_ss7mtp2_rtrv_req()

Requests the library to commence retrieval activity.

Synopsis

```
int acu_ss7mtp2_rtrv_req(acu_ss7mon_ep_t *ep, unsigned int interface_id,
          int action, int seq)
```

This function requests that MTP2 participates in retrieval procedures. It can be used to retrieve the BSNT (backward sequence number) from MTP2 Tx state machine, which is the FSN (forward sequence number) of the last received message. Alternatively, it can be passed a FSN that that has been obtained from the remote MTP2, to request retrieval of unacknowledged data messages from the transmission and retransmission queues.

Parameters

ep
The address of an endpoint created using `acu_ss7mon_create_endpoint()`.

interface_id
The value from `mm_interface_id` in a received message of type `ACU_SS7MON_MONITOR_ACK` (See section 5.2.6).

action
Must be set to one of:

`ACU_SS7MTP2_REQ_RTRV_BSNT`
To obtain the Tx State machines last BSNT value.

`ACU_SS7MTP2_REQ_RTRV_DATA`
To initiate retrieval of data messages.

seq
The sequence number of the last message received at the far end of the link. Only used when action is set to `ACU_SS7MTP2_REQ_RTRV_DATA`.

Following successful execution of this function, the retrieved sequence number, or data messages, will be indicated via the message interface as described in section 5.2.6.

The function returns zero to indicate success, or a negative value if an error occurred as listed in section 5.6.

5.6 Error handling functions

5.6.1 acu_ss7mon_get_error_text()

Converts an error code into human-readable text.

Synopsis

```
const char *acu_ss7mon_get_error_text(int error);
```

Parameters

error

Must be an error value returned by one of the monitor's API functions, for example:

```
ACU_SS7MON_NO_PT_MONITOR,       "Can't start pt_monitor"
ACU_SS7MON_NO_CARD_INFO,        "acu_get_card_info() failed"
ACU_SS7MON_BAD_STATE_REQ,       "Invalid state req"
ACU_SS7MON_BAD_INTERFACE_ID,    "Invalid interface id"
ACU_SS7MON_BAD_DATA,            "Invalid data ptr or length"
ACU_SS7MON_BAD_ACTION,          "Invalid action parameter"
ACU_SS7MON_INCORRECT_CARD_KEY, "Incorrect card key"
ACU_SS7MON_AUTH_FAILURE,        "Authentication failed"
ACU_SS7MON_NO_CARD_KEY,         "Card key required, but not configured"
ACU_SS7MON_AUTH_TIMEOUT,        "Authentication timed out"
ACU_SS7MON_BAD_HOSTNAME,        "Unable to resolve address"
ACU_SS7MON_BAD_LIB_REF,         "invalid library reference"
ACU_SS7MON_BAD_USER_REF,        "invalid user reference"
ACU_SS7MON_PRM_NOT_FOUND,       "parameter not found in message"
ACU_SS7MON_UNKNOWN_CONFIG,      "unknown configuration parameter"
ACU_SS7MON_BAD_PC_SIZE,         "invalid point code size"
ACU_SS7MON_BAD_SI,              "invalid si value"
ACU_SS7MON_BAD_NI,              "invalid ni value"
ACU_SS7MON_MSG_VERSION,         "message corrupt (bad version)"
ACU_SS7MON_MSG_OVERLONG,        "message corrupt (overlong)"
ACU_SS7MON_SEND_FAILED,         "send() failed"
ACU_SS7MON_RECV_FAILED,         "recv() failed"
ACU_SS7MON_DISCONNECTED,        "remote end disconnected"
ACU_SS7MON_ALREADY_CONNECTED,   "endpoint already connected"
ACU_SS7MON_CONNECT_FAILED,      "connect() failed"
ACU_SS7MON_NO_SOCKET,           "socket() failed"
ACU_SS7MON_MALLOC_FAIL,         "malloc() failed"
ACU_SS7MON_SUCCESS,             "Success'
```

The list of errors may change between product releases, so the above list may be incomplete. The authoritative (full) list is that provided in the API header file `ss7monitor.h`

6 MTP2 configuration parameters

The parameters are the same as those described in the `[MTP2]` section of the SS7 installation and administration guide.

To specify multiple parameters separate them with a space character.

The parameters split into 3 groups:

Link setup; these are normally set before calling `acu_ss7mon_monitor_link()` or passed to `acu_ss7mtp2_est_req()` immediately after receiving the `SS7MON_MSG_MONITOR_ACK` indication:

`speed=nn` (not Prosody 1U enterprise)
: Specifies the speed (in k bits/sec) of the signalling link. Set to 56 to monitor 56k signalling links on ANSI networks.
The default is 64.

`timeslot=nn` (not Prosody 1U enterprise)
: Forces the HDLC channel to be assigned to the specified timeslot (0 to 126).
In monitor mode this can be the same timeslot as is being used by an MTP2 signalling link and allows the receive traffic be monitored without changing the TDM switch matrix.
In monitor mode the HDLC transmit is disabled until a data frame is transmitted.
If not specified the timeslot is usually the same as the **`interface_id`**, these are allocated from zero upwards.

`monitor_tx_ts=nn` (not Prosody 1U enterprise)
: Monitors the transmit data from the local HDLC engine that is using the specified timeslot (0 to 126).
This allows the transmit traffic for a local signalling link be monitored without changing the TDM switch matrix.
HDLC transmit is completely disabled.

`monitor_audio=nn` (only pt monitor on Prosody X Evo)
: Monitor the raw A-law/u-law data stream in blocks of length `nn` samples (max 281).
If set to zero the data is scanned for hdlc frames.

Monitor control; these can also be changed dynamically using `acu_ss7mon_configure_link()`:

`raw_rx_min=nn`
: Specifies the shortest HDLC frame (‘`nn`’) that will be passed to the application, frames shorter than this will be filtered out. This improves performance by reducing processing overheads for the continuously repeated MTP2 LSSU and FISU traffic. The default is 6.

`raw_rx_min_short=nn`
: Specifies a once-only frame length (less than `raw_rx_min`). After this is called, the next frame with (`size >= raw_rx_min_short && size < raw_rx_min`) will be returned.
Subsequent short frames will be suppressed until the parameter is set again.
This can be used to force a FISU be returned, or to detect receipt of an LSSU indicating that the link has failed.

MTP2 protocol parameters; these would normally be passed to acu_ss7mtp2_est_req(), but can be changed while the MTP2 link is active:

`aerm_tin=`*`nn`*, `aerm_tie=`*`nn`*, `iac_m=`*`nn`*, `suerm_t=`*`nn`*, `suerm_d=`*`nn`*, `tx_n1=`*`nn`*, `tx_n2=`*`nn`*, `trace_fw=[`*`y/n`*`]`, `trace_rx=[flmv]`, `trace_tx=[flmv]`, `set_misc_flags=`*`nn`*, `clr_misc_flags=`*`nn`*, *`tn=sss`*
: Refer to the SS7 installation and admin guide for further details.

7 Using the MTP3 decoder

When messages containing SS7 user part data are received from the library, before decoding the user part data, the MTP3 'header' (consisting of the routing label and service information octet) must be decoded. The MTP3 decoder may be invoked automatically by the library when it detects a message containing an MTP3 payload, or it may be invoked manually by the application.

The MTP3 decoder sets the following values in the `acu_ss7mon_msg_t` structure:

`mm_dpc`
Will contain the destination point code from the routing label.

`mm_opc`
Will contain the originating point code from the routing label.

`mm_sls`
Will contain the sls/slc from the routing label.

`mm_ni`
Will contain the network indicator from the service information octet.

`mm_si`
Will contain the service indicator from the service information octet.

`mm_buflen` and `mm_buffer`
These will describe an MTP3 'payload', starting with the first byte of data following the routing label, and continuing up the end of the message (not including the HLDC checksum).

An MTP3 routing label can only be decoded if the point code lengths are known, so the application needs to provide the library with point code lengths as well as enabling the MTP3 decode flag. Enabling the MTP3 decoder is thus a two step process (in any order):

`acu_ss7mon_configure_endpoint()` is used to set the MTP3 decode flags.
`acu_ss7mon_set_pointcode_size()` is used to define the point code size.

The second step (above) may be repeated for different network indicators, enabling the application to specify different point code lengths for different network indicators.

Example: Enable the MTP3 decoder for all point codes, using 14 bit point code lengths for all network indicators,

```
response = acu_ss7mon_configure_endpoint(endpoint,
            ACU_SS7MON_CFG_DECODE_FLAGS, ACU_SS7MON_MEF_DECODE_MTP3);
response = acu_ss7mon_set_pointcode_size(endpoint, ~0, 14);
```

Example: Similar to the previous example, but using two alternative pointcode lengths, 14 bits and 24 bits, for "International" network indicators (0 & 1) and "National" network indicators (2 & 3) respectively.

```
response = acu_ss7mon_point(endpoint,
            ACU_SS7MON_CFG_DECODE_FLAGS, ACU_SS7MON_MEF_DECODE_MTP3);
response = acu_ss7mon_set_pointcode_size(endpoint, 0, 14);
response = acu_ss7mon_set_pointcode_size(endpoint, 1, 14);
response = acu_ss7mon_set_pointcode_size(endpoint, 2, 24);
response = acu_ss7mon_set_pointcode_size(endpoint, 3, 24);
```

8 Using the internal ISUP decoder

The SS7 signalling monitor incorporates an internal ISUP decoder which is capable of tracking individual ISUP calls. The decoder may be invoked automatically by the library when it detects an ISUP message, or it may be invoked manually by the application.

As well as keeping track of the progress of individual calls, the ISUP decoder automatically decodes calling and called number strings, which are converted at the API into printable character strings.

Regardless of whether or not the internal ISUP decoder is in use, ISUP messages are provided as raw data at the API, allowing applications to access additional parameters as raw data. To make this easier, the API includes a function to resolve the raw data address of individual parameters, which can then be decoded by the application with reference to the message and parameter specifications defined in ITU-T Q.763, or national variant thereof.

8.1 Enabling the decoder

The ISUP decoder is enabled using `acu_ss7mon_set_upart_decode()`, which may be called at some time after creating an endpoint. This function has other uses as well (see section 9), and includes in its input parameters the network indicator, service indicator, and two pointcodes. These parameters act as filters, so that ISUP decoding is only applied to messages that match the supplied values.

The service indicator must be set to the value that the network will use for ISUP traffic. The ITU recommendation for international ISUP traffic is that service indicator ‘5’ be used. This value has also been adopted by the vast majority of National variants, to the extent that deviation from it is almost unheard of. Strictly speaking, if the ISUP traffic is using a network indicator other that 0 (international) then the corresponding service indicator value is a national matter and reference should be made to the national protocol specification.

When enabling the ISUP decoder, `acu_ss7mon_set_upart_decode()` requires a *`cfg`* parameter that identifies the national protocol variant. Aculab currently support variants for ITU, China, and ANSI, for which the appropriate macro for *`cfg`* must be used, for example:

```
ACU_SS7MON_ISUP_ITU_CFG
ACU_SS7MON_ISUP_ANSI_CFG
ACU_SS7MON_ISUP_CHINA_CFG
```

The point codes and network indicator may be set to specific values or ~0, in any combination, ~0 meaning ‘all values’. The decoder may be enabled more than once with different combinations of point codes, network indicator, and protocol variants.

Example: For all traffic to/from point code 2020, enable ITU ISUP decoder for network indicator 0:

```
response = acu_ss7mon_set_upart_decode(endpoint, 0, 5, 2020, ~0,
                                       acu_ss7mon_decode_isup,
                                       ACU_SS7MON_ISUP_ITU_CFG);
```

Example: For traffic between point codes 2020 and 7070, Enable ITU ISUP decoder for all network indicators:

```
response = acu_ss7mon_set_upart_decode(endpoint, ~0, 5, 2020, 7070,
                                        acu_ss7mon_decode_isup,
                                        ACU_SS7MON_ISUP_ITU_CFG);
```

Example: For all point codes, enable ANSI ISUP decoder for network indicator 2, alongside ITU ISUP decoder for network indicator 0:

```
response = acu_ss7mon_set_upart_decode(endpoint, 0, 5, ~0, ~0,
                                       acu_ss7mon_decode_isup,
                                       ACU_SS7MON_ISUP_ITU_CFG) ;
response = acu_ss7mon_set_upart_decode(endpoint, 2, 5, ~0, ~0,
                                       acu_ss7mon_decode_isup,
```

```
                ACU_SS7MON_ISUP_ANSI_CFG);
```

8.2 Automatic or manual decode

Most application writers will probably choose to allow the library to automatically invoke the ISUP decoder based on the filtering criteria defined when it was enabled (see section 8.1). To do so, the application needs to configure the endpoint for automatic MTP3 and User Part decode. This involves two further steps in addition to the enabling the decoder, these steps may be performed in either order after creating an endpoint:

Step 1: `acu_ss7mon_configure_endpoint()` is used to add the ISUP and MTP3 decode flags.
Step 2: `acu_ss7mon_set_pointcode_size()` is used to define the point code size.

Example: enable automatic ITU ISUP decode, based on 14 bit point codes, for traffic on all Network indicators, Service indicator 5.

```
response = acu_ss7mon_configure_endpoint(endpoint,
             ACU_SS7MON_CFG_DECODE_FLAGS,
             ACU_SS7MON_MEF_DECODE_MTP3 | ACU_SS7MON_MEF_DECODE_USERPART);
response = acu_ss7mon_set_pointcode_size(endpoint,~0,14);
response = acu_ss7mon_set_upart_decode(endpoint, ~0, 5, ~0, ~0,
             acu_ss7mon_decode_isup,
             ACU_SS7MON_ISUP_ITU_CFG);
```

Occasionally, there may be reasons why an application writer does not want the library to perform automatic user part decode, but still wants to invoke the ISUP decoder himself. In this case he still has to set the point code size and enable MTP3 decode, after which he can invoke the ISUP decoder himself if and when he sees fit.

Example: manual invocation of ISUP decoder

```
/* During intialisation... */
   response = acu_ss7mon_configure_endpoint(endpoint,
                ACU_SS7MON_CFG_DECODE_FLAGS,
                ACU_SS7MON_MEF_DECODE_MTP3);
   response = acu_ss7mon_set_pointcode_size(endpoint, ~0, 14);
   response = acu_ss7mon_set_upart_decode(endpoint, ~0, 5, ~0, ~0,
                acu_ss7mon_decode_isup,
                ACU_SS7MON_ISUP_ITU_CFG);

/* After a message has been retrieved using acu_ss7mon_get_msg... */
   response = acu_ss7mon_decode_upart(msg);
```

8.3 Message formats

When in use, the ISUP decoder will set the following values in the `acu_ss7mon_msg_t` structures retrieved by `acu_ss7mon_get_msg()`:

Within the acu_ss7mon_msg_t structure:

All of the `mm`_... fields set by mtp3 decode will be set as described in section 6 with the following qualifications:

`mm_msg_type`
Will be set to `ACU_SS7MON_ISUP_DATA`

`mm_buffer` and `mm_buflen`
For messages received from the line, these will describe an ISUP message as shown in ITU-T Q.763, commencing with the ISUP CIC field. For messages generated within the library, such as when a call is aborted (see section 8.5), the pointer will be `NULL`.

Within the acu_ss7mon_isup_info_t structure (pointed at by mm_upart_info):

ii_generic will be set to one of the following:

ACU_SS7MON_ISUP_BEGIN
Occurs when an ISUP Initial Address Message (IAM) is received. The ii_lib_ref will contain a reference that the application must remember for the duration of the call, and ii_details will point to a valid acu_ss7mon_isup_details_t structure.

This is the first message for each ISUP call, and the application needs to decide whether it wants be notified of further messages for the same call. Factors involved in this decision might include the parameters decoded by MTP3 or ISUP in the acu_ss7mon_msg_t structure, the ISUP details provided in the acu_ss7mon_isup_details_t structure, or specific parameters retrieved using acu_ss7mon_isup_locate_parameter().

If the application wants further notifications, it must respond by calling acu_ss7mon_isup_accept(), including the application's own reference, which the library will remember for the duration of the call. Otherwise the application should simply ignore the message and call acu_ss7mon_get_msg() to process the next message, in which case the library will discard further messages for the ISUP call.

ACU_SS7MON_ISUP_CONTINUED
Indicates a message related to an ongoing call that the application has accepted using acu_ss7mon_isup_accept(). The ii_lib_ref and ii_details fields will be set as for the ACU_SS7MON_ISUP_BEGIN message, and the ii_user_ref will contain the user's reference provided in acu_ss7mon_isup_accept().

Some of the call details in the acu_ss7mon_isup_details() structure may have been updated, as described below. Further parameters can be decoded, if required, with the help of acu_ss7mon_locate_isup_parameter().

ACU_SS7MON_ISUP_RELEASE_COMPLETE
This occurs when an ISUP Release Complete (RLC) message has been received. The ii_lib_ref and ii_details fields will be set as for the ACU_SS7MON_ISUP_BEGIN message, and the ii_user_ref will contain the user's reference provided in acu_ss7mon_isup_accept().

The final message indication for each call. The library will discard all resources for the call the next time the application calls acu_ss7mon_get_msg(). There is no need for the application to call acu_ss7mon_isup_reject(), as the next call to acu_ss7mon_get_msg() will implicitly have the same effect.

ACU_SS7MON_ISUP_ABORT
This occurs when a call monitoring sequence had to be aborted within the library. The application must respond in the same way as for SS7MON_RELEASE_COMPLETE.

ACU_SS7MON_ISUP_OTHER
This message indicates that a non call-related message, or one that has not been recognised by the library, has been received. The ii_user_ref, ii_lib_ref, and ii_details fields will all be zero.

ii_cic
This field is always valid and contains the raw data value from the message's 'CIC' field.

ii_q763_type
This field is always valid and contains the raw data value from the message's 'type' field.

ii_variant
Contains the *cfg* value supplied by the application in acu_ss7mon_set_upart_decode().

ii_lib_ref, ii_user_ref and ii_details
These fields are valid for some message types but not others. Refer to description of ii_generic above.

Within the acu_ss7mon_isup_details_t structure (pointed at by ii_details):

`id_altered`
Contains a combination (bitwise 'or')of:

- `ACU_SS7MON_UPDATE_CALLING_NUMBER`
  When the message caused a change to the `cd_calling_number` field.
- `ACU_SS7MON_UPDATE_CALLED_NUMBER`
  When the message caused a change to the `cd_called_number` field.
- `ACU_SS7MON_UPDATE_STATE`
  When the message caused a change to the `id_state` field.

`id_calling_number`
Points to a character string containing the address digits of the calling number, or an empty string if the number is not available. This number may have been extracted from either an IAM message or a INF (Information) message. In normal protocol operation, the calling number is provided either in an IAM, or in an INF, but not both. If the library finds that the calling number parameter in an INF contradicts that from an IAM, it will discard the INF data and keep that from the IAM.

`id_calling_ndigits`
This fill contains the number of digits in `id_calling_number`.

`id_called_number`
This fill points to a character string containing address digits of the called number, or an empty string if the number is not available. Where 'overlap' sending is encountered, this string will be cumulative, appending digits from SAM (Subsequent address) messages after those in an IAM.

`id_called_ndigits`
This fill contains the number of digits in `id_called_number`.

`id_state`
Will be set to one of the following:

- `ACU_SS7MON_STATE_CALLING`
  The initial state after an IAM is detected, and remains until the call is 'connected'.
- `ACU_SS7MON_STATE_CONNECTED`
  This is entered when a call reaches the 'conversation' phase, i.e. after an ANM (Answer) or CON (Connect) message is received. If a call that had been suspended by the SUS (Suspend) message, and subsequently resumed by a RES (Resumed) message, this state is re-entered.
- `ACU_SS7MON_STATE_FORWARD_RELEASED`
  `ACU_SS7MON_STATE_BACKWARD_RELEASED`
  These states are entered after a release sequence is initiated. Forward release indicates release by the calling party or an exchange between the calling party and the monitor. Backward release indicates release by the called party or an exchange between the monitor and the called party. In the event of simultaneous release or a collision of backward and forward release messages, `cd_state` will indicate the first message to reach the library.

8.4 Accessing specific parameters

The monitor library automatically decodes the called and calling party numbers. To optimise performance it does not automatically decode other parameters, but they can be decoded on request by the application using `acu_ss7mon_locate_parameter()`.

Example: Obtain the 'Cause' parameter from ISUP REL message:

```
unsigned char * parm_ptr;
int parm_length ;

if (isup_info->ii_q763_type == 0x0c)   /* REL message */
    response = acu_ss7mon_isup_locate_paramater(msg,
                                                0x12, /* Cause parameter */
                                                &parm_ptr, &parm_length);
```

8.5 Message delivery errors

Under some circumstances ISUP message delivery errors may occur whereby messages are missed, duplicated, or seen in the wrong sequence. Providing the monitor is correctly configured for all possible signalling links these situations will be extremely rare, but can never be ruled out. For example, an HDLC data corruption may take place that affects only the monitor device or only the monitored switch. Other times, it may only be possible to monitor some, but not all, signalling links, in which case lost messages might be a relatively common event.

The ISUP decoder handles such errors partly by being 'forgiving' of apparent protocol errors. For example, the ITU protocol demands that an ACM message must be sent before ANM. If the monitor detects an ANM message without seeing an ACM however, it will simply proceed to the 'connected' state on the assumption the ACM was missed. If the ACM follows later it will still be processed, but the call's state, having already progressed to 'connected', will stay in that state.

If the monitor fails to detect an RLC message, it is possible for a call to appear to last forever. For this reason, application writers may wish to implement a 'sanity' timeout after which calls are assumed to have terminated. The duration of such a timeout would of course be application-dependent, and would need to take account of the nature of the expected traffic.

If the monitor detects a new call (IAM) while another call is apparently still connected for the same circuit, the library assumes that it must have missed an RLC message. It deals with this by suspending processing of the new call while it generates an `ACU_SS7MON_ISUP_ABORT` event (see section 8.3). After the application has processed this event, processing of the new call is resumed and is presented to the application in the usual way.

9 Writing additional user part decoders

User-defined decoders can be added using `acu_ss7mon_set_upart_decode()`. This instructs the library to add the user's decode function to a lookup table that will be called for messages that meet the following criteria:

- The network indicator in the message must contain a value for which the pointcode length has been specified, using `acu_ss7mon_set_pointcode_size()`.
- The Service Indicator in the message must contain the same value specified in `acu_ss7mon_set_upart_decode()`.
- The pointcode pair must match a pattern specified in `acu_ss7mon_set_upart_decode()`.
- The decode flag `ACU_SS7MON_MEF_DECODE_MTP` must have been configured using `acu_ss7mon_configure_endpoint()`.
- The decode flag `ACU_SS7MON_MEF_DECODE_USERPART`, may optionally have been configured using `acu_ss7mon_configure_endpoint()`.

For examples of setting up a user part decode, see section 8.1. This explains how to enable the libraries internal ISUP decoder, but the procedure for user-written decoders is the same.

When a user-supplied decode function is called by the library, either by virtue of the `ACU_SS7MON_MEF_DECODE_USERPART` flag or because the application called `acu_ss7mon_decode_upart()`, it receives a pointer to an `acu_ss7mon_msg_t` structure, and a pointer to an `acu_ss7mon_ui_t` structure.

The pointer to the `acu_ss7mon_msg_t` structure, if not `NULL`, will point to a library data area that will be set as for a message decoded by MTP3 as described in section 6. A `NULL` pointer indicates that the endpoint has been deleted, and the decoder should release all its resources.

The `acu_ss7mon_ui_t` structure contains two fields.

`ui_config`
The value that was supplied when `acu_ss7mon_set_upart_decode()` was called.

`ui_state`
Is a `void *` pointer which will be `NULL` for the first message of each signalling relation (point code pair). The application may (but does not need to) change this pointer to some other value. If the application does change `ui_state`, then the new value will be remembered by the library and will be provided with subsequent messages for the same signalling relation. This allows the application to allocate a per-relation structure of its own. When messages are passed to the decoder as a result of point codes being defined as ~0 (don't care), the user receives a distinct `ui_state` parameter for each individual relation that is found to match the ~0.

As the library does no validation, the decoder function can do whatever it likes to the message. Typically, it would allocate some structure of its own design for protocol-specific details, and attach it to the `mm_upart_info` pointer in the message. It would also change the `mm_type` field to something outside the range used by Aculab so that the application can recognise messages that the decoder has acted upon.

10 System configuration

In order to use the signalling monitor, each E1/T1 trunk that will be used for monitoring must be downloaded with the ss7 firmware file (eg `ss7.pmx`). This is necessary to establish the Layer 1 interface to external ports, and to allow TCP/IP connections to the monitor.

This can be achieved by using either the call control API, Aculab tools such as `fwdspldr`, or the Aculab configuration tool (ACT). The firmware parameters described below are explained in the Aculab SS7 installation and administration guide. Please refer to that document when reading the following paragraphs.

In the case of 'local monitoring', the TDM network ports should be downloaded with the ss7 firmware file and the firmware parameters as required for the signalling application(s), no special action is needed to configure the monitor.

If a TDM network port is to be used exclusively for monitoring, it should be downloaded with the ss7 firmware file. The only firmware parameters that need be specified are those that affect the physical interface, for example `-cT1` or `-cNCRC`.

Example: Using fwdspldr.exe, configure port 0 on card serial 123456 for E1 monitoring purposes only.

```
fwdspldr.exe 123456 0 ss7.pmx
```

Appendix A: DSP and switch matrix examples

This section provides further details of the procedures for DSP link assignment and TDM switch matrix initialisation. Each of the three different capture modes, identified earlier in section 2.5, is discussed and accompanied by sample code snippets.

The configuration principles described here can be applied to any and many of the external ports and timeslots of the card. For simplicity however, the examples assume that a signalling link to be monitored appears on timeslot 16 of each E1 trunk, and that the E1 trunks are connected to ports 0 and 1.

In each of the examples it can be seen that monitoring of traffic, in each direction on each individual signalling link, is a two step sequence:

Step 1

The application sends a request using `acu_ss7mon_monitor_link()` to the DSP, asking it to allocate and initialise a timeslot for monitoring. The application does not know at this point which DSP timeslot will be allocated; the DSP software will select one that is not already in use for monitoring or signalling.

Step 2

Some time later, after the DSP has processed the request from step 1 and allocated a timeslot, the application will receive a notification via the API function `acu_ss7mon_get_msg()` that identifies the timeslot. Once the application receives this notification, it uses the Aculab switch API to connect the DSP timeslot to the traffic source.

The above steps must be repeated for each signalling link, and for each direction of traffic. To monitor both `tx` and `rx` traffic on a single link, for example, the sequence is carried out twice. This sequence can be repeated and/or overlapped as desired, step 1 being performed many times followed by step 2 many times.

A.1 Passive interception example

In Figure 5, DSP timeslots are connected to the `rx` data from each of two external E1/T1 ports, which in turn are fed by cabling from the external line tapping device.

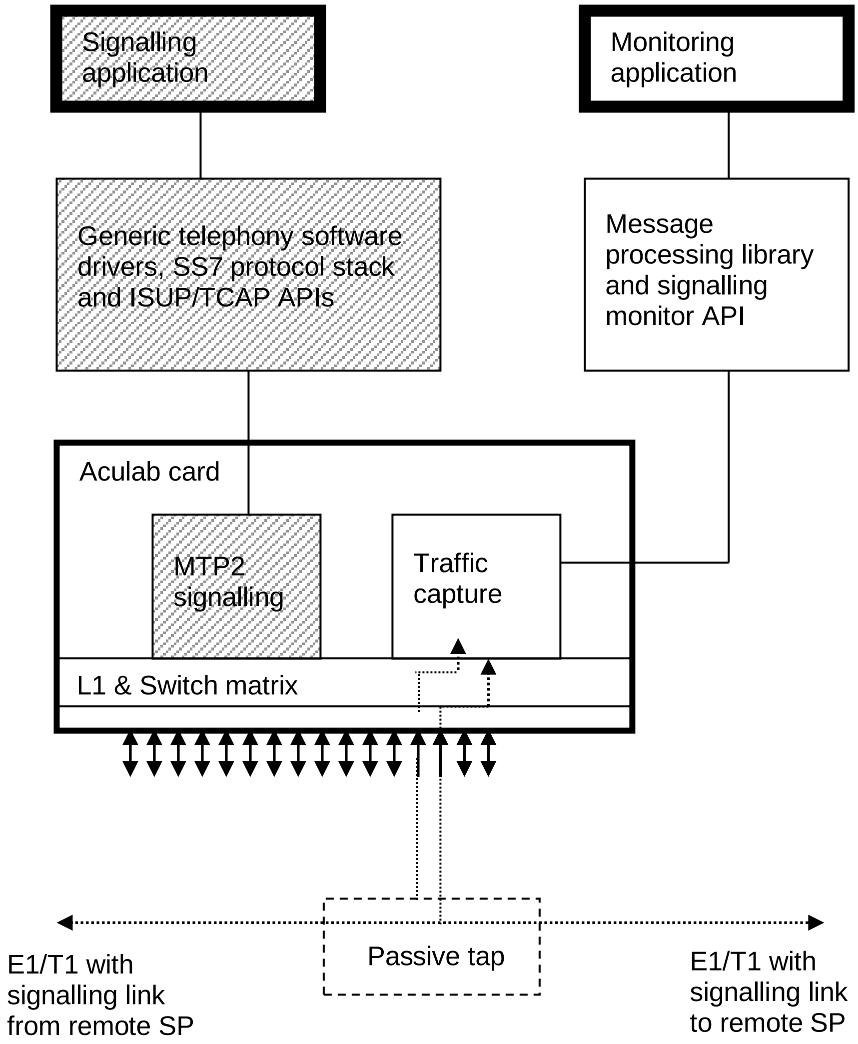

Figure 5: Passive interception example

The code to set up link monitoring and establish the DSP connections to the switch matrix, would look something like:

STEP 1: request a link monitor for traffic on each of two links, using two different application-defined `link_ids`:

```
resp = acu_ss7mon_monitor_link(ep, link_id_1, 32, 16); /* Port 0, TS 16 */
resp = acu_ss7mon_monitor_link(ep, link_id_2, 33, 16); /* Port 1, TS 16 */
```

STEP 2 for first link: after `acu_ss7mon_get_msg()` returns a message with `mm_msg_type ACU_SS7MONITOR_ACK`, with `mm_link_id` matching `link_id_1`:

```
OUTPUT_PARMS output_parms;
INIT_ACU_STRUCT(&output_parms);
output_parms.mode = CONNECT_MODE;
output_parms.ost = acu_ss7mon_interface_id_to_stream(msg->interface_id);
output_parms.ots = acu_ss7mon_interface_id_to_ts(msg->interface_id);
output_parms.ist = 32;
output_parms.its = 16;
resp = sw_set_output(card_id, &output_parms);
```

STEP 2 for second link: after `acu_ss7mon_get_msg()` returns a message with `mm_msg_type` `ACU_SS7MONITOR_ACK`, with `mm_link_id` matching `link_id_2`:

```
OUTPUT_PARMS output_parms;
INIT_ACU_STRUCT(&output_parms);
output_parms.mode = CONNECT_MODE;
output_parms.ost = acu_ss7mon_interface_id_to_stream(msg->interface_id);
output_parms.ots = acu_ss7mon_interface_id_to_ts(msg->interface_id);
output_parms.ist = 33;
output_parms.its = 16;
resp = sw_set_output(card_id, &output_parms);
```

A.2 Active interception example

In Figure 6, the DSP timeslots are fed in precisely the same way as for the passive monitor, from the `rx` data of two E1/T1 ports. In addition, the two E1/t1 ports are patched through to one another in loopback mode, eliminating the need for an external tap.

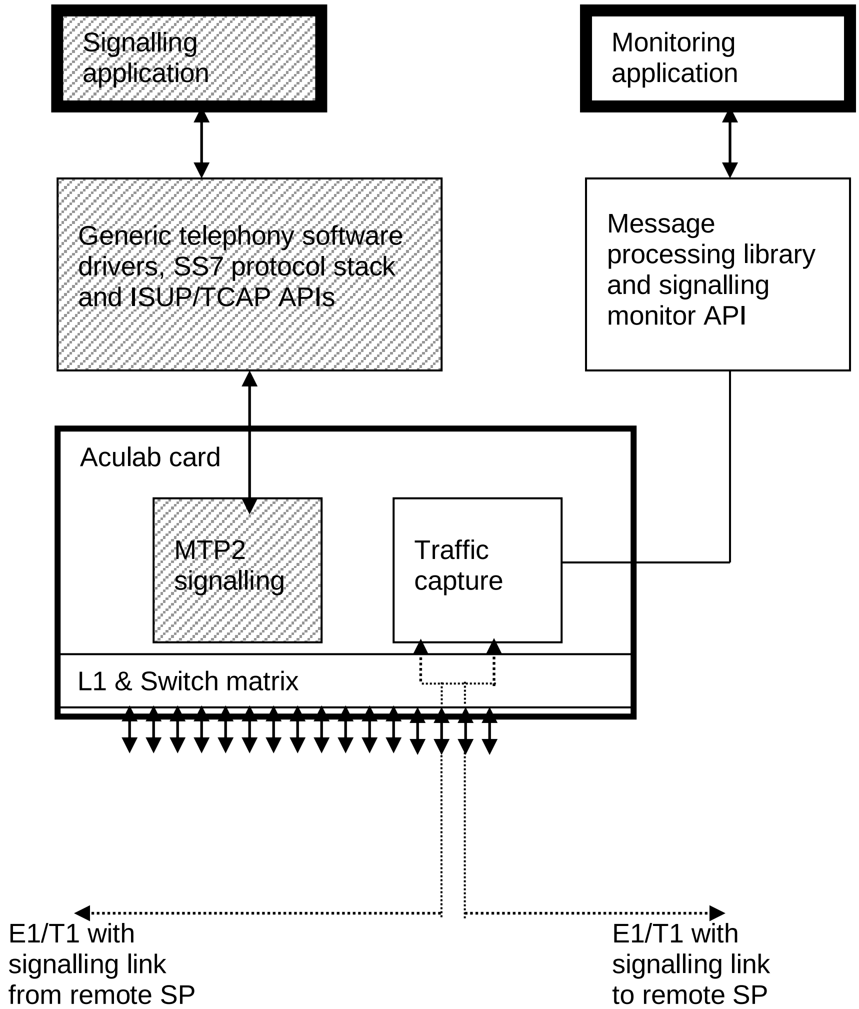

Figure 6: Active interception example

The monitor configuration code for this configuration would be exactly the same steps 1 and 2 of the passive monitoring example, see section A.1. In addition, the user would need to make the necessary pass-through connections in each direction between the two ports. Assuming an E1 trunk is in use, i.e. timeslots 1 through 31, the additional code would look something like:

```
OUTPUT_PARMS output_parms;
int ts;

for (ts = 1; ts < 32 ; ts++) {
     INIT_ACU_STRUCT(&output_parms);
     output_parms.mode = CONNECT_MODE;
     output_parms.ost = 32;
     output_parms.ots = ts;
     output_parms.ist = 33;
     output_parms.its = ts;
     resp = sw_set_output(card_id, &output_parms);

     INIT_ACU_STRUCT(&output_parms);
     output_parms.mode = CONNECT_MODE;
     output_parms.ost = 33;
     output_parms.ots = ts;
```

```
        output_parms.ist = 32;
        output_parms.its = ts;
        resp = sw_set_output(card_id, &output_parms);
    }
```

A.3 Local monitor example

In Figure 7, one DSP timeslot is fed from data received at the card's external network port and timeslot, similar to the active and passive interception. The second DSP timeslot needs to be fed from the source of the transmitted traffic from the signalling application.

The transmit traffic source will in fact be another internal DSP timeslot, which is allocated by the signalling drivers and firmware, and connected through the switch matrix to the card's external E1/T1 signalling link.

This process of DSP signalling timeslot allocation is not documented for the signalling API, as it is performed automatically, and the signalling application will have no knowledge of the DSP timeslot that is in use. The monitor application must discover it by using the Aculab switch API to enquire which internal timeslot has been connected to the E1/T1 port's signalling link. This resolves the timeslot that then needs to be connected to the monitor DSP timeslot for capturing `tx` traffic.

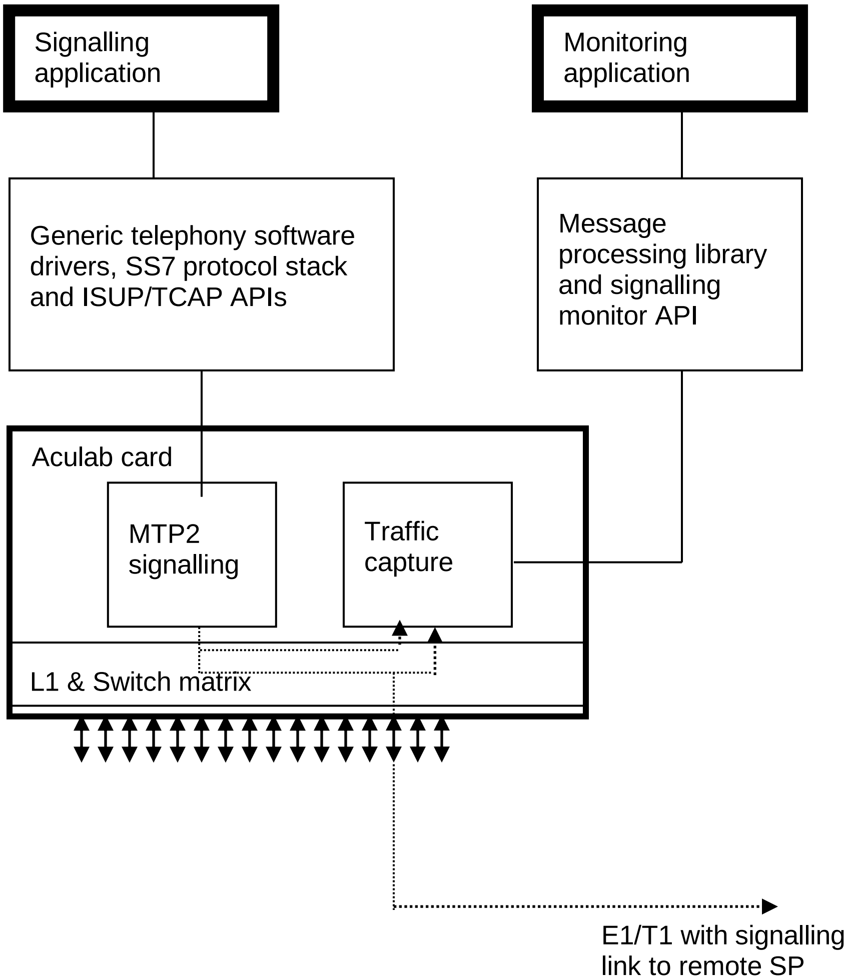

Figure 7: Local monitor example

In this configuration, the `rx` data connection is made in the same way as the for active and passive interception, but the `tx` data source needs to be established by querying the switch matrix, something like:

Step 1 (rx): request an `rx` link monitor similar to the previous examples:

```
resp = acu_ss7mon_monitor_link(ep, link_id_1, 32, 0);
resp = acu_ss7mon_monitor_link(ep, link_id_2, tx_stream, tx_timeslot);
```

Step 1 (tx): request a `tx` link monitor, using stream and timeslot resolved by `sw_query_output`:

```
OUTPUT_PARMS output_parms;
INIT_ACU_STRUCT(&output_parms);
output_parms.ost = 32;              /* Port 0 */
output_parms.ots = 16;              /* TS 16 */
resp = sw_query_output(card_id, &output_parms);

tx_stream = output_parms.ist;
tx_timeslot = output_parms.its;
resp = acu_ss7mon_monitor_link(ep, link_id_2, tx_stream, tx_timeslot);
```

Step 2 (rx): after `acu_ss7mon_get_msg()` returns a message with `mm_msg_type` `ACU_SS7MONITOR_ACK`, with `mm_link_id` matching `link_id_1`:

```
OUTPUT_PARMS output_parms;
INIT_ACU_STRUCT(&output_parms);
output_parms.mode = CONNECT_MODE;
output_parms.ost = acu_ss7mon_interface_id_to_stream(msg->interface_id);
output_parms.ots = acu_ss7mon_interface_id_to_ts(msg->interface_id);
output_parms.ist = rx_stream;
output_parms.its = rx_timeslot;
resp = sw_set_output(card_id, &output_parms);
```

Step 2 (tx): after `acu_ss7mon_get_msg()` returns a message with `mm_msg_type` `ACU_SS7MONITOR_ACK`, with `mm_link_id` matching `link_id_2`:

```
OUTPUT_PARMS output_parms;
INIT_ACU_STRUCT(&output_parms);
output_parms.mode = CONNECT_MODE;
output_parms.ost = acu_ss7mon_interface_id_to_stream(msg->interface_id);
output_parms.ots = acu_ss7mon_interface_id_to_ts(msg->interface_id);
output_parms.ist = tx_stream;
output_parms.its = tx_timeslot;
esp = sw_set_output(card_id, &output_parms);
```

Alternatively (not Prosody 1U enterprise cards):

1) Obtain the `tx_stream` and `tx_timeslot` (as in step 1 above) and calculate `mtp2_timeslot = (tx_stream - 48)*32 + tx_timeslot`.
   Or force the signalling link to have a specific timeslot (e.g. by specifying `-cMTP2_timeslot=nn` when downloading the firmware).
2) Monitor a link configured with `"timeslot=mtp2_timeslot"` to monitor the receive traffic.
3) Monitor a link configured with `"monitor_tx_ts=mtp2_timeslot"` to monitor the transmit traffic.

This method doesn't require that the application use the switch API, the monitoring program can then be run on a completely different computer.

Appendix B: Complete application example

This example illustrates a complete monitoring application that identifies and monitors ISUP calls to numbers beginning with '123'. When a call that has been accepted for monitoring eventually ends, the application generates a simple call description record, which is sent to `stdout` in plain text.

This application is for illustrative purposes only, and is not intended to represent a complete strategy for a real-world application. For instance, it assumes that the DTMF tones from the calling subscriber are transposed into the address signals of the ISUP called party number parameter, which may not always be the case – especially for services such as the speaking clock. A real-world application that was, for example, trying to identify calls to some specific country, may have to not only inspect the address signals, but also take account of the 'nature of address' or 'numbering plan' fields in the called party number, or the national/international indicator in the forward call indicators, and maybe other parameters too.

Additional parameters, such as those mentioned in the previous paragraph, are accessible to the application by using `acu_ss7mon_isup_locate_parameter()`, whereupon they can be decoded with reference to ITU-T Q.763. To illustrate this process, the example application extracts the value of the calling party's category parameter and includes it in the call description record.

The source code for this example is available for download via the Aculab AIT. The contents of that source file are reproduced and described in the following subsections. The order of these subsections is in reverse order to the source file, for example `main()` is described first in this document but appears last in the source file.

B.1 Using the sample application

The sample application takes command-line parameters for card serial number, and the two port/timeslot combinations to be monitored, as follows:

```
monitor_demo serial_number port timeslot port timeslot
```

The following shows some typical output:

```
ss7mon: dsp version 060702  Aculab SS7      V6.07.02
CALL RECORD:
        Started, Fri Sep 01 15:18:06 2006
                Called subscriber: "123"
                Caller was priority subscriber, "01908273800"
        Connected, Fri Sep 01 15:18:09 2006
        Released by calling party, Fri Sep 01 15:18:21 2006
CALL RECORD:
        Started, Fri Sep 01 15:18:23 2006
                Called subscriber: "123"
                Caller was payphone, "01234567890"
        Connected, Fri Sep 01 15:18:29 2006
        Released by called party, Fri Sep 01 15:18:31 2006
```

B.2 Source code description

B.2.1 Source file outer Scope

The outer scope of the source file contains the usual 'C' library `#include` statements (`stdio.h`, etc.), the Aculab header files required by the resource manager and switch APIs, and the monitor's own header file, '`ss7monitor.h`'.

The `#include` statements are followed by some macros for generating and parsing the *`link_id`* required by `acu_ss7mon_monitor_link()`, and a `call_record_t` structure that is used to maintain a record of each call.

The `link_id` is used to index an array index that contains application-specific information about the monitored link. In the example it only contains the port and timeslot information.

The `call_record_t` structure is used to accumulate parameters that will not be available from the acu_ss7mon_isup_details() structure, such as timestamps for state transitions, and the 'calling party category' parameter as it is not included in the library's automatic decode. There is no need to store called and calling number parameters in the call record, as they are available from the library's acu_ss7mon_isup_details() structure.

B.2.2 Function: main()

This is, of course, the entry point. It calls functions to open a card, initialise an endpoint, and requests two link monitors (one for the tx traffic and one for rx). It then calls consume_traffic(), which will loop forever until the process is terminated either by an error condition or by a manual intervention, such as a control & C break-in.

B.2.3 Function: open_card()

This function uses the Aculab resource manager to open a card identified by its serial number. The switch driver is also opened so that the tdm data can be connected. For the sake of simplicity, there is no matching close_card(), the card remains open until the application terminates.

B.2.4 Function: initialise_endpoint()

This function allocates an endpoint, turns on the library trace, and connects to the card by calling acu_ss7mon_connect_card_id(). It then sets the appropriate decode flags and pointcode sizes, and enables the library's ISUP decoder.

B.2.5 Function: request_link_monitor()

This function simply calls acu_ss7mon_monitor_link(), using function parameters derived from the user's command-line parameters.

B.2.6 Function: consume_traffic()

This function loops forever, calling acu_ss7mon_get_msg() with a 1000mS timeout. Each of these calls will resolve a pointer to a library data area that contains an acu_ss7mon_msg_t structure, which is further identified by its mm_msg_type field as follows:

ACU_SS7MON_MSG_NO_DATA
These messages can occur frequently if the traffic rate is low, as they are the result of the 1000mS timeout expiring. They are ignored, and the loop continues.

ACU_SS7MON_MSG_MONITOR_ACK
A successful response to a request to monitor a link. acu_ss7mon_set_sw() is called to complete the link initialisation.

ACU_SS7MON_MSG_MONITOR_FAIL
Indicates a failure to monitor a link. The sample code treats this as a fatal error and exits.

ACU_SS7MON_MSG_ISUPDATA
These are messages that have been decoded as ISUP messages. They are passed to process_isup_msg() for further analysis.

Other message types are ignored. This includes not only the expected messages such as MTP3 link tests, diagnostic traces, and ...get_msg() timeouts, but also includes undefined values of mm_msg_type. No error message is printed which is a fair defensive practice as future versions of the library may define other message types.

B.2.7 Function: process_isup_msg()

This function is called after the library returns a message that has been processed by the ISUP decoder. Such messages contain pointers to two further library data areas, one containing an acu_ss7mon_info_t structure, and for call-related messages, one containing an acu_ss7mon_isup_details_t structure. The first contains a field name ii_generic, which is examined to decide how the message is treated. The values of ii_generic are treated follows:

ACU_SS7_MON_ISUP_BEGIN

Indicates a new call has been detected. The function `handle_new_call()` is used to deal with it.

`ACU_SS7_MON_ISUP_CONTINUE`
Indicates any call-related message within a sequence of messages for an individual call. The function `check_details()` is used to deal with it.

`ACU_SS7_MON_ISUP_RELEASE_COMPLETE`
This type indicates a call has completed. The function `call_completed()` is used to deal with it.

`ACU_SS7_MON_ISUP_ABORT`
For simplicity, this `ii_generic` type is treated by the sample application exactly like release complete. Real-world applications may want to log at least an additional error message.

B.2.8 Function: handle_new_call()

This function is called when a new call is detected. It checks to see if the call meets the criteria for monitoring.

If the call needs to be monitored the a call record structure is allocated and `acu_ss7mon_isup_accept()` is called to tell the library to start monitoring.

If the call does not need to be monitored, this function simply returns to its caller (`consume_traffic()`), whereupon `acu_ss7mon_get_msg()` will be called again. The library deduces, from the fact that this was called without the call having been accepted, that the application does not want to monitor this call.

B.2.9 Function: call_completed()

Invoked when the last message for a call is detected. A call summary is printed based on the details provided by the library, and the timestamps accumulated from `check_details()`.

B.2.10 Function: check_details()

This function is called for each continuation message within a call. It looks for changes to the call details, and if the calling number has changed, checks to see if the call still needs to be monitored. If monitoring continues, timestamps are taken relating to the call's progress; else, the monitoring sequence of the call is aborted.

B.2.11 Function: call_wanted()

This function is called for each new ISUP call that is detected, and called again when any change called party's number is detected. The return value is a boolean indicating whether the call meets the defined criteria for monitoring, which is that the called number begins (or might begin, after further digits are received) with the string '`123...`'.

B.3 Source code listing

The source code listed here is also available for download from the AIT. If the version obtained via AIT should differ from that in this document, it should be assumed that the AIT version is the more up to date.

```
/* Stop windows compiler complaining about strcpy() and ctime() */
#define _CRT_SECURE_NO_WARNINGS

#include <string.h>
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <malloc.h>

#include <acu_type.h>
#include <ss7monitor.h>
#include <res_lib.h>
#include <sw_lib.h>

/* This struct maintains details of an ISUP call... */
typedef struct {
    acu_ss7mon_isup_lib_ref_t cr_lib_ref;
    int                       cr_connected;
    int                       cr_released;
    time_t                    cr_start_time;
    time_t                    cr_connected_time;
    time_t                    cr_release_time;

    /* Calling category is not auto-decoded, so we retrieve and
     * remember it ourselves, in the call record. */
    unsigned char             cr_calling_category;
} call_record_t;

/* Per monitored link information */
struct link_info {
    int                       lk_port;
    int                       lk_timeslot;
    int                       lk_timeslot_count; // For High speed links
    char                      lk_config[32];     // Not actually set
};

#define NUM_LINKS 2
struct link_info link_info[NUM_LINKS];

/* Called when the monitor library returns an unexpected error. */
static void
mon_lib_fatal(const char *user_text, int error)
{
    fprintf(stderr, "%s, monitor library error: %s(%d), exiting.\n",
            user_text, acu_ss7mon_get_error_text(error), error);
    exit(EXIT_FAILURE);
}

/* This function tests whether the called party number is
 * one we want to monitor. */
static int
call_wanted(acu_ss7mon_isup_details_t *details)
{
    int offset;
    int ndigits = details->id_called_ndigits;

    if (ndigits > 3)
        ndigits = 3; /* We only look at the first 3 digits */

    for (offset = 0; offset < ndigits; offset++) {
       if (details->id_called_number[offset] != "123"[offset]) {
           printf("Call to %s not wanted\n", details->id_called_number);
           return 0;
       }
    }

    /* Called number matches (or may grow to match) "123..." */
    return 1;
```

```
}

/* This function is called when the details may have changed.
 * Make sure we still want the call, and save timestamps for
 * interesting transitions. */
static int
check_details(call_record_t *call_record,
              acu_ss7mon_isup_details_t *details)
{
    int resp;

    if (details->id_altered & ACU_SS7MON_UPDATE_CALLED_NUMBER) {
        if (!call_wanted(details)) {
            resp = acu_ss7mon_isup_reject(call_record->cr_lib_ref);
            if (resp != 0)
                mon_lib_fatal("acu_ss7mon_isup_reject", resp);
            free(call_record);
        }
    }

    if (details->id_altered & ACU_SS7MON_UPDATE_STATE) {
        switch (details->id_state) {
        case ACU_SS7MON_STATE_CONNECTED:
            /* Flag shows timestamp is valid... */
            call_record->cr_connected = 1;
            time(&call_record->cr_connected_time);
            break;

        case ACU_SS7MON_STATE_BACKWARD_RELEASED:
        case ACU_SS7MON_STATE_FORWARD_RELEASED:
            /* Flag shows timestamp is valid... */
            call_record->cr_released = 1;
            time(&call_record->cr_release_time);
            break;
        }
    }
    return 0;
}

/* This is called for ISUP "Release complete" and "Abort".
 * Print then free the call record. */
static void
call_completed(call_record_t *call_record,
               acu_ss7mon_isup_details_t *details)
{
    const char *released_by, *cg_cat;
    time_t release_time;

    /* Translate some of the recognised categories into text.
     * Note, this translation is done with reference to Q.763... */
    switch (call_record->cr_calling_category) {

    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
        cg_cat = "operator";
        break;
    case 0x0a:
        cg_cat = "ordinary subscriber";
        break;
    case 0x0b:
        cg_cat = "priority subscriber";
        break;
    case 0x0c:
        cg_cat = "data call";
        break;
    case 0x0d:
        cg_cat = "test call";
        break;
    case 0x0f:
        cg_cat = "payphone";
        break;
    default:
        cg_cat = "unknown category";
```

```
        break;
    }

    printf("CALL RECORD:\n"
           "\tStarted, %s\t\tCalled subscriber: \"%s\"\n"
           "\t\tCaller was %s, \"%s\"\n",
           ctime(&call_record->cr_start_time),
           details->id_called_number,
           cg_cat, details->id_calling_number);

    if (call_record->cr_connected)
        printf("\tConnected, %s", ctime(&call_record->cr_connected_time));

    switch (details->id_state) {
    case ACU_SS7MON_STATE_FORWARD_RELEASED:
        released_by = "calling";
        release_time = call_record->cr_release_time;
        break;

    case ACU_SS7MON_STATE_BACKWARD_RELEASED:
        released_by = "called";
        release_time = call_record->cr_release_time;
        break;

    default:
        released_by = "unknown";
        time(&release_time);
        break;
    }

    printf("\tReleased by %s party, %s", released_by, ctime(&release_time));

    free(call_record);
}

/* A new ISUP call has been detected.
 * If it might be of interest accept it, else let it pass. */
static void
handle_new_call(acu_ss7mon_msg_t *msg,
                acu_ss7mon_isup_info_t *isup_info,
                acu_ss7mon_isup_details_t *details)
{
    unsigned char *prm_ptr;
    int prm_length;
    int resp;
    call_record_t *call_record;

    if (!call_wanted(details))
        return;

    /* Allocate a struct for accumulating data for call record */
    call_record = malloc(sizeof *call_record);
    if (!call_record) {
        fprintf(stderr, "Malloc failed, unable to capture call for %s\n",
                details->id_called_number);
        return;
    }

    memset(call_record, 0, sizeof *call_record);

    /* Save start time for the call. */
    time(&call_record->cr_start_time);

    /* Save the library reference for future use. */
    call_record->cr_lib_ref = isup_info->ii_lib_ref;

    /* Tell the library we want to monitor the call... */
    resp = acu_ss7mon_isup_accept(isup_info->ii_lib_ref, call_record);
    if (resp != 0)
        mon_lib_fatal("acu_ss7mon_isup_accept", resp);

    /* Retrieve calling party's category (Q.763 parameter 0x9) */
    resp = acu_ss7mon_isup_locate_parameter(msg, 0x09, &prm_ptr, &prm_length);

    /* The message _must_ have been an IAM, and CPC is _always_
     * present, so ...locate_parameter() _cannot_ have failed,
```

```
     * and the length of CPC parameter is _always_ '1'.
     *
     * If these conditions weren't met then we had a major
     * protocol breach, but just ignore it - our job here is not
     * protocol enforcement. */
    if (resp == 0 && prm_length == 1)
        call_record->cr_calling_category = *prm_ptr;
}

/* Handle a message that's been through the library's ISUP decoder */
static void
process_isup_msg(acu_ss7mon_msg_t *msg)
{
    acu_ss7mon_isup_info_t *isup_info = msg->mm_upart_info;
    acu_ss7mon_isup_details_t *details;
    call_record_t *call_record;

    /* All ISUP messages are expected to have an ISUP info area... */
    if (!isup_info) {
        /* Should never happen. */
        fprintf(stderr,
               "ISUP message: unexpected NULL pointer in mm_upart_info.\n");
        exit(EXIT_FAILURE);
    }

    if (isup_info->ii_generic == ACU_SS7MON_ISUP_OTHER) {
        /* This can happen if supervisory messages
         * (blocking etc.) are encountered */
        fprintf(stderr,
                "ISUP message: Ignoring non-call message q763 type %d.\n",
                isup_info->ii_q763_type);
        return;
    }

    /* Other ISUP messages should provide ISUP details buffer... */
    details = isup_info->ii_details;
    if (!details) {
        /* This should never happen. */
        fprintf(stderr, "ISUP message: ii_generic 0x%x, ii_q763_type 0x%x, "
                "unexpected NULL pointer in ii_details.\n",
                isup_info->ii_generic, isup_info->ii_q763_type);
        exit(EXIT_FAILURE);
    }

    if (isup_info->ii_generic == ACU_SS7MON_ISUP_BEGIN) {
        handle_new_call(msg, isup_info, details);
        return;
    }

    /* Messages other than BEGIN must refer to existing library
     * reference */
    call_record = isup_info->ii_user_ref;
    if (!call_record) {
        /* Should never happen */
        fprintf(stderr, "ISUP message: ii_generic 0x%x ii_q763_type 0x%x, "
                "unexpected NULL pointer in ii_user_ref.\n",
                isup_info->ii_generic, isup_info->ii_q763_type);
        exit(EXIT_FAILURE);
    }

    switch (isup_info->ii_generic) {
    case ACU_SS7MON_ISUP_CONTINUE:
        check_details(call_record, details);
        break;

    case ACU_SS7MON_ISUP_RELEASE_COMPLETE:
    case ACU_SS7MON_ISUP_ABORT:
        call_completed(call_record, details);
        break;

    default:
        fprintf(stderr, "Unexpected ISUP message: ii_generic 0x%x.\n",
                isup_info->ii_generic);
        break;
    }
}
```

```
/* Loop forever calling get_msg */
static void
consume_traffic(acu_ss7mon_ep_t *ep, ACU_CARD_ID card_id)
{
    int resp;
    acu_ss7mon_msg_t *msg;
    struct link_info *li;

    for (;;) {
        msg = NULL;
        resp = acu_ss7mon_get_msg(ep, &msg, 1000);
        if (resp != 0)
            mon_lib_fatal("acu_ss7mon_get_msg", resp);

        if (!msg) {
            fprintf(stderr, "acu_ss7mon_get_msg returned NULL message.\n");
            exit(EXIT_FAILURE);
        }

        if (msg->mm_link_id < NUM_LINKS)
            li = &link_info[msg->mm_link_id];
        else
            li = NULL;

        switch (msg->mm_msg_type) {
        default:
            /* Other message types are ignored, including
             * ACU_SS7MON_MSG_NO_DATA (e.g. timeouts)
             * ACU_SS7MON_MSG_L3DATA  (MTP3 link tests etc.)
             * ACU_SS7MON_MSG_TRACE   (diagnostics)
             */
            break;

        case ACU_SS7MON_MSG_ISUPDATA:
            /* This example doesn't care which link the message arrived on. */
            process_isup_msg(msg);
            break;

        case ACU_SS7MON_MSG_MONITOR_ACK:
            if (!li) {
                fprintf(stderr, "monitor ack on unexpected link %d\n",
                        msg->mm_link_id);
                exit(EXIT_FAILURE);
            }
            /* The timeslot count needs resetting before calling
             * acu_ss7mon_set_sw() if different links have different
             * counts (actually unlikely). */
            acu_ss7mon_configure_endpoint(ep, ACU_SS7MON_CFG_HIGH_SPEED,
                    li->lk_timeslot_count);
            acu_ss7mon_set_sw(msg, 32 + li->lk_port, li->lk_timeslot);
            /* Since ACU_SS7MON_L2_MODE_APP_EST_REQ was set we need to
             * request monitoring be started.
             * This lets per-link configuration be set.
             * eg the ss7_pt_monitor supports tracing of raw audio,
             * this can be enabled by passing "monitor_audio=length". */
            acu_ss7mtp2_est_req(ep, msg->mm_interface_id, li->lk_config);
            break;

        case ACU_SS7MON_MSG_MONITOR_FAIL:
            fprintf(stderr, "Library failed to monitor link %d: %s\n",
                    msg->mm_link_id, msg->mm_buffer);
            exit(EXIT_FAILURE);
        }
    }
}

/* Request a tdm trunk timeslot be monitored */
static void
request_link_monitor(acu_ss7mon_ep_t *ep, int link_no)
{
    int resp;

    acu_ss7mon_configure_endpoint(ep, ACU_SS7MON_CFG_HIGH_SPEED,
            link_info[link_no].lk_timeslot_count);
    resp = acu_ss7mon_monitor_link(ep, link_no,
```

```
                link_info[link_no].lk_port,
                link_info[link_no].lk_timeslot);

        if (resp != 0)
            mon_lib_fatal("acu_ss7mon_monitor_link", resp);
}

/* Initialise endpoint & connect to card */
static acu_ss7mon_ep_t *
initialise_endpoint(ACU_CARD_ID card_id)
{
    int resp;
    acu_ss7mon_ep_t *ep;

    /* Create a monitoring endpoint */
    ep = acu_ss7mon_create_endpoint();
    if (!ep) {
        fprintf(stderr, "acu_ss7mon_create_endpoint returned NULL.\n");
        exit(EXIT_FAILURE);
    }

    /* Enable library trace */
    acu_ss7mon_configure_endpoint(ep, ACU_SS7MON_CFG_TRACE_FILENAME,
            "monitor_demo.log");
    /* Turn trace level up (from 5) so that data messages get traced. */
    acu_ss7mon_configure_endpoint(ep, ACU_SS7MON_CFG_TRACE_LEVEL, 6);

    /* Tracing can also be enabled by setting the environment variables:
     * ACULAB_SS7_MONITOR_CFG_TRACE_LEVEL=level
     * ACULAB_SS7_MONITOR_CFG_TRACE_FILENAME=log_filename
     * (log_filename can only contain alphanumerics, '.', '-' or '_') */

    /* Uncomment to use the ss7_pt_monitor (supported by some cards).
     * The "" is additional configuration "-t 128" would avoid using
     * the first 128 timeslots (ie probably stream 256) */
    // acu_ss7mon_configure_endpoint(ep, ACU_SS7MON_CFG_PT_MONITOR, "");

    /* Disable automatic est_req() so that we can change the configuration
     * per monitored link. */
    acu_ss7mon_configure_endpoint(ep, ACU_SS7MON_CFG_L2_MODE,
            ACU_SS7MON_L2_MODE_RAW | ACU_SS7MON_L2_MODE_APP_EST_REQ);

    resp = acu_ss7mon_connect_card_id(ep, card_id);
    if (resp != 0)
        mon_lib_fatal("acu_ss7mon_connect", resp);

    /* Request auto-decode of MTP3 & ISUP */
    acu_ss7mon_configure_endpoint(ep, ACU_SS7MON_CFG_DECODE_FLAGS,
           ACU_SS7MON_MEF_DECODE_MTP3 | ACU_SS7MON_MEF_DECODE_USERPART);

    resp = acu_ss7mon_set_pointcode_size(ep, ~0, 14);
    if (resp != 0)
        mon_lib_fatal("acu_ss7mon_set_pointcode_size", resp);

    /* Enable an ITU ISUP decoder for all relations */
    resp = acu_ss7mon_set_upart_decode(ep,
                           ~0, /* All Ni */
                           5, /* SI ISUP */
                           ~0, ~0, /* All pointcodes */
                           acu_ss7mon_decode_isup,
                           ACU_SS7MON_ISUP_ITU_CFG);
    if (resp != 0)
        mon_lib_fatal("acu_ss7mon_set_upart_decode", resp);

    return ep;
}

static void
open_card(char *serial_no, ACU_CARD_ID *card_id)
{
    ACU_OPEN_CARD_PARMS open_card_parms;
    ACU_OPEN_SWITCH_PARMS open_switch_parms;
    int resp;

    INIT_ACU_STRUCT(&open_card_parms);
```

```
    strcpy(open_card_parms.serial_no, serial_no);
    resp = acu_open_card(&open_card_parms);
    if (resp != 0) {
        fprintf(stderr, "Resource manager error open card, %d\n", resp);
        exit(EXIT_FAILURE);
    }
    *card_id = open_card_parms.card_id;

    /* Open switch driver so we can tweak the matrix */
    INIT_ACU_STRUCT(&open_switch_parms);
    open_switch_parms.card_id = *card_id;

    resp = acu_open_switch(&open_switch_parms);
    if (resp != 0) {
        fprintf(stderr, "acu_open_switch: Switch API error %d\n", resp);
        exit(EXIT_FAILURE);
    }

}

static void
set_link_info(int id, char **argv)
{
    char *cp;
    int ts_hi;

    link_info[id].lk_port = strtol(argv[0], &cp, 0);
    if (*cp) {
        fprintf(stderr, "invalid port number %s\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    link_info[id].lk_timeslot = strtol(argv[1], &cp, 0);
    ts_hi = link_info[id].lk_timeslot;
    if (*cp == '-')
        /* Timeslot range for HSL */
        ts_hi = strtol(cp + 1, &cp, 0);
    if (*cp) {
        fprintf(stderr, "invalid timeslot number %s\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (ts_hi < link_info[id].lk_timeslot || ts_hi >= 32) {
        fprintf(stderr, "timeslot out of range %s\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    link_info[id].lk_timeslot_count = ts_hi - link_info[id].lk_timeslot + 1;
}

int
main(int argc, char **argv)
{
    ACU_CARD_ID card_id;
    char *serial_num;
    acu_ss7mon_ep_t *ep;

    if (argc != 6) {
        fprintf(stderr, "Usage: monitor_demo <serial> "
            "<port> <timeslot> <port> <timeslot>\n");
        exit(EXIT_FAILURE);
    }

    argv++; /* Skip past program name */
    serial_num = *argv++;

    /* Setup per-link information.
     * We only need the port and timeslot. */
    set_link_info(0, argv);
    set_link_info(1, argv + 2);

    /* Open the PMXC card */
    open_card(serial_num, &card_id);

    /* Initalise the endpoint & connect to PMX */
    ep = initialise_endpoint(card_id);

    /* Request monitoring for TX and RX traffic */
    request_link_monitor(ep, 0);
```

```
    request_link_monitor(ep, 1);

    /* Consume all future message traffic */
    consume_traffic(ep, card_id);

    return EXIT_SUCCESS;
}
```

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