Aculab SS7

Signalling monitor user's guide and API reference

MAN 1711 Revision 6.10.3

aculab

PROPRIETARY INFORMATION

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

All trademarks recognised and acknowledged.

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

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

Copyright © Aculab plc. 2006, 2007, 2008: All Rights Reserved.

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

CONTENTS

1 Introduction **5**
2 Product overview **6**
2.1 Signalling traffic capture 7
2.2 Message processing library & API 7
2.3 Traffic capture modes 7
2.4 Bearer traffic capture 9
2.5 Monitoring capacity 10
2.5.1 Signalling links 10
2.5.2 E1/T1 network ports 10
3 Application structure **11**
3.1 Threading model 11
3.2 Libraries and header files 11
4 API description **12**
4.1 Endpoints 12
4.2 TCP/IP connections 12
4.3 Event notification 12
4.4 Link monitoring 12
4.5 Receiving monitored data 12
4.6 Automatic ISUP decode 13
4.7 Additional/alternative user part decoders 13
5 API function reference **14**
5.1 Endpoint management functions 14
5.1.1 acu_ss7mon_create_endpoint() 14
5.1.2 acu_ss7mon_configure_endpoint() 14
5.1.3 acu_ss7mon_delete_endpoint() 15
5.1.4 acu_ss7mon_connect() 15
5.1.5 acu_ss7mon_get_socket() 15
5.1.6 acu_ss7mon_get_socket_error() 15
5.2 Link monitoring functions 16
5.2.1 acu_ss7mon_monitor_link() 16
5.2.2 acu_ss7mon_unmonitor_link() 16
5.2.3 acu_ss7mon_con figure_link() 17
5.2.4 acu_ss7mon_get_msg() 17
5.2.5 acu_ss7mon_interface_id_to_stream(), acu_ss7mon_interface_id_to_ts() 19
5.2.6 acu_ss7mon_get_dsp_version() 19
5.3 Message decoder functions 20
5.3.1 acu_ss7mon_set_pointcode_size() 20
5.3.2 acu_ss7mon_decode_mtp3() 20
5.3.3 acu_ss7mon_set_upart_decode 21
5.3.4 acu_ss7mon_decode_upart() 22
5.4 ISUP decoder functions 23
5.4.1 acu_ss7mon_isup_accept() 23
5.4.2 acu_ss7mon_isup_reject() 23
5.4.3 acu_ss7mon_isup_locate_parameter() 24
5.5 Error handling functions 25
5.5.1 acu_ss7mon_get_error_text() 25
6 Using the MTP3 decoder **26**
7 Using the internal ISUP decoder **27**
7.1 Enabling the decoder 27
7.2 Automatic or manual decode 28
7.3 Message formats 28
7.4 Accessing specific parameters 30
7.5 Message delivery errors 31
8 Writing additional user part decoders **32**
9 System configuration **33**

10 DSP and switch matrix examples 34
10.1 Passive interception example 35
10.2 Active interception example 37
10.3 Local monitor example 38
11 Complete application example 40
11.1 Using the sample application 40
11.2 Source code description 41
11.2.1 Source file outer Scope 41
11.2.2 Function: main() 41
11.2.3 Function: open_card() 41
11.2.4 Function: initialise_endpoint() 41
11.2.5 Function: request_link_monitor() 41
11.2.6 Function: consume_traffic() 41
11.2.7 Function: connect_link() 42
11.2.8 Function: process_isup_msg() 42
11.2.9 Function: handle_new_call() 42
11.2.10 Function: call_completed() 42
11.2.11 Function: check_details() 43
11.2.12 Function: call_wanted() 43
11.3 Source code listing 43

1 Introduction

This manual describes the Aculab SS7 signalling monitor, which allows applications to 'eavesdrop' on SS7 signalling traffic between SS7 signalling points (SP), using an Aculab card fitted with a PMXC module. It includes a description of the monitor itself with a full API specification, and explains how it relates to other Aculab products and APIs.

In order to use the signalling monitor, 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 11, with a working example application. This demonstrates a simple monitor which is triggered by calls to the 'speaking clock', accessed (in the UK) by dialling '`123`'. Each call to the speaking clock 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

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, (L1 interface and DSP firmware) 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 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.

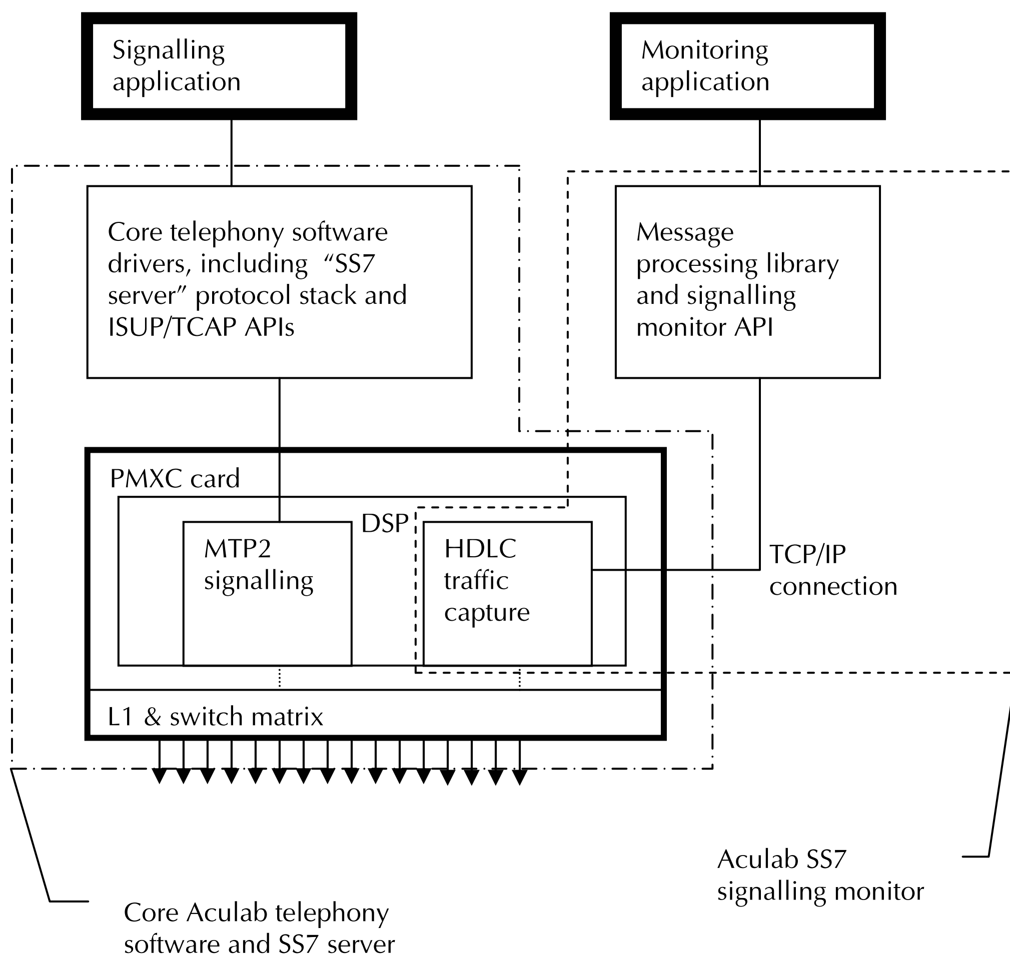

Figure 1: Major system components for monitoring and signalling

2.1 Signalling traffic capture

Traffic is captured using software that runs on a digital signal processor (DSP) on a PMXC module. This same DSP provides MTP2 signalling links for applications using the ISUP or TCAP APIs.

The DSP 'capture' software performs HDLC decoding, identifying valid HDLC frames that may need to be seen by the monitor application. In addition, the DSP software can filter out certain messages, such as MTP2 LSSUs and FISUs, which occur at very high traffic rates but are usually of no interest to a monitor application.

2.2 Message processing library & API

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

The data on the TCP/IP connection between DSP and application is encapsulated in a proprietary and unpublished Aculab protocol. One of the purposes of the message-processing library is to implement that protocol. Rather than establish a connection directly with the card, the application actually makes its connection(s), and reads the TCP/IP data, using functions provided in the monitor API.

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.

Full details of the API are provided later in this document (see sections 4 to 8).

2.3 Traffic capture modes

To capture the signalling traffic, individual DSP timeslots are first assigned for monitoring. These are then connected, using the Aculab switch matrix, to traffic sources containing the actual SS7 signalling. Two DSP timeslots are required for each SS7 signalling link being monitored, each timeslot 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 10, 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.

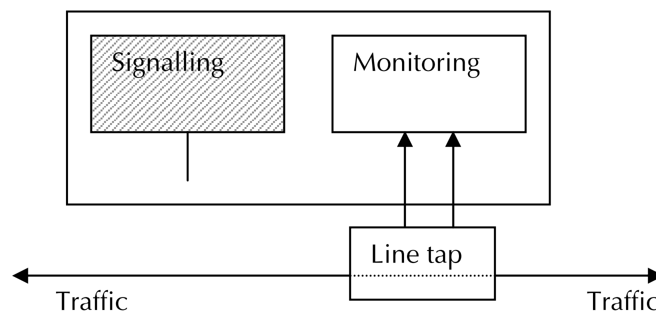

Figure 2: passive interception

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 PMXC 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 PMXC module 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.

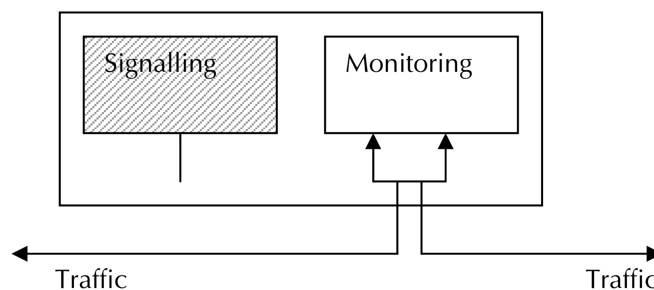

Figure 3: Active interception

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).

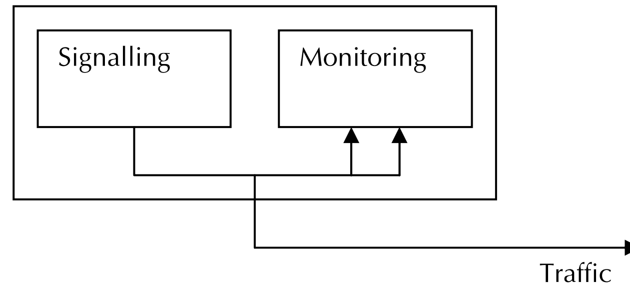

Figure 4: Local traffic monitor

All of the modes described above can coexist. A single PMXC module could 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.**

2.4 Bearer traffic capture

Each of the traffic capture modes, described in section 2.3, can also be used for monitoring of bearer traffic within E1/T1 trunks. For bearer traffic, no connection is needed to the signalling monitor DSP. The monitoring application simply uses the Aculab switch and Prosody APIs to access data on the bearer timeslots in the same way as if it were a signalling application. Refer to the Aculab switch and Prosody API guides for details.

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.**

2.5 Monitoring capacity

2.5.1 Signalling links

The total number of signalling links that can be concurrently monitored is restricted by the 'capture' software running on the DSP. The same DSP also plays a part in signalling applications, hence the actual limit needs to take account of both monitored links and driven signalling links.

As explained in section 2.3, two DSP timeslots are required to monitor a single signalling link. If an SS7 signalling application (call control or TCAP) is also in use, then each signalling link used for signalling will consume a further single DSP timeslot. This yields the following formula:

```
Total DSP timeslots = (number of signalling links) + ( 2 x number of monitored links)
```

With current software versions, Aculab can support up to 128 total DSP timeslots on each PMXC module. For example, a PMXC dedicated to monitoring could capture traffic from 64 links. Alternatively, a PMXC in use for both monitoring and signalling system could use 32 links for signalling while monitoring 48 links.

Note **This limitation is not specifically enforced with current versions of Aculab software, but configurations that exceed it are not recommended or supported.**

The monitored timeslots, and the driven signalling timeslots, can be distributed in any way among the available E1/T1 ports, subject only to the limitations mentioned in section 2.5.2.

2.5.2 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 PMXC 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 Solaris and 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 2.4, applications will also need to make use of the Aculab switch API for establishing the switch matrix connections for data capture. 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 the DSP on a single PMXC module, but may use that DSP to 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 TCP/IP connections

A TCP/IP connection is required between the DSP on the PMXC module and the monitor library, and this must be set up by the application. An API function is provided for the application to setup the necessary connection:

```
acu_ss7mon_connect().
```

4.3 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.

4.4 Link monitoring

In order to monitor a signalling link, the application endpoint requests the DSP software to assign a monitor timeslot. 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()
```

Having assigned a timeslot for monitoring, the application then needs to establish a connection between the DSP timeslot and the source of data to be monitored. This is done using the Aculab switch API functions.

4.5 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.

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.

4.6 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.

4.7 Additional/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.

5 API function reference

5.1 Endpoint management functions

5.1.1 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 the DSP on a single PMXC module. To monitor traffic from multiple cards, a different endpoint should be created for each card, and a different TCP/IP connection established for each endpoint. See also `acu_ss7mon_connect()`.

5.1.2 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
The parameter which is to be modified, followed by additional, parameter-dependent, information. The following values are allowed:

`ACU_SS7MON_CFG_CARD_KEY`
Sets the value of the security key that will be used in the connection with the PMXC. 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_RXBUF_SIZE`
Changes the size of the rx 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 64K, which should suffice for most applications.

`ACU_SS7MON_CFG_DECODE_FLAGS`
Changes the level of decoding that the library will perform automatically. It must be followed by a combination of (bitwise 'or').

```
ACU_SS7MON_MEF_DECODE_MTP3
ACU_SS7MON_MEF_DECODE_USERPART
```

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

`ACU_SS7MON_CFG_PMXC_PORT`
Modifies the TCP/IP port number that will be used for subsequent connections to the DSP. It must be followed by an integer containing the port number. Default is `0x2030`.

`ACU_SS7MON_CFG_L2_CONFIG`
Provides a configuration string that will be used for monitor links set up by this endpoint. It can be modified on a per-link basis using `acu_ss7mon_configure_link()`. Refer to `acu_ss7mon_configure_link()` 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 5.5.

5.1.3 acu_ss7mon_delete_endpoint()

Frees all resources (closing any open 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_connect()

Establishes a TCP/IP connection to a DSP on a PMXC module.

Synopsis

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

Parameters

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

serial_number
ThePMX card's serial number.

The PMX card security key must be configured before this function is called, as described in section 5.1.2.

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 5.5.

5.1.5 acu_ss7mon_get_socket()

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

Synopsis

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

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

Parameters

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

5.1.6 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 5.5.

5.2 Link monitoring functions

5.2.1 acu_ss7mon_monitor_link()

Requests that the DSP allocate a timeslot for data capture.

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 with every indication for this link.

stream and ***timeslot***
Must specify a logical signalling traffic source within the PMX module. For monitoring traffic that is received at the card's network ports, the stream is obtained by adding 32 to the port number. The timeslot is unchanged (see the Aculab switch API guide).

For monitoring transmit traffic, stream and timeslot must be obtained using the switch driver API call `sw_query_output()`, as described in the Aculab switch API guide. This will identify the source of the HLDC traffic that is appearing as an output on the card's network port.

The request is processed asynchronously by the DSP. Successful return from this function indicates that the request has been accepted. If the request is accepted, the DSP will respond by generating a message, which can be subsequently received by the application using `acu_ss7mon_get_msg()`. Within that message, the `mm_type` field will be set to either `SS7MON_MSG_MONITOR_ACK` or `SS7MON_MSG_MONITOR_FAIL`, to indicate success or failure respectively.

The function returns zero to indicate the request was accepted, or a negative value if an error occurred as listed in section 5.5.

5.2.2 acu_ss7mon_unmonitor_link()

Requests that the PMXC code 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 5.2.4).

The function returns zero to indicate success, or a negative value if an error occurred as listed in section 5.5.

5.2.3 acu_ss7mon_con figure_link()

Allows a configuration string to be passes to the DSP that modifies 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 5.2.4).

cfg_string
A NULL terminated string containing one or more space-separated parameter strings. Currently, two such strings are supported:

> `raw_rx_min=nn`
> Specifies the shortest HDLC frame ('nn') that will be passed to the application, frames shorter than this will be suppressed within the DSP. This optimises SS7 MTP2 performance by reducing processing overheads for the continuously repeated 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.
>
> The value of zero is reserved for both of the above parameters and must not be specified.

The function returns zero to indicate success, or a negative value if an error occurred as listed in section 5.5.

5.2.4 acu_ss7mon_get_msg()

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

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 DSP responses, for example following `acu_ss7mon_monitor_link()`, or they may provide pointers to buffers containing data captured from a signalling link.

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.

Upon successful return (if a message is available), the item pointed at by `msg` will have been filled in with a pointer to an `acu_ss7mon_msg_t` structure. This structure is held in a data buffer within the library, and will remain valid until the next call to `acu_ss7mon_get_msg()`, or `acu_ss7mon_isup_reject()`, for the same endpoint. Fields within the `acu_ss7mon_msg_t` structure will be set as described below.

For messages that contain data captured from a signalling link, the following fields will be set:

`mm_link_id`
Indicates the signalling link from which the message was captured, which will be the same value provided when calling `acu_ss7mon_monitor_link()`.

`mm_buffer` and `mm_buflen`
Describes a buffer in the library's data area containing captured line data. The data contents will depend upon the value of the `mm_msg_type` field as explained below.

For all messages, the `mm_msgtype` will be set to one of the following:

`ACU_SS7MON_MSG_NO_DATA`
A timeout occurred waiting for a message, or a received message (line data) was discarded by library owing to filtering conditions.

`ACU_SS7MON_MSG_TRACE`
This is a diagnostic trace message, which may sometimes be seen by the application. They are for Aculab diagnostic purposes and should be ignored by the application.

`ACU_SS7MON_MSG_MONITOR_ACK`
Indicates that the DSP has successfully processed an earlier call to `acu_ss7mon_monitor_link()`.

`mm_link_id`
Will contain the user's token from `acu_ss7mon_monitor_link()`.

`mm_interface_id`
Will contain a value that the application must quote when referring to the link in subsequent API calls such as `acu_ss7mon_configure_link()`, or `acu_ss7mon_unmonitor_link()`. In addition, this field is used by the application to discover the DSP stream and timeslot that have been assigned for monitoring the link as described in section 5.2.5.

`mm_buffer` `mm_buflen`
Will refer to the version string of the DSP software.

`ACU_SS7MON_MSG_MONITOR_FAIL`
Indicates that the DSP encountered an error while processing an earlier call to `acu_ss7mon_monitor_link`. `mm_link_id` will contain the user's token from `acu_ss7mon_monitor_link`, `mm_status` gives the reason for failure.

`ACU_SS7MON_MSG_UNMONITOR_ACK`
Indicates that the DSP has processed an earlier call to `acu_ss7mon_unmonitor_link()`. `mm_link_id` will contain the user's token from `acu_ss7mon_monitor_link()`, `mm_status` indicates whether it was successful.

`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, message, from the first byte after the opening flag to (but not including) the HDLC checksum.

`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 handled by the MTP3 message decoder. `mm_buffer` and `mm_buflen` will describe an MTP3 'payload', from the first byte following the routing label to

(but not including) the HDLC checksum. Refer to section 6 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 7 for further details.

`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 NULL terminated character string containing the error message.

The function returns zero to indicate success, or a negative value if an error occurred as listed in section 5.5.

5.2.5 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 DSP 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 5.2.4).

5.2.6 acu_ss7mon_get_dsp_version()

This function obtains the version string of the DSP 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 software running on the DSP. Will be `NULL` is the endpoint isn't connected to the DSP.

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 ~0 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 5.5.

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 be of `mm_type` `ACU_SS7MON_L1DATA`, with fields valid as described in section 5.2.4.

This function may be used as an alternative to setting the `ACU_SS7MON_MEF_DECODE_MTP3` flag, and allows the application to perform further filtering before deciding that an MTP3 decode is required. Refer to section 6 for a description of messages that have been processed by the library's MTP3 decoder.

If the `ACU_SS7MON_MEF_DECODE_MTP3` flag was set when configuring the endpoint, then MTP3 decode is performed automatically within the library, in which case there is no need to use this function. It returns zero to indicate success, or a negative value if an error occurred as listed in section 5.5.

The function returns zero to indicate success, or a negative value if an error occurred as listed in section 5.5.

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 7).

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'.

cfg
This is not necessarily used within the library, but is passed to the decoder function each time it is invoked as explained below.

decode_fn
The message decoder. 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 function returns zero to indicate success, or a negative value if an error occurred as listed in section 5.5.

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 function provided by `acu_ss7mon_set_upart_decode()` to be invoked for the message. It function 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 may be called automatically within the library, in which case there is no need to use this function.

A return value of zero indicates success, a negative value indicates an error as listed in section 5.5.

5.4 ISUP decoder 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 lib_ref,
                           acu_ss7mon_isup_user_ref_t user_ref)
```

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 5.5.

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 5.5.

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 5.5.

5.5 Error handling functions

5.5.1 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_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 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_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 steps 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_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_pointcode_size(endpoint,1,14) ;
response = acu_ss7mon_set_pointcode_size(endpoint,2,24) ;
response = acu_ss7mon_set_pointcode_size(endpoint,3,24) ;
```

7 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.

7.1 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 8), 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) ;
```

7.2 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 7.1). 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) ;
```

7.3 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 7.5), this 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 ,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.

7.4 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) ;
```

7.5 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 (PMXC), 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 7.3). After the application has processed this event, processing of the new call is resumed and is presented to the application in the usual way.

8 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 7.1. 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.

9 System configuration

In order to use the signalling monitor, each PMXC port that will be used for monitoring must be downloaded with the ss7 firmware file (`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 PMXC network ports should be downloaded with the `ss7.pmx` file and firmware parameters as required for the signalling application(s), no special action is needed to configure the monitor.

If a PMXC network port is to be used exclusively for monitoring, it should be downloaded with the `ss7.pmx` file. In this case, however, 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
```

10 DSP and switch matrix examples

This section provides further details of the procedures for DSP link assignment and PMXC switch matrix initialisation. Each of the three different capture modes, identified earlier in section 2.4, is discussed and accompanied by sample code snippets.

The configuration principles described here can be applied to any and many of the external ports & timeslots of the PMXC. 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 PMXC 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.

10.1 Passive interception example

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

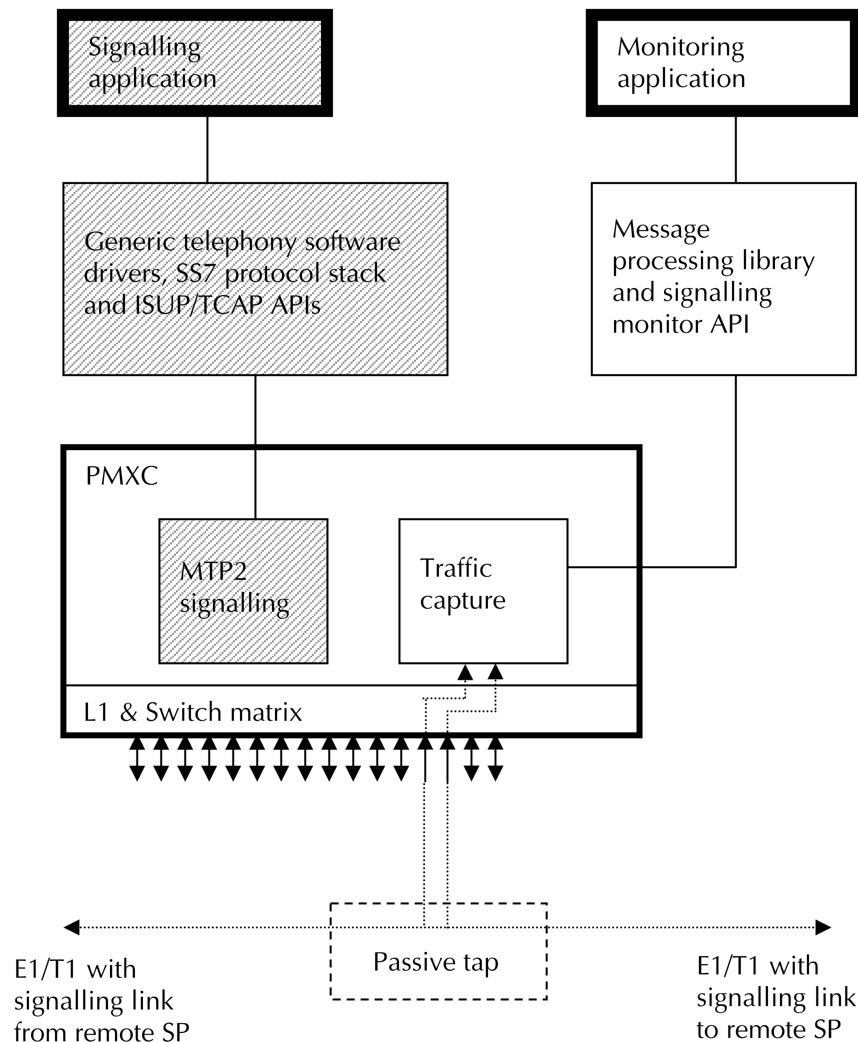

Figure 5: Passive interception example

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);
```

10.2 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 PMXC ports. In addition, the two PMXC ports are patched through to one another in loopback mode, eliminating the need for an external tap.

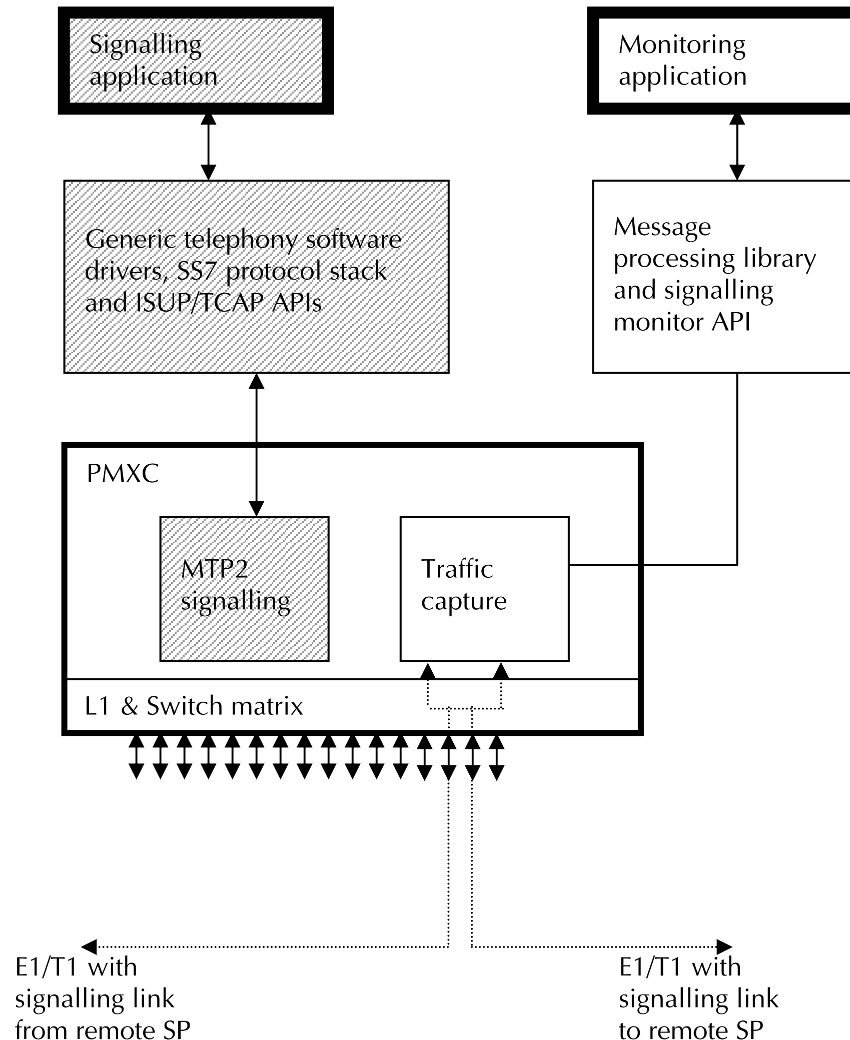

Figure 6: Active interception example

The monitor configuration code for this configuration would be exactly the same steps 1 and 2 of the passive monitoring example, see section 10.1. 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);
}
```

10.3 Local monitor example

In Figure 7, one DSP timeslot is fed from data received at the card's PMXC module 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.

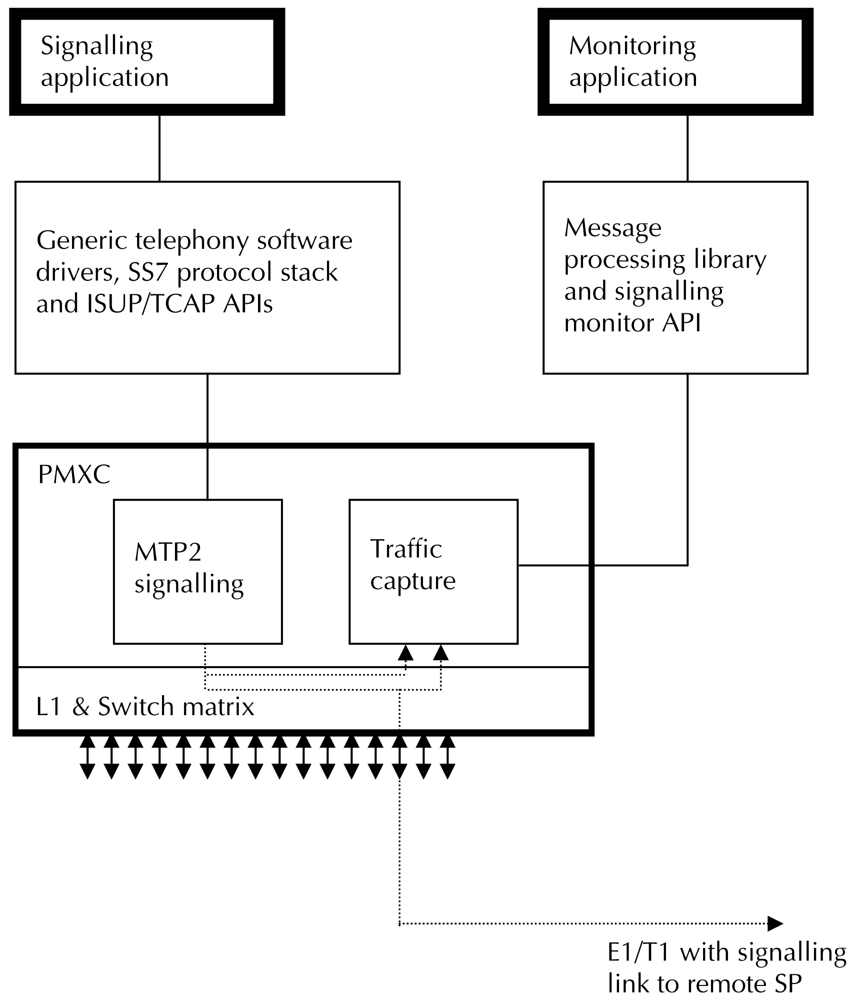

Figure 7: Local monitor example

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);
```

11 Complete application example

This example illustrates a complete monitoring application that identifies and monitors ISUP calls to numbers beginning with '123', as used in the UK for a Speaking Clock service. 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.

11.1 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
```

11.2 Source code description

11.2.1 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 macros for manipulating link ID show just one example of how the token might be constructed, by encrypting the stream and timeslot number into a single integer. The `link_id` is however an application-defined token that may be given any value. There are many alternatives ways of using it, for example it could be an array index that allows the application to resolve a 'link info structure'.

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.

11.2.2 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.

11.2.3 Function: open_card()

This function uses the Aculab resource manager to open a card identified by its serial number. For the sake of simplicity, there is no matching `close_card()`, the card remains open until the application terminates.

11.2.4 Function: initialise_endpoint()

This function allocates an endpoint and connects to the card, using an IP address and security key resolved from the card id using the resource manager function `acu_get_card_info()`. It then sets the appropriate decode flags and pointcode sizes, and enables the library's ISUP decoder.

11.2.5 Function: request_link_monitor()

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

11.2.6 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. `connect_link()` 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.

11.2.7 Function: connect_link()

This is called when the `acu_ss7mon_get_msg()` returns a message that indicates successful allocation of a signalling link. The code provided in the sample makes the connections that would be required for passive interception, which is the simplest case. It could be easily modified for local monitoring or active interception, as explained elsewhere in this document.

The switch API is opened, then `sw_set_output()` is issued to connect the DSP monitor stream and timeslot to the network port and timeslot chosen by the user. The DSP stream and timeslot are derived from the `mm_interface_id` as explained in section 5.2.5, while the network port and timeslot are resolved from `mm_link_id`, using the application's own macros `LINK_ID_TO_PORT` and `LINK_ID_TO_TS`. The switch driver is closed before returning to the caller.

11.2.8 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.

11.2.9 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.

11.2.10 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()`.

11.2.11 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.

11.2.12 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...`'.

11.3 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.

```
#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;

/* Macro for manipulating our own link ids */
#define MAKE_LINK_ID(port,ts)    (((port)<<8) + (ts))
#define LINK_ID_TO_STREAM(id)      ((id)>>8)
#define LINK_ID_TO_TS(id)          ((id)&0x7f)

/* 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)
{
    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) {  /* CPC length should always be 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 unexpected generic %d q763 %d.\n",
                isup_info->ii_generic, 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;
    }
}

/* Make switch matrix connections for DSP to monitor traffic.
 */
static void
connect_link(ACU_CARD_ID card_id,
             unsigned int if_id,
             int network_stream, int network_ts)
{
    ACU_OPEN_SWITCH_PARMS open_switch_parms;
    ACU_CLOSE_SWITCH_PARMS close_switch_parms;
    OUTPUT_PARMS output_parms;
```

```
    int dsp_stream, dsp_ts, resp;

    /* Recover the DSP stream & timeslot from the interface id */
    dsp_stream = acu_ss7mon_interface_id_to_stream(if_id);
    dsp_ts = acu_ss7mon_interface_id_to_ts(if_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) {
        printf("acu_open_switch: Switch API error %d\n", resp);
        exit(EXIT_FAILURE);
    }

    INIT_ACU_STRUCT(&output_parms);

    output_parms.mode = CONNECT_MODE;

    /* The output stream & timeslot connect to the DSP */
    output_parms.ost = dsp_stream;
    output_parms.ots = dsp_ts;

    /* The input stream & timeslot connect to the card's external
     * network ports */
    output_parms.ist = network_stream;
    output_parms.its = network_ts;

    resp = sw_set_output(card_id, &output_parms);
    if (resp !=0) {
        printf("sw_set_output: Switch API error %d\n", resp);
        exit(EXIT_FAILURE);
    }

    INIT_ACU_STRUCT(&close_switch_parms);
    close_switch_parms.card_id = card_id;
    acu_close_switch(&close_switch_parms);
}

/* 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;

    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);
        }

        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:
            process_isup_msg(msg);
            break;

        case ACU_SS7MON_MSG_MONITOR_ACK:
            connect_link(card_id,
                         msg->mm_interface_id,
                         LINK_ID_TO_STREAM(msg->mm_link_id),
                         LINK_ID_TO_TS(msg->mm_link_id));
            break;

        case ACU_SS7MON_MSG_MONITOR_FAIL:
            fprintf(stderr,
                    "Library failed to monitor link for "
                    "network stream %d, ts %d\n",
                    LINK_ID_TO_STREAM(msg->mm_link_id),
                    LINK_ID_TO_TS(msg->mm_link_id));
            exit(EXIT_FAILURE);
        }
    }
}

/* Request the DSP to monitor traffic at a network port
 */
static void
request_link_monitor(acu_ss7mon_ep_t *ep, int port, int timeslot)
{
    int resp;
    int network_stream;

    /* Ref the switch API guide, stream number for network ports is
     * port + 32... */
    network_stream = port + 32;

    resp = acu_ss7mon_monitor_link(ep,
                       MAKE_LINK_ID(network_stream, timeslot),
                       network_stream, timeslot);

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

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

    ep = acu_ss7mon_create_endpoint();

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

    /* Use get_card_info to resolve IP address & security key */
    INIT_ACU_STRUCT(&card_info_parms);
    card_info_parms.card_id = card_id;
    resp = acu_get_card_info(&card_info_parms);
    if (resp !=0) {
        printf("Resource manager error %d from acu_get_card_info\n",
               resp);
        exit(EXIT_FAILURE);
```

```
    }

    /* Check the ip_address is more than an empty string,
     * i.e. that the the connect stands some chance of success */
    if (card_info_parms.ip_address[0] == '\0') {
        printf("No IP address for card\n");
        exit(EXIT_FAILURE);
    }

    /* Connect to the card, using security key from Resource Mgr */
    acu_ss7mon_configure_endpoint(ep, ACU_SS7MON_CFG_CARD_KEY,
                                  card_info_parms.card_key);
    resp = acu_ss7mon_connect(ep, card_info_parms.ip_address);
    if (resp != 0)
        mon_lib_fatal("acu_ss7mon_connect", resp);

    /* Request auto-decode of MTP3 & ISUP */
    resp = acu_ss7mon_configure_endpoint(ep,
                   ACU_SS7MON_CFG_DECODE_FLAGS,
                   ACU_SS7MON_MEF_DECODE_MTP3
                       | ACU_SS7MON_MEF_DECODE_USERPART);
    if (resp != 0)
        mon_lib_fatal("acu_ss7mon_configure_endpoint", resp);

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

    resp = acu_ss7mon_set_upart_decode(ep,
    /* Enable an ITU ISUP decoder for all relations */
                          ~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)
{
    int resp;
    ACU_OPEN_CARD_PARMS open_card_parms;

    INIT_ACU_STRUCT(&open_card_parms);

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

    *card_id = open_card_parms.card_id;
}

int
main(int argc, char ** argv)
{
    ACU_CARD_ID card_id;
    char *serial_num;
    int port_a, ts_a, port_b, ts_b;
    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++;
    port_a = strtol(*argv++, NULL, 0);
    ts_a = strtol(*argv++, NULL, 0);
    port_b = strtol(*argv++, NULL, 0);
    ts_b = strtol(*argv++, NULL, 0);

    /* 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, port_a, ts_a);
    request_link_monitor(ep, port_b, ts_b);

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

    return EXIT_SUCCESS;
}
```