aculab

ACULAB.COM

Aculab SS7

Distributed TCAP API Guide.

MAN1201 Revision 6.16.1

PROPRIETARY INFORMATION

Document Revision

Rev	Date	By	Detail
6.8.3	21.03.07	DSL	First full issue
6.10.1	12.09.08	DSL	Typographical corrections, improved explanations. `acu_asn1_decode_object_id_int/str()` added block and cyclic trace modes added.
6.10 2	16.10.08	DSL	Typographical corrections, `acu_tcap_msg_copy_rx_buffer()` and the ability to send a pre-built tcap message added.
6.10.3	30.10.08	DSL	Minor clarifications.
6.11.0	10.09.10	DSL	Message based ASN.1 codec, additional configuration parameters and support functions.
6.11.3	25.03.11	DSL	Minor corrections.
6.11.11	01.11.11	DSL	Additions and clarifications to message based codec functions.
6.12.2	28.06.13	DSL	Remove references to Solaris. Support IPv6 connections to the ss7 driver.
6.13.0	10.10.14	DSL	Typographical corrections and clarifications.
6.14.0	15.09.16	DSL	Align with the 6.14.0 software release.
6.15.1	31.08.18	DSL	Align with the 6.15.1 software release.
6.15.7	23.07.19	DSL	Addition of transaction restore functions.
6.15.8	16.08.19	DSL	Add `acu_tcap_transaction_restore_app_ctx()`
6.16.0	30.03.22	DSL	Align with the 6.16.0 software release.
6.16.1	13.02.23	DSL	Update title page

CONTENTS

1 Introduction

This document describes the Distributed TCAP API.

1.1 Structure of TCAP

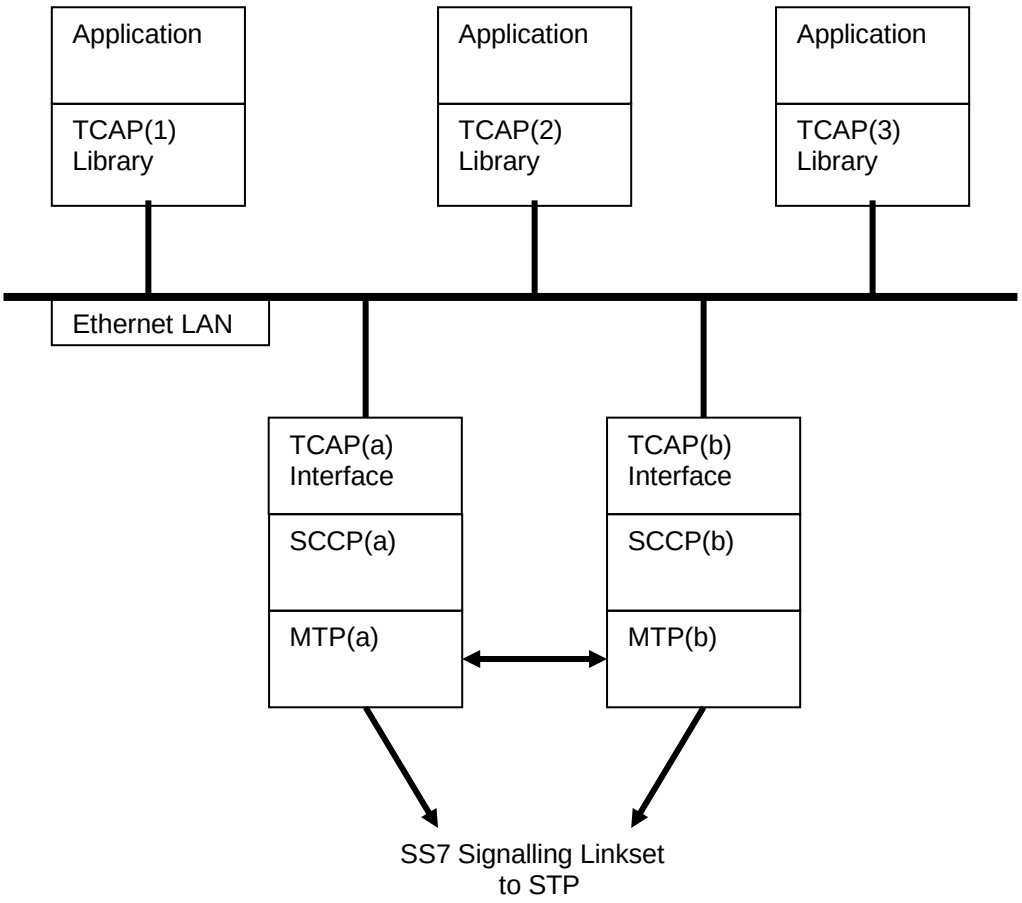
The Aculab Distributed TCAP has the SCCP code tightly coupled with MTP3 (or M3UA) and a separate TCAP library that is linked to the application. The TCAP library communicates with SCCP using a proprietary protocol over TCP/IP and a small TCAP interface driver.

The product supports:

- Multiple TCAP applications using different SSNs.
- Structured and unstructured TCAP dialogues.
- A TCAP application connecting to multiple SCCP endpoints (e.g.: several SSNs or multiple MTP3 local pointcodes).
- TCAP applications in a dual-MTP3 environment
- Multiple copies of the same TCAP application running on multiple chassis.
- Multiple TCAP applications running in a single chassis.
- ITU and ANSI TCAP, including ITU TCAP over ANSI SCCP and vice versa.

1.2 TCAP with dual resilient MTP3

The diagram below shows the components of a dual resilient MTP3 system.



The TCAP interface, SCCP and MTP3 code all reside in the kernel. The application is shown running in a different system, but can run in the same system as SCCP and MTP3 by connecting to 'localhost'.

SCCP must pass inward messages to the correct application. This is done by allocating different ranges of TCAP transaction identifiers to each application. Begin (ANSI Query) and Unidirectional messages are given to each application in a round-robin fashion.

1.3 TCAP library data structures

The TCAP library defines the following major data structures:

- The ssap structure (SCCP service access point). One of these must be created for each SSN. The TCP/IP connection to SCCP is controlled from this structure. Everything refers (directly or indirectly) to an ssap structure. An application would normally only create a single ssap structure.
- The transaction structure. This holds all the information for a TCAP message exchange with the TCAP peer. Multiple transaction structures can be created on each ssap.
- The msg structure. This is used for passing TCAP messages between the application and the library.
- The sccp_addr structure. This is used to hold address information.

All the fields of the ssap and transaction structures are private to the library.

1.4 Functional Overview

A TCAP application should perform the following steps:

- Create a ssap structure and initialise the configurable fields (from a configuration file or directly).
- Connect to the SCCP systems.

If the application is going to initiate a TCAP transaction:

- Create a transaction structure.

If the application is a server, wait for the first message; the library will allocate a transaction area.

- Allocate a message, build and send it to SCCP.
- Wait for a response on either the ssap queue, or transaction queue (the response is added to both).
- When all messages have been sent/received, delete the transaction.

All messages must be explicitly deleted by the application.

Unidirectional dialogues work the same way, except that all inbound messages are queued on a single transaction. Unidirectional messages can be sent from that transaction, or from another transaction area.

An application can create any (reasonable) number of transactions.

1.5 Thread safety

The library can be used by threaded programs. The data structures are protected by a per-ssap mutex.

The transaction functions only acquire the mutex to protect the integrity of the data structures, not the transaction state. The application should not allow concurrent processing of a single transaction by more than one thread.

Whenever a message is removed from the ssap queue `acu_tcap_trans_block()` is implicitly called. This allows multiple threads to process messages from the ssap queue. The application must call `acu_tcap_trans_unblock()` after processing each message. Alternatively, a single threaded application can set the `SINGLE_THREADED` configuration option.

The `acu_tcap_trans_block()` and `acu_tcap_trans_unblock()` functions can also be used to stop messages being read from the ssap queue during other application processing for the transaction.

1.6 Restarting applications

Some applications have long lasting transactions that they may want to recover after an unexpected application restart.

The TCAP library has support for applications reattaching to the driver and restoring transactions. The application is responsible for saving all the relevant state.

At least the following must have been saved for each transaction:
- The local and remote transaction ids.
- The remote SCCP address.
- The invoke-id of any pending outbound operations (sent invoke components).
- Additional application specific state.

In addition to restoring the transactions it is necessary to stop the TCAP driver code from sending P-Aborts for unknown local transactions ids while the application isn't running.

It may also be necessary to minimise the number of messages that get discarded (application level retries may recover from discards). The driver code can hold onto its queue of receive message, but any already read by the library code will be lost. The number of messages held by the library can be reduced – although it will reduce attainable throughput.

The following configuration options control the reattach and restore:

`ACU_TCAP_CFG_REATTACH_TIMEOUT=`*`timeout_seconds`*
This parameter specifies the length of time the driver waits for an application to reattach before deleting its data and queued messages.
If non-zero and `ACU_TCAP_CFG_TRANID_RANGE` non-zero `acu_tcap_ssap_connect_sccp()` will reattach to the old data area and be given any queued messages.
Changes to this parameter are copied to the driver; it is recommended that it be set to zero before normal program exit.

Note Setting the timeout to longer than that used for invokes may have undesirable side effects.

`ACU_TCAP_CFG_TRANID_RANGE=`*`local_id_range`*
This parameter requests the driver allocate local transaction ids that have their top 12 bits set to *`local_id_range`* rather than an unused range.
This must be set before calling `acu_tcap_ssap_connect_sccp()`.

`ACU_TCAP_CFG_REATTACH_DISCARD=`*`n/y/1`*
If set to y (or 1) the driver will discard all queued messages while waiting for the reattach timer to expire. Without reattach it would send P-Aborts.
Changes to this parameter are copied to the driver.

`ACU_TCAP_CFG_TRAN_RESTORE=`*`n/y/1`*
If set to y (or 1) the library will create transactions when unexpected `CONTINUE` or `END` messages are received (rather than send a P-Abort).
The transactions are created as if this is the first backwards message.
This lets the application decide on the required response.
If the application uses `acu_tcap_trans_set_userptr()` on transactions it knows about, it can detect these transactions because `acu_tcap_trans_get_userptr()` will return `NULL`.

Once the restarted application has reconnected to the driver with the correct transaction id range (i.e. received the `ACU_TCAP_MSG_CON_STATE` message indicating the connection is active) it must create the TCAP library data area of each transaction.
- Call `acu_tcap_transaction_restore()` to create the transaction data area.
- Set the remote SCCP address in the same way as for an outward transaction.
- Call `acu_tcap_operation_restore()` for every outstanding outward operation (sent invoke).

The application can then proceed in the normal way, retrieving messages from the ssap or transaction queues.

1.7 Transmit and receive flow control

Since networks don't have infinite bandwidth and processors have finite speed there will always be places where large numbers of messages are likely to get queued. Some kind of mitigation is needed to stop these queues growing indefinitely and using all system memory.

For TCAP applications messages can be queued in kernel space (in the ss7 driver) or in user space (by the TCAP library). It is generally better to queue in user space as overlarge queues have a smaller impact.

The TCAP library uses a separate thread to transmit and receive data, so messages can be transferred to/from the driver even when the application isn't making calls into the TCAP library.

Most applications don't need to worry about the inner workings or any the configuration parameters. The defaults should be fine.

1.7.1 Monitoring queued messages

Sometimes it is useful to know where (or even if) messages are being queued within the protocol stack. While queued messages can be just be caused by a burst of data being processed, sometimes they show up processing bottlenecks and deadlocks.

Flow control in the driver works by limiting the number of messages that each 'source' can allocate.

Run `ss7maint osstatus -m` to see how many messages are allocated. If the limit is reached (some message pools have no limit – shown as '`Free == 0`') the count of `Fails` is increased, this doesn't always mean messages are discarded – eg acks to the TCAP library just get delays.

Run `ss7mant osstatus -q` to see where messages are queued. These are actually schedulable entities, sometimes messages do get queued elsewhere. The `Nobuf` column doesn't relate to the number of messages queued, it is the number of times the entity has been scheduled because messages have become available is a message pool (this is the mechanism used to send acks to the TCAP library).

Run `ss7maint tcapstatus -A` to get statistics from the tcap driver code.

Provided the connection between the driver and library is working it is also possible to get statistics from the library by running `ss7maint tcapstatus -labT`. While this includes the number of queued transmit messages it doesn't indicate the total number of queued receive messages only the number on each transaction.

1.7.2 TCP connection flow control

The TCAP library always uses a TCP connection to pass information to and from the SS7 driver. Application level acks are used for data messages to ensure this connection never blocks (in either direction) so that keepalives and status requests are always processed.

The sender controls the flow by requesting an ack (say) every 8 messages. Only one such request can be outstanding, so after requesting an ack and sending 7 more messages it must wait for the ack before sending any more data. The receiver will send an ack if it has enough buffer space for (in this case) 15 messages.

An additional benefit of a relatively small windows is that on receipt of an ack multiple messages can be sent in a single request and processed more efficiently by the network stack – it may end up in one or two Ethernet frames instead of each message being in its own frame.

1.7.3 Transmit messages

The ss7 driver limits the number of transmit messages it will queue for each ssap to 140 (not configurable). Once that limit is reached it stops sending acks to the library and the library will stop sending it more data. Any further transmit requests from the application are queued within the library.

The driver queues the messages from the TCAP application waiting for MTP2/M2PA transmit window, M3UA/M2PA licences, M3UA socket buffer space and MTP3 timed control diversion/changeback (messages for MTP2 retransmit and MTP2/M2PA retrieval are accounted for separately). As soon as queued messages get freed the library will send more to the driver.

While 140 may seem small, sending that many 200 byte messages over a 64k MTP2 signalling link takes around 4 seconds. If a lot of messages might get queued on slow links the application operation timeouts may need to allow for this delay.

The library will request an ack after either 24 data messages (not configurable) or 2920 bytes (configurable as "`tx_byte_window`" maximum 32768) of data message have been sent. Increasing `tx_byte_window` may improve the maximum message rate especially if fragmented XUDT are being generated (24 messages are otherwise unlikely to exceed 8k bytes).

The library doesn't limit the size of its transmit queues (one for each server connection), but does generate the status indication `CON_STATE_TX_FLOW` if the length exceeds 16 messages (configurable as "`tx_queue_len`" maximum 10000) and clears the indication when the queue length reduces to half this value.

While configurations that use MTP2 links probably don't want to queue many transmit messages in userspace (to avoid the delays caused by the physical link speed), configurations that use M3UA or M2PA may need to significantly increase "`tx_queue_len`" to avoid repeated `CON_STATE_TX_FLOW` indications.

A TCAP application that initiates requests (eg sending SMS) can limit the number of transmit messages queued by controlling the number of outstanding operations. This may also be needed to avoid overloading the target system(s).

1.7.4 Receive messages

Whereas transmit messages get queued data waiting 'line time' to transmit, receive messages are queued to allow bursts of messages by processed by the application. If the application can't keep up then messages have to get discarded somewhere. Hopefully any timeout+retry doesn't cause a catastrophic increase in the number of messages being received.

The library reads messages from the driver into a (usually) large cyclic buffer (configurable as "`rx_buflen`" default 130944). It will send an ack if there is enough space for two windows full of data from the driver (actually smaller of 64k and half the buffer size). The driver will queue up to 250 (configurable as "`rx_max_qlen`" in the TCAP library or as "`rx_qlen`" in the driver) on each connection (library ssap) before tracing and discarding receive messages.

Although the driver is responsible for requesting acks for receive messages, the parameters are configured by the library with the driver being updated whenever they are changed. Acks for receive data are requested after either 24 messages (configurable as "`rx_msg_window`") or 2916 bytes (configurable as "`rx_byte_window`") have been sent. Setting either value to 1 causes an ack to be requested for every message.

Note **With the default values the library receive buffer never gets anywhere near being full.**

Under normal conditions it is expected that all receive messages will get read into the libraries receive buffer and messages will only be transiently queued by the driver.

Very bursty high throughput systems may need larger receive buffer space (usually) in the library.

The most likely problem to affect the receive processing is that receive messages given to the application contain pointers into the main receive buffer area (the data and SCCP addresses are not copied). While the application can free messages in any order, the buffer space beyond an allocated message cannot be reused. If the application fails to free a messages (or doesn't free one for a long time) then further messages cannot be received.

1.7.5 Minimising library buffering of receive messages

Although buffering receive messages in userspace maximises throughput and minimises

latency there are some configurations where it isn't desirable. In particular an application is going to restore transactions after an unexpected restart you don't want a lot of messages to have been lost because they were buffered in userspace.

The following TCAP library configuration options will force almost all the receive messages to be queued by the driver:

`rx_msg_window=1`
Force the driver to request an ack for every received message.

`rx_byte_window=1`
The library ensures the receive buffer is large enough for two windows full of data. Reduce the byte window size to allow a very small receive buffer be allocated.

`rx_buflen=512`
Allocate a receive buffer that is only really big enough for a single message.

- **If segmented XUDT are received the buffer will need to a larger.**

`rx_max_qlen=nnn`
Increase the number of messages the driver will queue to allow for bursty data.

With the parameters above it is unlikely that the library will send an ack until the application calls either `acu_tcap_msg_free()` or `acu_tcap_msg_copy_rx_buffer()` to free the space in the receive buffer.

2 API Functions

2.1 TCAP API functions

2.1.1 Abbreviations and nomenclature

The following are used:

component	the component part of a TCAP message
comp	component
conv	conversation (ANSI message type)
condvar	condition variable
dialogue	the dialogue portion of a TCAP message
dlg	dialogue
pdu	protocol data unit
ssap	SCCP service access point
transaction	a set of messages using the same transaction-id
trans	transaction

Note **Q.771 especially section 3.1 uses the word dialogue for a sequence of messages. This implementation doesn't have separate component and transaction sublayers and uses 'transaction' throughout to avoid confusion between message dialogues and the dialogue portion of a message.**

2.1.2 TCAP Header files

All the definitions start `acu_tcap_` or `ACU_TCAP_` (or similar) in order to avoid polluting other namespaces.

The definitions are all in C, but can be used from C++ applications.

Note **A significant amount of pre-processor 'magic' is used to avoid replicating information (See Appendix E:).**

2.1.2.1 tcap_api.h

This header file contains all the definitions for the TCAP API.

The majority of the structures are described with the function that uses them; additional information is in Appendix B:

2.1.2.2 tcap_asn1_codec.h

This header file contains the definitions for the ASN.1 BER encoder/decoder. See Section 2.2.1 for more information.

2.1.2.3 tcap_synch.h

This header file contains the definitions for the synchronisation and thread-pool functions.

2.1.2.4 Protocol message definitions

The following headers contain some definitions for the message based ASN.1 codec (see 2.3), they are installed into `$ACULAB_ROOT/ss7/sample_code/tcap/asn1` rather than `$ACULAB_ROOT/include`.

`tcap_defn.h`	ASN.1 definitions for TCAP.
`acu_map_common_asn1.h`	General MAP (Mobile Application part) message.
`acu_map_sms_asn1.h`	MAP SMS messages.

The definitions in `tcap_defs.h` will encode and decode normal TCAP messages; however the library doesn't use them.

2.1.3 Configurable parameters

TCAP's configurable values can either be read from a configuration file when an ssap is created, or set directly on the ssap or transaction by function call.

Whenever a transaction is created, it gets a copy of its configuration information from its ssap.

Once the TCAP application has connected to the SS7 driver, parameters can also be changed using `ss7maint tcapconfig`. This is particularly useful for changing the trace parameters.

Configurable parameters can be placed into three groups: global ssap parameters, general transaction parameters, and address parameters.

When calling the functions to set configuration item, the names below must be preceded by `ACU_TCAP_CFG_` (e.g. `ACU_TCAP_CFG_REMOTE_PC`).

2.1.3.1 Global ssap parameters

These include trace control and the connection to the SS7 driver:

Name	Type	Default	Description
`LOGFILE`	string		Name of logfile to open.
`LOGFILE_MAX_SIZE`	integer	`1000000`	Size (bytes) before logfile rotated.
`LOGFILE_APPEND`	boolean	`no`	Append to existing logfile.
`LOGFILE_OLD_KEPT`	integer	`5`	Number of old logfiles kept.
`LOGFILE_FLOCK_INDEX`	boolean	`yes`	flock() logfile.index during log rotation.
`TRACE_TAG`	string	`pid:`*`nnn`*	Name for trace entries.
`TRACE_BUFFER_SIZE`	integer	`32768`	Size of cyclic trace buffer.
`TRACE_MODE`	integer	`0`	Determines when trace buffer is written to file, see section 2.1.4
`SERVER`	boolean	`no`	Process inward `BEGIN` messages.
`UNI_SERVER`	boolean	`no`	Process inward `UNIDIRECTIONAL`.
`SINGLE_THREADED`	boolean	`no`	Block on removing messages from ssap queue is not applied.
`TRANID_RANGE`	integer	`0`	Specifies an explicit value to the high 12 bits of the transaction ID. `0` requests an unused range be allocated.
`TRAN_RESTORE`	boolean	`no`	Restore transactions from unexpected inward `CONTINUE` and `END` messages.
`REATTACH_TIMEOUT`	integer	`0`	Time (seconds) the driver will wait for an application to reattach. If non-zero and `TRANID_RANGE` non-zero attempt to reattach.
`REATTACH_DISCARD`	boolean	`no`	If set messages received while waiting for reattach will be discarded.
`NI`	integer	from mtp3	Network Indicator.
`TRACE_LEVEL_ALL`	integer	`5`	Set all trace levels.
`TRACE_LEVEL_xxx`	integer	`5`	Set trace level for source '`xxx`', see section 2.1.4.
`TRACE_LEVEL(n)`	integer	`5`	Set trace level for source '`n`'.
`HOST_A_NAME`	string	`127.0.0.1`	Name and IP addresses of host A (see below).
`HOST_A_PORT`	integer	`8256`	TCP/IP port number.
`HOST_A_PASSWORD`	string		Password for host A.
`HOST_B_NAME`	string		Name and IP addresses of host B (see below).
`HOST_B_PORT`	integer	`8256`	TCP/IP port number.
`HOST_B_PASSWORD`	string		Password for host B.
`RX_BUFLEN`	integer	`130944`	Size (bytes) of TCP/IP receive buffer.
`TX_QUEUE_LEN`	integer	`16`	Number of TCAP messages queued before transmit flow control reported.
`KEEPALIVE_TIMEOUT`	integer	`10`	Seconds between keepalives, set to zero to disable keepalives.
`CONNECT_TIMEOUT`	integer	`10`	Timeout (seconds) for TCP/IP connection establishment.

TX_BYTE_WINDOW	integer	2920	Number of data bytes sent to driver before an ack is requested.
RX_MSG_WINDOW	integer	16	Maximum number of messages the driver will send before an ack is requested.
RX_BYTE_WINDOW	integer	2916	Maximum number of bytes the driver will send before an ack is requested.
RX_MAX_QLEN	integer	250	Maximum number of messages the driver will queue before discarding messages.

Enclose string parameter values that contain spaces (or other special characters) in double quotes.

If the SERVER or UNI_SERVER options are changed after the connection to the driver is made, then the driver is informed of the new value. This allows one node of a distributed application to gracefully shutdown.

The HOST_A_NAME and HOST_B_NAME fields consist of a hostname optionally followed a comma separated list of numeric IP addresses (IPv4 or IPv6). If there are no numeric addresses getaddrinfo() is called to resolve the hostname to a list of addresses, otherwise the hostname is ignored unless it is a valid numeric IP address. The returned addresses are tried in turn when connecting to the server.

2.1.3.2 General transaction parameters

These are settable on both ssap and transactions; transactions inherit the values from the ssap:

Name	Type	Default	Description
QOS_RET_OPT	boolean	no	SCCP 'return on error'.
QOS_SEQ_CTRL	boolean	no	SCCP 'sequential delivery (class 1)', if enabled the 'sls' value is taken from the bits of the local transaction id.
QOS_PRIORITY	integer	~0u	SCCP 'message priority'. ~0u requests the default of 0 for ANSI and absent for ITU.
QOS_RESPONSE_PRI	integer	~0u	SCCP 'message priority' for response messages, default (~0u) is 1 for ANSI and absent for ITU.
OPERATION_TIMEOUT	integer	60	Default operation timeout (seconds).
ADD_TCAP_VERSION	boolean	no	Add protocol version parameter to dialogue messages.
REVERSE_ROUTE	boolean	no	Send messages back to the point code from which they came, ignoring global title translation in the local SCCP.
RESPOND_RX_LOC_GT	boolean	no	Respond to a BEGIN using the received called party address.
STICKY_CON	boolean	no	Try to only use the same Host (A or B) for all messages.
ENC_DEF_LEN	boolean	no	Use the definite length encoding for all ASN.1 constructed items. Needed to encode full length SMS.
PREFERRED_MAXLEN	integer	238	Use definite length ASN.1 if it would reduce the TCAP data below the specified size.

2.1.3.3 Address parameters

The local and remote (replace LOCAL with REMOTE) address parameters are settable on ssaps and transactions; transactions inherit the values from the ssap. See section B.2 for further details:

Name	Type	Description
LOCAL_FLAGS	integer	Address flags.
LOCAL_GTI	integer	Global Title Indicator.
LOCAL_SSN	integer	SSN.

LOCAL_PC	integer	SCCP address pointcode.
LOCAL_RL_PC	integer	MTP routing label pointcode (received messages only).
LOCAL_TT	integer	Translation Type.
LOCAL_NP	integer	Numbering Plan.
LOCAL_ES	integer	Encoding Scheme.
LOCAL_NAI	integer	Nature of Address Indicator.
LOCAL_GT_DIGITS	BCD	Global Title digits.

The local SSN must be set before the connection to the driver is established, and should not be changed. The other values can be changed at any time.

The eight address fields (GTI, SSN, PC, RL_PC, TT, NP, ES and NAI) have a 'data valid' bit set whenever they are set via the configuration interface. This bit can be cleared by setting the parameter CLEAR_LOCAL_SSN (etc) to an empty string. This might be needed to stop SCCP including the parameter (e.g. the local ssn) in a message.

The addresses can also be modified by calling **acu_tcap_ssap/trans_get_loc/remaddr()** and directly modifying the structure.

For ANSI/China networks the pointcodes can be specified in 8-8-8 format, although they are currently always traced in decimal.

Note **The addresses are added to a message when acu_tcap_msg_init() is called, not when it is sent.**

Note **The transaction's configured values for the remote address are overwritten with the actual remote address when the first backwards message arrives.**

2.1.3.4 Configuration file format

The TCAP configuration file has a similar format to that of the ss7 protocol stack. It should contain a single block of configuration data bracketed between [TCAP] and [endTCAP]. Each line inside the configuration block has the format '*parameter = value*', where *parameter* is one of the configurable parameter names, and *value* is the required value.

Comments can be added to any line by preceding the comment with a '#' character. Blank lines are ignored. The lines before [TCAP] and after [endTCAP] are currently ignored, but this isn't guaranteed as additional sections may be added at some later release.

The parameter names can be specified in upper or lower case. For compatibility with other parts of the Aculab SS7 protocol stack, the configuration file can contain localxxx and remotexxx instead of local_xxx and remote_xxx.

For example:

```
[TCAP]
    trace_tag = program_name
    logfile_append = y
    logfile = tcap2020.log
    localpc = 2020
    localssn = 27
    remote_pc = 7070
    remote_ssn = 143
    server = y
    host_a_name = sccp_host_a,192.168.1.1
    host_a_password = tcap_password
    host_b_name = sccp_host_b,192.168.1.2
    host_b_password = tcap_password
[EndTCAP]
```

The SS7 stack configuration file on sccp_host_a (that for sccp_host_b is similar) needs to contain the following:

```
[SP]
    LocalPC = 2020
    [TCAP]
        password = tcap_password
    [EndTCAP]
```

```
    [SCCP]
        master = y
    [EndSCCP]
    [MTP3]
        [DUAL]
            host = sccp_host_b
            ipaddresses = 192.168.1.2
            master = y
            listen = 14
            connect = 15
            password = dual_password
        [EndDUAL]
        [DESTINATION]
             RemotePC = 7070
        [EndDESTINATION]
    [EndMTP3]
[EndSP]
```

2.1.4 Tracing

The TCAP library contains extensive tracing of the API calls and the interface to SCCP. Each trace call specifies a trace source (0 to 63) and trace level (0 to 15). The level of trace output can be set separately for each trace source from the application configuration file, from the program by calling `acu_tcap_ssap_set_cfg_int()`, or from the command line by running `ss7maint tcapconfig`.

Tracing starts when the `LOGFILE` parameter is set for the ssap.

By default the trace buffer is written to the logfile after each trace entry is complete. This can be modified by setting `TRACE_MODE` to `ACU_TCAP_TRACE_MODE_BLOCK` (`1`) or `ACU_TCAP_TRACE_MODE_CYCLIC` (`2`). In block mode the buffer is written when full, in cyclic mode the buffer just wraps (discarding trace entries). The buffer is always written when a message with trace level 0 or 1 is written, or when the trace mode is set (even if the value doesn't change).

The logfile is always opened in 'append' mode (although it may be truncated). On Linux systems this allows multiple programs and ssaps to log to a common file.

Note **On Windows systems, using a common log file can lead to corrupted log entries.**

If the size of the logfile in bytes exceeds the `LOGFILE_MAX_SIZE` parameter, then a new logfile *`logfile`*`.1` (et seq) is opened. The number of old logfiles is restricted to `LOGFILE_OLD_KEPT` (default is 5). The sequence number of the current logfile is kept in *`logfile`*`.index`.

On Linux systems the logfile rotation uses `flock()` (on the index file) to maintain consistency between multiple applications. Some NFS file systems block the flock() call indefinitely, it can be disabled by setting `LOGFILE_FLOCK_INDEX` to `0`.

The logfile is formatted so that '`ss7maint decode`' can be used to pretty-print the tcap messages.

Functions are supplied so that the application can trace messages to the library log file.

Note **The default level of tracing has a significant performance penalty.**

Trace sources:

`APPLICATION(0)` to `APPLICATION(15)`	`0x00` to `0x0f`	16 trace sources available for application use
`API_ENTRY`	`0x10`	Entry to API routine (not all functions make trace calls)
`API_EXIT`	`0x11`	Normal exit from API function
`API_ERROR`	`0x12`	Error exit from API function (might be an internal function)
`API_EVENT`	`0x13`	Significant event
`API_INFO`	`0x14`	Additional information
`API_CONFIG`	`0x15`	Configuration changes
`API_OP_TIMER`	`0x16`	Operation state engine and timers
`ENCODER`	`0x20`	TCAP message encoder
`DECODER`	`0x21`	TCAP message decoder
`TCP`	`0x30`	TCP/IP connection establishment and control
`TCP_SEND`	`0x31`	TCP/IP messages being sent
`TCP_RECV`	`0x32`	TCP/IP messages being received

Other values are reserved for future use.

Setting `TRACE_LEVEL_ALL=14` during development may help identify application bugs.

2.1.4.1 acu_tcap_trace/trace_v/trace_buf

```
void acu_tcap_trace_v(acu_tcap_ssap_t *ssap, unsigned int flags,
     const void *buf, int buf_len, const char *fmt, va_list ap);
void acu_tcap_trace(acu_tcap_ssap_t *ssap, unsigned int flags,
     const char *fmt, ...);
void acu_tcap_trace_buf(acu_tcap_ssap_t *ssap, unsigned int flags,
     const void *buf, int buf_len, const char *fmt, ...);
```

Purpose

These functions output text to the trace buffer, `acu_tcap_trace_buf()` adds a hexdump of `buf` following the text output.

Parameters

`ssap` The ssap structure the trace is for.

`flags` Usually `ACU_TCAP_TRF(part, source, level)`
or `ACU_TCAP_TRFF(part, source, level, format)`

- `part` One of `FIRST`, `MIDDLE`, `LAST` or `ONLY` indicating which part of the trace entry is being generated.
- `source` `APPLICATION(n)` for n between 0 and 9, identifying the source of the trace.
- `level` 0 to 15 indicating the level (high number for more verbose trace) of this call, the default is usually 5.
- `format` Buffer format for `ss7maint decode`, one of:
  - `TCAP` Complete ASN.1 BER encoded TCAP message.
  - `ASN1` Any ASN.1 BER encoded ASN.1 buffer.

`buf` Address of buffer area to hexdump following the format output.

`buf_len` Number of bytes to hexdump.

`fmt` `printf` format for trace arguments.

`ap` Variable argument list for underlying `printf` call.

The `flags` parameter specifies the trace source and level and also indicates which part of a trace entry is being generated (allowing a single trace entry to be generated by multiple calls to the trace functions). A short header including the system time is output at the start of each trace entry. The trace is locked while a trace entry is generated (i.e. from the call specifying `FIRST` to that specifying `LAST`) to avoid trace output from different threads being intermixed – even when multiple threads try to write concurrently to the same log file.

The trace is output if the `level` in the call is less than that set using `acu_tcap_ssap_set_cfg_int()` for the same source.

Note **The trace is formatted by a fast local version of `snprintf()` which does not support floating point format specifiers.**

2.1.4.2 acu_tcap_trace_error

```
int acu_tcap_trace_error(acu_tcap_ssap_t *ssap, const char *fname, int rval,
     const char *fmt, ...);
```

Purpose

This function is used to generate a trace entry when one of the TCAP error codes is generated.
It is loosely equivalent to calling `acu_tcap_trace()` with `flags` of `ACU_TCAP_TRF(ONLY, API_ERROR, 5)`.

Parameters

`ssap`	The ssap structure the trace is for.
`fname`	The name of function that is returning the error.
`rval`	TCAP error number (one of `ACU_TCAP_ERROR_xxx`).
`fmt`	`printf` style format string, followed by the arguments.

Return value

Always `rval`.

2.1.4.3 acu_tcap_strerror

```
const char *acu_tcap_strerror(int rval, unsigned int flags);
```

Purpose

This function returns a text string that describes a TCAP library error code.

Parameters

`rval`	TCAP error number (one of `ACU_TCAP_ERROR_xxx`).
`flags`	0 => return descriptive text, see B.1. 1 => return the C name `"ACU_TCAP_ERROR_xxx"`.

Return value

A pointer to a static const string describing the error, unless the error number is unknown in which case the address of a static array filled with the text "`error %d unknown`" is returned.

The error text strings are defined by the `ACU_TCAP_ERRORS` define in `tcap_api.h`.

2.1.5 SCCP access functions

2.1.5.1 acu_tcap_ssap_create

```
acu_tcap_ssap_t *acu_tcap_ssap_create(const char *cfg_file,
    acu_tcap_ssap_flags_t flags);
```

Purpose

This function creates a new SCCP access point without establishing the connection to SCCP. The application may set parameters from its own configuration information before the connection to SCCP is established.

Parameters

`cfg_file` Name of the configuration file to use, may be `NULL`
If the file cannot be opened, and the name doesn't contain a '/' (or '\') then the library will look for the file in the directories `${HOME}` and `${ACULAB_ROOT}/ss7`

`flags` Bitwise OR of:

`ACU_TCAP_ITU`	Use ITU TCAP message formats.
`ACU_TCAP_ANSI`	Use ANSI TCAP message formats.
`ACU_TCAP_SERVER`	Application is a server process and will be given new transactions (i.e.: inward `BEGIN/QUERY` messages).
`ACU_TCAP_UNI_SERVER`	Application will be given inward `UNIDIRECTIONAL` messages.
`ACU_TCAP_STATUS_IND`	The application will be given all the status indications from SCCP.
`ACU_TCAP_LOG_STDERR`	Write initialisation errors to `stderr`.

The `SERVER`, `UNI_SERVER` and `LOG_APPEND` flags can also be set from the configuration.

Return value

The address of an initialised `acu_tcap_ssap_t` structure, or `NULL` if `malloc()` fails or the configuration file cannot be accessed.

2.1.5.2 acu_tcap_ssap_delete

```
void acu_tcap_ssap_delete(acu_tcap_ssap_t *ssap);
```

Purpose

This function deletes a SCCP access point, and any TCAP transactions created on it.

Parameters

`ssap` The address of the `acu_tcap_ssap_t` structure to delete.

Return value

`None.`

2.1.5.3 acu_tcap_ssap_connect_sccp

```
int acu_tcap_ssap_connect_sccp(acu_tcap_ssap_t *ssap);
```

Purpose

This function causes the TCAP library to try to establish a TCP/IP connection between the ssap and the SCCP driver code.

The local SSN and POINTCODE must be set before this is called.

After this function completes the TCP connection attempt continues asynchronously, and it may subsequently succeed or fail and be automatically retried. When the connection attempt completes, a message of type `ACU_TCAP_MSG_CON_STATE` will be sent to the ssap, indicating a state transition. When that message is seen, the application can check the ssap connection state, using `acu_tcap_get_con_state()`, to see whether the connection was successfully established.

Note **TCAP transactions cannot be created until the connection to SCCP has been established.**

Parameters

`ssap` The address of the `acu_tcap_ssap_t` structure to connect to SCCP.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.5.4 acu_tcap_ssap_set_cfg_int/str

```
int acu_tcap_ssap_set_cfg_int(acu_tcap_ssap_t *ssap,
     acu_tcap_cfg_param_t param, unsigned int i_val);
int acu_tcap_ssap_set_cfg_str(acu_tcap_ssap_t *ssap,
     acu_tcap_cfg_param_t param, const char *s_val);
```

Purpose

These functions set a configurable value of the ssap.

Integer parameters can be set using either function.

Refer to section 2.1.3 for a list of configurable parameters.
Transactions inherit their configuration from the ssap.

Parameters

`ssap` The address of the `acu_tcap_ssap_t` structure to modify.
`param` Configuration parameter to modify.
`i_val` Integer value for parameter.
`s_val` String value for parameter.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.5.5 acu_tcap_ssap_get_locaddr/remaddr

```
acu_sccp_addr_t *acu_tcap_ssap_get_locaddr(acu_tcap_ssap_t *ssap);
acu_sccp_addr_t *acu_tcap_ssap_get_remaddr(acu_tcap_ssap_t *ssap);
```

Purpose

These functions return a pointer to the local/remote SCCP address information for this ssap. The application can change the structure through the returned pointer. The values can also be set from the configuration file and by the configuration functions.

The local SSN and POINTCODE values are used when connecting to SCCP.

Parameters

`ssap` The address of the `acu_tcap_ssap_t` structure.

Return value

The address of the `acu_sccp_addr_t` structure within the ssap data area, or `NULL` if the ssap pointer is invalid.

See section B.2 for details of the `acu_sccp_addr_t` structure.

2.1.6 Transaction functions

2.1.6.1 acu_tcap_transaction_create

```
acu_tcap_trans_t *acu_tcap_transaction_create(acu_tcap_ssap_t ssap);
```

Purpose

This function creates a new TCAP transaction on the specified ssap.

The local transaction id allocated is guaranteed not to be reused for at least 65536 allocate/free pairs. Under normal conditions it will take much longer for the id to get reused.

Parameters

`ssap`	The ssap on which to create a transaction.

Return value

The address of an initialised `acu_tcap_trans_t` structure, or `NULL` if the ssap isn't connected to SCCP or if `malloc()` fails.

Note **TCAP transactions cannot be created until the connection to SCCP has been established.**

2.1.6.2 acu_tcap_transaction_restore

```
acu_tcap_trans_t *acu_tcap_transaction_restore(acu_tcap_ssap_t ssap,
     unsigned int loc_id, unsigned int rem_id, unsigned int rem_id_len);
```

Purpose

This function creates a TCAP transaction on the specified ssap with the specified local and remote transaction ids and in any state.

This allows an application that has saved the transaction ids (etc) in filestore to continue after being killed and restarted.

The remote (and maybe local) SCCP addresses will need to be set (eg by writing through the pointer returned by `acu_tcap_trans_get_remaddr()`) and any operation timers for pending invokes restarted by calling `acu_tcap_operation_restore()`.

If the original BEGIN (sent or received) contained an application context, call `acu_tcap_transaction_restore_app_ctx()` to enable application contexts in later messages.

While it can be used to allocate transactions in the idle state this is not recommended as requesting some pairs of values may require the library allocate 16MB of memory for the transaction id lookup table.

Parameters

`ssap`	The ssap on which to create a transaction.
`loc_id`	The local transaction id required. The high 12 bits must be correct for the ssap. The required range can be set by configuring `TRANID_RANGE`. If zero a local transaction id will be allocated.
`rem_id`	The remote transaction id required.
`rem_id_len`	The length (1 to 4 bytes) of the remote transaction id. If zero the transaction will be created in the outgoing (`BEGIN` sent) state, if 5 (or more) in the idle state.

Return value

The address of an initialised `acu_tcap_trans_t` structure, or `NULL` if the ssap isn't connected to SCCP, `malloc()` fails or if the local transaction is already allocated or isn't in the correct range for the ssap.

Note **To avoid potential duplicate transactions ids restore all old transactions before creating any new ones.**

Note **Transactions can also be automatically restored on reception of unexpected CONTINUE or END messages by configuring `TRAN_RESTORE`.**

2.1.6.3 acu_tcap_transaction_restore_app_ctx

```
int acu_tcap_transaction_restore_app_ctx(acu_tcap_trans_t *tran,
     const void *app_ctx, int app_ctx_len);
```

Purpose

This function sets the flag (usually set if the received/transmitted BEGIN contains an application context) that allows applications contexts in the later messages – especially the first backwards message.

If an application context is specified it is assumed that only a BEGIN has been received and the application context will be sent in the next (assumed first) backwards message. Normally the library saves this from the received BEGIN.

Parameters

`tran`	Restored transaction.
`app_ctx`	The application context name, or (ANSI only) `NULL` for an integer application context name.
`app_ctx_len`	Length in bytes of the application context name, or the numeric application context name if `app_ctx` is `NULL`.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.6.4 acu_tcap_transaction_delete

```
void acu_tcap_transaction_delete(acu_tcap_trans_t *tran);
```

Purpose

This function deletes a TCAP transaction data area and all memory associated with it.

This has the effect of a 'pre-arranged' end on any active TCAP transaction.

Every transaction structure (including those created when a BEGIN message is received) must be explicitly deleted.

Parameters

`tran`	The address of the `acu_tcap_trans_t` structure to delete.

Return value

None.

Note **The transaction data isn't actually deleted until the last message that references the transaction is freed.**

2.1.6.5 acu_tcap_ssap_get_uni_transaction

```
acu_tcap_trans_t *acu_tcap_ssap_get_uni_transaction(acu_tcap_ssap_t ssap);
```

Purpose

This function returns the address of the transaction on which received unidirectional messages are queued.

The transaction is created either by this call, or when the first unidirectional message is received. If the transaction is deleted it will be re-created when needed.

Note **An application will only be given unidirectional messages if '`uni_server = y`' is set in the ssap's configuration.**

Unidirectional messages can be sent from this transaction, or from another transaction created by `acu_tcap_transaction_create()`.

Parameters

`ssap`	The ssap whose unidirectional transaction is required.

Return value

The address of an `acu_tcap_trans_t` structure, or `NULL` if `malloc` fails.

2.1.6.6 acu_tcap_trans_set_userptr

```
void acu_tcap_trans_set_userptr(acu_tcap_trans_t *tran, void *userptr);
```

Purpose

This function saves the pointer to an application data area for this transaction.

Parameters

`tran`	The address of the `acu_tcap_trans_t` structure to modify.
`userptr`	The pointer to save.

2.1.6.7 acu_tcap_trans_get_userptr

```
void *acu_tcap_trans_get_userptr(acu_tcap_trans_t *tran);
```

Purpose

This function retrieves the pointer saved by `acu_tcap_set_userptr()`.

Parameters

`tran`	The address of the `acu_tcap_trans_t` structure.

Return value

The pointer saved previously.

2.1.6.8 acu_tcap_trans_get_ids

```
int acu_tcap_trans_get_ids(acu_tcap_trans_t *tran, unsigned int *loc_id,
    unsigned int *rem_id, unsigned int *rem_id_len);
```

Purpose

This function gets the local and remote transaction identifiers.

Parameters

`tran`	The address of the `acu_tcap_trans_t` structure to modify.
`loc_id`	Address of location to write the local transaction id.
`rem_id`	Address of location to write the remote transaction id.
`rem_id_len`	Address of location to write the length of the remote transaction id.

Any of `loc_id`, `rem_id` and `rem_id_len` may be `NULL` in which case nothing is returned. `*rem_id_len` will be set to zero if the remote transaction identifier is unknown.

ITU TCAP allows transaction identifiers to be between 1 and 4 bytes. ANSI TCAP always uses 4 byte transaction identifiers.

All transaction identifiers created by this product are 4 bytes. The upper 12 bits are the same for all the transactions allocated on a specific ssap, the lower 20 bits are allocated to allow fast lookup while still guaranteeing that, even in the worst case, a transaction ID won't be reallocated for over 98000 allocate/free (and usually much, much, less often).

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.6.9 acu_tcap_trans_set_cfg_int/str

```
int acu_tcap_trans_set_cfg_int(acu_tcap_trans_t *tran,
     acu_tcap_cfg_param_t param, unsigned int i_val);
int acu_tcap_trans_set_cfg_str(acu_tcap_trans_t *tran,
     acu_tcap_cfg_param_t param, const char *s_val);
```

Purpose

These functions set a configurable value of the transaction data area. The default values for these are inherited from the ssap when a transaction is created.
`acu_tcap_trans_set_cfg_str()` can be used to set an integer parameter from a character string value.

Refer to section 2.1.3 for a list of the configurable parameters.

Parameters

`tran`	The address of the `acu_tcap_trans_t` structure to modify.
`param`	Configuration parameter to modify.
`i_val`	Integer value for parameter.
`s_val`	String value for parameter.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.6.10 acu_tcap_trans_get_locaddr/remaddr

```
acu_sccp_addr_t *acu_tcap_trans_get_locaddr(acu_tcap_trans_t *tran);
acu_sccp_addr_t *acu_tcap_trans_get_remaddr(acu_tcap_trans_t *tran);
```

Purpose

These functions return a pointer to the local/remote SCCP address information for this transaction. The application can change the structure through the returned pointer.

The default values for these are inherited from the ssap when a transaction is created.

The remote address will be set from information in the first message received for each transaction.

To respond from the destination address in a received Begin or Unidirectional message (rather than from the configured address) either configure **`respond_rx_loc_gt=y`** (not Unidirectional) or set the transactions local address with:

```
*acu_tcap_trans_get_locaddr(transaction) = *msg->tm_local_addr;
```

when processing the received message.

Parameters

`tran`	Transaction.

Return value

The address of the structure or `NULL` if the `tran` pointer is invalid.

See section B.2 for details of the `acu_sccp_addr_t` structure.

2.1.7 General message functions

The TCAP library uses a single structure to describe both transmit and receive messages. For transmit messages the actual data is allocated using `malloc()`, receive messages usually contain pointers into a large buffer used to receive the data from the driver. Failure to free receive messages leads to communication problems with the driver.

The user-visible part of the message structure contains some fields that are written when messages are decoded. These fields are not used for transmit messages.

2.1.7.1 acu_tcap_msg_alloc

```
acu_tcap_msg_t *acu_tcap_msg_alloc(acu_tcap_trans_t *tran);
```

Purpose

This function allocates a TCAP message for the specified transaction.

The message must be freed later by calling `acu_tcap_msg_free()`.

Parameters

`tran` Transaction this message is for.

Return value

The address of a msg structure if successful, `NULL` on failure.

2.1.7.2 acu_tcap_msg_free

```
void acu_tcap_msg_free(acu_tcap_msg_t *msg);
```

Purpose

This function releases all resources associated with the specified `msg`.

Parameters

`msg` Address of message to free.

Note **Every message must be explicitly freed using this function.**

2.1.7.3 acu_tcap_msg_copy_rx_buffer

```
int acu_tcap_msg_copy_rx_buffer(acu_sccp_msg_t *msg);
```

Purpose

This function copies any data that `msg` references that is in the TCP/IP receive buffer area to a malloced memory area and updates all of the pointers within the message structure to reference the correct locations in the new buffer.

Freeing the space in the receive buffer area is necessary to stop the TCP connection blocking if the message isn't going to be freed quickly (e.g.: when waiting for further responses from a remote system).

Parameters

`msg` Address of message to process.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.7.4 acu_tcap_msg_get_a1b

```
acu_asn1_buf_t *acu_tcap_msg_get_a1b(acu_tcap_msg_t *msg);
```

Purpose

This function returns the address of the asn1 encoder structure used to encode/decode the message. This isn't needed for normal applications, but gives additional flexibility.

Parameters

`msg` Address of message.

2.1.8 Message sending functions

To send a TCAP message, the application must call the following functions in turn:

- `acu_tcap_msg_alloc()` to allocate a message structure.
- `acu_tcap_msg_init()` to add the SCCP address information and the initial part of the TCAP message.
- optionally `acu_tcap_msg_add_dialogue()` to a dialogue portion, usually used to convey an application context.
- optionally `acu_tcap_msg_add_dlg_userinfo()` to add userinfo to the dialogue portion.
- optionally `acu_tcap_msg_add_comp_xxx()` to add each TCAP component.
- `acu_tcap_msg_send()` to finalise the message and send to SCCP and the remote system.
- `acu_tcap_msg_free()` to free the message structure (or reuse it to send another message for the same transaction).

`acu_tcap_msg_add_comp_xxx()` can be called multiple times in order to add more than one component to a message.

Component parameters must be a single piece of BER encoded ASN.1.

ANSI component parameters must be coded as national/private, constructor with code 18, or as universal, constructor with code 16 (i.e.: the first byte is either `0xf2` or `0x30`). If the supplied parameter is invalid it is enclosed in a PRV(18) constructed item. The constructed item is always left unterminated when a component is added (allowing additional parameters to be added).

It is possible to omit the component parameter when adding an invoke, result or error component and then to build the component directly to the buffer using the ASN.1 encoder functions defined in sections 2.2.3 and 2.3.2.1.

Note **ITU specifies that the valid range for an invoke_id is -128 to 127. The TCAP library treats values 128 to 255 as equivalent to -128 to -1 to avoid problems with sign extension.**

Note **Do not directly modify any of the members of the `acu_tcap_msg_t` structure.**

2.1.8.1 acu_tcap_msg_init

```
int acu_tcap_msg_init(acu_tcap_msg_t *msg, acu_tcap_msg_type_t type);
```

Purpose

This function starts building a TCAP message for the specified transaction. The SCCP address information and transaction identifiers are written to the start of `msg`.

The QOS settings are taken from the transaction data area, if necessary they can be changed by calling `acu_tcap_trans_set_cfg_int()` prior to initialising the message.

For a 'pre-arranged' end, just call `acu_tcap_transaction_delete()` to delete the transaction data areas.

Requesting `ACU_TCAP_MSG_DATA` allows the application to send an entire TCAP message (e.g. one extracted from a received message). The data can be added using the ASN.1 encoder functions e.g. `acu_asn1_put_raw_octets()` having called `acu_tcap_msg_get_a1b()` to obtain the encoder's data area. This is useful if the application is acting as an SCCP STP and forwarding TCAP begin messages to a different global title or pointcode. This can also be achieved through the SCCP API.

Parameters

`msg`	Message being built	
`type`	Type of message to build, one of:	
	`ACU_TCAP_MSG_ITU_UNI`	Unidirectional.
	`ACU_TCAP_MSG_ITU_BEGIN`	Begin.
	`ACU_TCAP_MSG_ITU_END`	End.
	`ACU_TCAP_MSG_ITU_CONTINUE`	Continue.
	`ACU_TCAP_MSG_ITU_ABORT`	Abort.
	`ACU_TCAP_MSG_ANSI_UNI`	Unidirectional.
	`ACU_TCAP_MSG_ANSI_QUERY`	Query with permission (to release).
	`ACU_TCAP_MSG_ANSI_QUERY_WO`	Query without permission (to release).
	`ACU_TCAP_MSG_ANSI_RESPONSE`	Response.
	`ACU_TCAP_MSG_ANSI_CONV`	Conversation with permission (to release).
	`ACU_TCAP_MSG_ANSI_CONV_WO`	Conversation without permission (to release).
	`ACU_TCAP_MSG_ANSI_ABORT`	Abort.
	`ACU_TCAP_MSG_DATA`	TCAP message header not added.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.8.2 acu_tcap_msg_add_dialogue

```
int acu_tcap_msg_add_dialogue(acu_tcap_msg_t *mag, unsigned int errval,
     const void *app_ctx, int app_ctx_len);
```

Purpose

This function adds a dialogue portion containing the requested application context name to msg (any dialogue userinfo can be added later).

Dialogue portions are valid in Begin (ANSI Conversation), Unidirectional, Abort, and the first backward message for a transaction. If necessary a dialogue portion will automatically be added to the first backward message if one isn't explicitly requested.

The type of the dialogue PDU is determined by the type of the TCAP message being built.

The `app_ctx` parameter is ignored when building an ITU Abort message unless it is the first backwards message and the `errval` parameter is either `ACU_TCAP_AARE_REJECT_USER_APPLICATION_CONTEXT_NOT_SUPPORTED` or `ACU_TCAP_AARE_REJECT_PROVIDER_NO_COMMON_DIALOGUE_PORTION` in which case an AARE pdu is generated (i.e.: that used in a Continue/End message) instead of the ABRT pdu that an Abort message would normally contain.

A dialogue portion with a user-specified abstract syntax can be added to a ITU Abort message by calling `acu_tcap_msg_add_dialogue_userinfo()` without calling this routine.

Parameters

`msg`	Message being built.

`errval`	Error numbers for dialogue PDUs:
	ITU Begin/Unidirectional: ignored.
	ITU Continue/End/Abort (AARE) typically one of:
	`ACU_TCAP_AARE_ACCEPTED_USER`
	`ACU_TCAP_AARE_REJECT_USER_NULL`
	`ACU_TCAP_AARE_REJECT_USER_NO_REASON`
	`ACU_TCAP_AARE_REJECT_USER_APPLICATION_CONTEXT_NOT_SUPPORTED`
	ITU Abort (ABRT): abort source: 0 => user, 1 => provider.
	ANSI Abort: P-Abort-Cause, see 2.1.9.4.
	ANSI other messages: ignored.

`app_ctx`	The application context name, or (ANSI only) `NULL` for an integer application context name.
`app_ctx_len`	Length in bytes of the application context name, or the numeric application context name if `app_ctx` is `NULL`.

ITU 'application context names' are encoded as ASN.1 'object identifiers'. The `app_ctx` pointer should reference the start of the object identifier data, (not the `0x06` byte at the start of a BER encoded object identifier).

The application context will be omitted from an ANSI Unidirectional or Query message if `app_ctx` is not `NULL` and `app_ctx_len` is zero.

An object identifier can be encoded using the `acu_asn1_encode_object_id_str/int()` routines.

E.g.:

```
acu_tcap_msg_add_dialogue(msg, 0, app_ctx_buf,
            acu_asn1_encode_object_id_str(app_ctx_buf, sizeof app_ctx_buf,
                                          "0.0.17.773.1.1.1"))
```

Although the above object identifier is of TCAP itself, and would never actually appear as an application context.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.8.3 acu_tcap_msg_add_dlg_userinfo

```
int acu_tcap_msg_add_dlg_userinfo(acu_tcap_msg_t *msg, const void *uinfo,
     unsigned int len);
```

Purpose

This function adds user information to the dialogue. It can be called multiple times for a single message.

The user information should be encoded as an ASN.1 EXTERNAL item or as a sequence of ASN.1 EXTERNAL data items (with either a sequence tag (`0x30`), or the implicit constructed tag (`0xbe` or `0xfd`) that appears in the generated message).

It can also be used to add a dialogue portion with user-defined syntax to the later messages of an ITU dialogue. In this case only a single piece of userinfo is allowed.

Parameters

`msg`	Message being built.
`uinfo`	Address of ASN.1 encoded user information.
`len`	Number of bytes of user information.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.8.4 acu_tcap_msg_add_dlg_security_context

```
int acu_tcap_msg_add_dlg_security_context(acu_tcap_msg_t *msg,
     const void *sec_ctx, int sec_ctx_len);
```

Purpose

This function adds a security context to an ANSI dialogue.

Parameters

`msg`	Message being built.
`sec_ctx`	The security context identifier, or `NULL` for an integer security context identifier.
`sec_ctx_len`	Length in bytes of the security context identifier, or the numeric security context identifier.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.8.5 acu_tcap_msg_add_dlg_confidentiality

```
int acu_tcap_msg_add_dlg_confidentiality(acu_tcap_msg_t *msg,
     const void *cfd_alg, int cfd_alg_len, const void *cfd_val,
     unsigned int cfd_val_len);
```

Purpose

This function adds confidentiality information to an ANSI dialogue.

Parameters

`msg`	Message being built.
`cfd_alg`	The confidentiality algorithm identifier, or `NULL` for an integer identifier.
`cfd_alg_len`	Length in bytes of the confidentiality algorithm identifier, or the numeric confidentiality algorithm identifier.
`cfd_val, cfd_val_len`	Pointer to, and length of, the ASN.1 encoded confidentiality value.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.8.6 acu_tcap_msg_add_comp_invoke

```
int acu_tcap_msg_add_comp_invoke(acu_tcap_msg_t *msg, int invoke_id,
     int linked_id, int last_class, int tmo_secs, const void *op_code,
     int op_code_len, const void *param, int param_len);
```

Purpose

This function adds a TCAP invoke component to the message being built.

Parameters

msg	Message being built.
invoke_id	Invoke identifier, In ANSI it may be ACU_TCAP_NO_INVOKE_ID to suppress the inclusion of an invoke id.
linked_id	Linked id, or ACU_TCAP_NO_INVOKE_ID if no linked-id is required.
last_class	The TCAP operation class 0 or 1 to 4, bitwise 'or' in ACU_TCAP_LAST to generate an ANSI 'INVOKE_LAST' component. The operation classes are: 0 Operation state engine disabled, all message sequences are valid. 1 Report success or failure (all responses valid). 2 Report failure only (return-result not valid). 3 Report success only (return-error not valid). 4 Outcome not reported (neither return-result nor return-error valid).
tmo_secs	Operation timeout, if zero the configured value is used.
op_code	Address of an ITU global operation object identifier, or an ANSI private operation. NULL for an ITU local operation or an ANSI national operation.
op_code_len	Length in bytes of the operation code, or the numeric value if op_code is NULL.
param, param_len	Pointer to, and length of, any invoke parameter. If param is NULL or param_len is 0 no parameter will be added.

Note **The maximum timeout is 9 hours.**

Return value

Zero if successful, ACU_TCAP_ERROR_xxx on failure.

2.1.8.7 acu_tcap_msg_add_comp_result

```
int acu_tcap_msg_add_comp_result(acu_tcap_msg_t *msg, int invoke_id,
     int last, const void *op_code, int op_code_len, const void *param,
     int param_len);
```

Purpose

This function adds a TCAP result component to the message being built.

Parameters

msg	Message being built.
invoke_id	Invoke identifier, In ANSI it may be ACU_TCAP_NO_INVOKE_ID to suppress the inclusion of an invoke id.
last	0 or ACU_TCAP_LAST to generate a 'RESULT_LAST' component.
op_code	Address of a global operation object identifier, or NULL for a local operation (ignored for ANSI).
op_code_len	Length in bytes of the operation code, or the numeric value if op_code is NULL (ignored for ANSI).
param	Pointer to, and length of, any result parameter. If param is NULL or param_len is 0 no parameter will be added.
param_len	

ITU TCAP requires that the operation code and parameter both be present or both be absent. The operation code will be added if any of op_code, op_code_len, param or param_len are not NULL or zero. If the application specifies an operation code but doesn't specify a parameter it must add a parameter directly to the message before sending it.

Return value

Zero if successful, ACU_TCAP_ERROR_xxx on failure.

2.1.8.8 acu_tcap_msg_add_comp_error

```
int acu_tcap_msg_add_comp_error(acu_tcap_msg_t *msg, int invoke_id,
     const void *error_code, int error_code_len, const void *param,
     int param_len);
```

Purpose

This function adds a TCAP error component to the message being built.

Parameters

msg	Message being built.
invoke_id	Invoke identifier, In ANSI it may be ACU_TCAP_NO_INVOKE_ID to suppress the inclusion of an invoke id.
error_code	Address of an ITU global error object identifier, or an ANSI private error, NULL for an ITU local error or an ANSI national error.
error_code_len	Length in bytes of the error code, or the numeric value if error_code is NULL.
param	Pointer to, and length of, any error parameter. If param is NULL or
param_len	param_len is 0 no parameter will be added.

Return value

Zero if successful, ACU_TCAP_ERROR_xxx on failure.

2.1.8.9 acu_tcap_msg_add_comp_reject

```
int acu_tcap_msg_add_comp_reject(acu_tcap_msg_t *msg, int invoke_id,
     acu_tcap_reject_problem_t problem, const void *param, int param_len);
```

Purpose

This function adds a TCAP component to the message being built.

Parameters

msg	Message being built.
invoke_id	Invoke identifier, or ACU_TCAP_NO_INVOKE_ID to suppress the invoke-id field
problem	Problem type and code.
param,	ANSI only; pointer to, and length of, any reject parameter. If param is NULL
param_len	or param_len is 0 no parameter will be added.

The problem parameter encodes the problem type and the error value itself as a single field. Valid values are listed in 2.1.9.6

Return value

Zero if successful, ACU_TCAP_ERROR_xxx on failure.

2.1.8.10 acu_tcap_msg_add_ansi_abort_userinfo

```
int acu_tcap_msg_add_ans_abort_userinfo(acu_tcap_msg_t *msg,
     const void *uinfo, unsigned int len);
```

Purpose

This function adds user information to an ANSI Abort message.

The userinfo is an arbitrary sequence of bytes.

Parameters

msg	Message being built.
uinfo	Address of the user information.
len	Number of bytes of user information.

Return value

Zero if successful, ACU_TCAP_ERROR_xxx on failure.

2.1.8.11 acu_tcap_msg_send

```
int acu_tcap_msg_send(acu_tcap_msg_t *msg);
```

Purpose

This function adds any outstanding parameter terminators to the message being built, sets the overall length fields, and sends the built message to SCCP over the TCP/IP connection.

When a TCAP message is built a single byte is allocated for the length field of all constructed items. For lengths less than 128 this is later written with the actual length, for longer items a two byte indefinite length terminator is normally appended. Two configuration parameters affect the behaviour. Setting `ENC_DEF_LEN=y` (before calling `acu_tcap_msg_init()`) causes the definite length encoding to be used for all constructed items. Setting `PREFERRED_MAXLEN=nnn` causes this function to re-encode using the definite length encoding if doing so would reduce the length of the tcap data below the specified size.

Note **The definite length encoding for lengths above 127 is not normally done because it requires an overlapping `memmove()` to make space for the additional byte.**

This function does not free the message buffer. The application may use the buffer to build and send another TCAP message, or call `acu_tcap_msg_free`() to free the message.

Parameters

`msg` Message to send.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.8.12 acu_tcap_msg_reply_reject

```
int acu_tcap_msg_reply_reject(acu_tcap_msg_t *msg);
```

Purpose

This function generates and sends a continue message containing a single reject component that is the correct response to an `ACU_TCAP_COMP_LOCAL_REJECT` component returned by `acu_tcap_msg_get_component()`.

The application can also use the information from the local reject to add the reject component to a different message.

Parameters

`msg` Received message containing the local reject component.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.9 Message receiving functions

TCAP messages received from SCCP (via TCP/IP) are queued on, and can be retrieved from queues on both the ssap and transaction data areas.

Every message must be freed at some point by calling `acu_tcap_msg_free()`.

The application must normally call `acu_tcap_trans_unblock()` after processing messages that refer to a transaction in order to make any further messages for that transaction available from the ssap queue. The block is applied in order to stop an application having more than one thread processing messages for a single transaction.

If the application only ever uses a single thread to access TCAP then `SINGLE_THREADED=y` can be configured and the block will not be applied.

The data bytes of the message itself are within a circular buffer used to receive data from the TCP/IP connection. The application must call `acu_tcap_msg_free()` or `acu_tcap_msg_copy_rx_buffer()` in a timely manner to avoid blocking messages for other transactions.

The initial elements of `acu_tcap_msg_t` are exposed in the header file and can be read by the application.

As well as received TCAP messages, other indications from the library to the application are passed through this interface. These additional messages are only added to the ssap queue.

2.1.9.1 acu_tcap_ssap_msg_get

```
int acu_tcap_ssap_msg_get(acu_tcap_ssap_t *ssap, int tmo_ms,
     acu_tcap_msg_t **msgp);
```

Purpose

This function retrieves the next inbound tcap message from the queue associated with the specified ssap.

If the received message refers to an existing transaction then the `tm_trans` field will be set.

Note **An application will only be given Begin/Query messages if '`server = y`' is set in the ssap's configuration.**

Parameters

`ssap`	Address of ssap data area.
`tmo_ms`	Time in milliseconds to wait for a message, 0 => don't wait, -1 => wait forever.
`msgp`	Address of parameter where the message structure address will be written.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.
`*msgp` will be set to `NULL` if the function fails.

Note **There are some circumstances where `ACU_TCAP_ERROR_NO_MESSAGE` will be returned even when asked to wait indefinitely.**

Note **An indefinite wait will be interrupted if the application calls `acu_tcap_ssap_wakeup_msg_get()` from a different thread.**

The following fields of the message are set:

`tm_msg_type` Type of message/indication, one of:

`ACU_TCAP_MSG_DATA`	Data from remote TCAP ({L|X}UDT messages).
`ACU_TCAP_MSG_NOTICE`	Error report from SCCP ({L|X}UDTS message).
`ACU_TCAP_MSG_TIMEOUT`	Operation timeout.
`ACU_TCAP_MSG_CON_STATE`	Change in state of TCP/IP connections to SCCP.
`ACU_TCAP_MSG_USER_STATUS`	Change in status of remote user (from SCCP).
`ACU_TCAP_MSG_SP_STATUS`	Change in status of remote signalling point from SCCP.

`tm_ssap` Address of associated ssap.

tm_trans	Address of associated existing transaction (may be NULL).

For DATA and NOTICE message the following are also set:

tm_local_addr	Destination (ie our) address from SCCP message.
tm_remote_addr	Source (ie remote) address from SCCP message.
tm_ret_opt	Received SCCP 'return on error' option.
tm_seq_ctrl	Received SCCP 'sequential delivery' option.
tm_ret_cause	'Return cause' from received {L|X}UDTS message.
tm_priority	ANSI message priority, ITU importance (0xff if ITU option not present).
tm_p_abort_cause	Not set until acu_tcap_msg_decode is called.

The dialogue portion and components of DATA and NOTICE messages can be decoded by calling acu_tcap_msg_decode() and then acu_tcap_msg_get_component().

For TIMEOUT messages call acu_tcap_msg_get_component() to find the timed out operation.

For CON_STATUS call acu_tcap_msg_get_con_state() to find the connection states at the time the message was generated, or acu_tcap_get_con_state() to find the current state.

For USER_STATUS and SP_STATUS call acu_tcap_msg_get_sccp_status() to determine the concerned pointcode and SSN.

Note **Remember to call acu_tcap_trans_unblock() when processing is finished, otherwise further messages for the transaction cannot be retrieved from the ssap queue.**

2.1.9.2 acu_tcap_trans_msg_get

```
int acu_tcap_trans_msg_get(acu_tcap_trans_t *trans, int tmo_ms,
     acu_tcap_msg_t **msgp);
```

Purpose

This function retrieves the next inbound tcap message from the queue associated with the specified transaction.

Refer to acu_tcap_ssap_msg_get() for information on the possible message types.

Parameters

trans	Address of transaction data area.
tmo_ms	Time in milliseconds to wait for a message, 0 => don't wait, -1 => wait forever.
msgp	Address of parameter where the message structure address will be written.

Return value

Zero if successful, ACU_TCAP_ERROR_xxx on failure.
*msgp will be set to NULL if the function fails.

2.1.9.3 acu_tcap_event_msg_get

```
int acu_tcap_event_msg_get(acu_tcap_event_t *event, acu_tcap_msg_t **msgp);
```

Purpose

This function retrieves the next inbound tcap message from one of the queues associated with event. Refer to section 2.1.13 for more information on the event mechanism.

Refer to acu_tcap_ssap_msg_get() for information on the possible message types.

Parameters

event	Address of an event data area.
msgp	Address of parameter where the message structure address will be written.

Return value

Zero if successful, ACU_TCAP_ERROR_xxx on failure.
*msgp will be set to NULL if the function fails.

2.1.9.4 acu_tcap_msg_decode

```
int acu_tcap_msg_decode(acu_tcap_msg_t *msg,
     const acu_tcap_dialogue_t **dlgp);
```

Purpose

This function performs the initial decode of an inbound tcap data or notice message

If a data message is an ITU BEGIN or an ANSI QUERY then a new transaction is created by the library. The application is responsible for freeing these transaction structures.

Parameters

`msgp` Message structure address (from one of the msg_get() functions).
`dlgp` Address of parameter where the dialogue structure address will be written.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

If the message doesn't refer to a valid transaction then `msg->tm_trans` will be set to `NULL` (even if it was not `NULL` before the call).

This function may return an error code due to an invalid TCAP message; in this case `tm_msg_type` will have been changed (typically to `ACU_TCAP_MSG_LOCAL_ABORT`).

The `tm_msg_type` field of `msg` is changed to indicate the type of the TCAP message:

`ACU_TCAP_MSG_LOCAL_ABORT`	The received message was a protocol error, a P-ABORT message has been sent. `msg->tm_p_abort_cause` contains the sent cause.
`ACU_TCAP_MSG_P_ABORT`	A P-ABORT was received. `msg->tm_p_abort_cause` contains the cause, one of:

```
ACU_TCAP_P_ABORT_ITU_UNRECOGNIZED_MESSAGE_TYPE
ACU_TCAP_P_ABORT_ITU_UNRECOGNIZED_TRANSACTION_ID
ACU_TCAP_P_ABORT_ITU_BADLY_FORMATTED_TRANSACTION_PORTION
ACU_TCAP_P_ABORT_ITU_INCORRECT_TRANSACTION_PORTION
ACU_TCAP_P_ABORT_ITU_RESOURCE_LIMITATION
ACU_TCAP_P_ABORT_ITU_ABNORMAL_DIALOGUE
ACU_TCAP_P_ABORT_ANSI_UNRECOGNIZED_PACKAGE_TYPE
ACU_TCAP_P_ABORT_ANSI_INCORRECT_TRANSACTION_PORTION
ACU_TCAP_P_ABORT_ANSI_BADLY_STRUCTURED_TRANSACTION_PORTION
ACU_TCAP_P_ABORT_ANSI_UNASSIGNED_RESPONDING_TRANSACTION_ID
ACU_TCAP_P_ABORT_ANSI_PERMISSION_TO_RELEASE_PROBLEM
ACU_TCAP_P_ABORT_ANSI_RESOURCE_UNAVAILABLE
ACU_TCAP_P_ABORT_ANSI_UNRECOGNIZED_DIALOGUE_PORTION_ID
ACU_TCAP_P_ABORT_ANSI_BADLY_STRUCTURED_DIALOGUE_PORTION
ACU_TCAP_P_ABORT_ANSI_MISSING_DIALOGUE_PORTION
ACU_TCAP_P_ABORT_ANSI_INCONSISTENT_DIALOGUE_PORTION
```

`ACU_TCAP_MSG_ITU_UNI`
`ACU_TCAP_MSG_ITU_BEGIN`
`ACU_TCAP_MSG_ITU_END`
`ACU_TCAP_MSG_ITU_CONTINUE`
`ACU_TCAP_MSG_ITU_ABORT`
`ACU_TCAP_MSG_ANSI_UNI`
`ACU_TCAP_MSG_ANSI_QUERY`
`ACU_TCAP_MSG_ANSI_QUERY_WO`
`ACU_TCAP_MSG_ANSI_RESPONSE`
`ACU_TCAP_MSG_ANSI_CONV`
`ACU_TCAP_MSG_ANSI_CONV_WO`
`ACU_TCAP_MSG_ANSI_ABORT`
	A valid TCAP message of the specified type was decoded.

Note **The ANSI user abort information is passed to the application as if it were a component.**

If the message has a dialogue portion then `dlgp` will point to an `acu_tcap_dialogue_t` structure (embedded in `msg`) that contains the following fields:

`td_type` — The type of the received dialogue, one of:

- `ACU_TCAP_DLG_ITU_USER_SYNTAX` — ITU dialogue with user defined syntax.
- `ACU_TCAP_DLG_ITU_AARQ_AUDT` — ITU dialogue request or unitdata pdu.
- `ACU_TCAP_DLG_ITU_AARE` — ITU dialogue response pdu.
- `ACU_TCAP_DLG_ITU_ABRT` — ITU dialogue abort pdu.
- `ACU_TCAP_DLG_ANSI` — ANSI dialogue.

`td_flags` — Bit pattern indicating which of the fields below are valid:

- `ACU_TCAP_DF_HAS_UI` — `td_ui` and `td_ui_len`.
- `ACU_TCAP_DF_HAS_HEX_APP_CTX` — `td_app_ctx` and `td_app_ctx_len`.
- `ACU_TCAP_DF_HAS_INT_APP_CTX` — in `td_app_ctx_len` (ANSI only).
- `ACU_TCAP_DF_HAS_AARE_DIAG` — `td_result` is from an ITU `AARE` pdu, one of:
  - `ACU_TCAP_AARE_ACCEPTED_USER`
  - `ACU_TCAP_AARE_ACCEPTED_PROVIDER`
  - `ACU_TCAP_AARE_REJECT_USER_NULL`
  - `ACU_TCAP_AARE_REJECT_USER_NO_REASON`
  - `ACU_TCAP_AARE_REJECT_USER_APPLICATION_CONTEXT_NOT_SUPPORTED`
  - `ACU_TCAP_AARE_REJECT_PROVIDER_NULL`
  - `ACU_TCAP_AARE_REJECT_PROVIDER_NO_REASON`
  - `ACU_TCAP_AARE_REJECT_PROVIDER_NO_COMMON_DIALOGUE_PORTION`
- `ACU_TCAP_DF_HAS_ABRT_SOURCE` — `td_result` is user/provider field from an ITU `ABRT` pdu.
- `ACU_TCAP_DF_HAS_USER_SYNTAX` — User syntax data in `td_ui` and `td_ui_len`.
- `ACU_TCAP_DF_HAS_INT_SEC_CTX` — Integer security context in `td_sec_ctx_len`.
- `ACU_TCAP_DF_HAS_OBJ_SEC_CTX` — `td_sec_ctx` and `td_sec_ctx_len`.
- `ACU_TCAP_DF_HAS_INT_CFD_ALG` — Integer confidentiality algorithm ID in `td_cfg_alg_len`.
- `ACU_TCAP_DF_HAS_OBJ_CFD_ALG` — `td_cfd_alg` and `td_cfg_alg_len`.
- `ACU_TCAP_DF_HAS_CFD_VAL` — `td_cfd_val` and `td_cfg_val_len`.

`td_app_ctx`, `td_app_ctx_len` — Pointer to, and length of the application context name (object identifier).

`td_result` — Error code from ITU AARE or ABRT.

`td_sec_ctx`, `td_sec_ctx_len` — Pointer to, and length of, security context object identifier (ANSI only).

`td_ui`, `td_ui_len` — Pointer to, and length of, dialogue user information.

`td_cfd_alg`, `td_cfg_alg_len` — Pointer to, and length of, confidentiality algorithm ID object identifier (ANSI only).

`td_cfd_val`, `td_cfg_val_len` — Pointer to, and length of, ASN.1 encoded confidentiality value (ANSI only).

2.1.9.5 acu_tcap_msg_has_components

```
int acu_tcap_msg_has_components(acu_tcap_msg_t *msg);
```

Purpose

This function is a predicate for determining whether an inbound message has components.

Note **There is no need to call this before calling `acu_tcap_msg_get_component()`.**

Parameters

`msg` Address of message to check.

Return value

Non-zero if the message has components, zero otherwise.

2.1.9.6 acu_tcap_msg_get_component

```
int acu_tcap_msg_get_component(acu_tcap_msg_t *msg,
     const acu_tcap_component_t **component);
```

Purpose

This function decodes the next component from the given message. It should be called in a loop until it returns an error.

Some information which isn't strictly part of a TCAP component is also made available through this interface.

Note **acu_tcap_msg_decode() must be called on a received message before this function.**

The component information is overwritten when `acu_tcap_msg_get_component` is called again for the same `msg`, and discarded when `msg` itself is freed.

Parameters

`msg`	Address of message to decode.
`component`	Address where a pointer to the component information is written.

Return value

Zero if successful, `ACU_TCAP_ERROR_NO_COMPONENT` if there are no more components in the message, `ACU_TCAP_ERROR_xxx` on failure.

If successful `component` will point to a structure with the following members:

`tc_type`	Type of the received component, one of:	
	`ACU_TCAP_COMP_LOCAL_REJECT`	Malformed component received.
	`ACU_TCAP_COMP_OP_TIMEOUT`	Operation timed out.
	`ACU_TCAP_COMP_ANSI_ABORT`	Userinfo from ANSI abort.
	`ACU_TCAP_COMP_ITU_INVOKE`	
	`ACU_TCAP_COMP_ITU_RESULT_LAST`	
	`ACU_TCAP_COMP_ITU_ERROR`	
	`ACU_TCAP_COMP_ITU_REJECT`	
	`ACU_TCAP_COMP_ITU_RESULT_NOTLAST`	
	`ACU_TCAP_COMP_ANSI_INVOKE_LAST`	
	`ACU_TCAP_COMP_ANSI_RESULT_LAST`	
	`ACU_TCAP_COMP_ANSI_ERROR`	
	`ACU_TCAP_COMP_ANSI_REJECT`	
	`ACU_TCAP_COMP_ANSI_INVOKE_NOTLAST`	
	`ACU_TCAP_COMP_ANSI_RESULT_NOTLAST`	
`tc_flags`	Bit-pattern indicating which of the fields below are valid.	
	`ACU_TCAP_CF_HAS_INVOKE_ID`	`tc_invoke_id`.
	`ACU_TCAP_CF_HAS_LINKED_ID`	`tc_linked_id`.
	`ACU_TCAP_CF_HAS_PARAMETER`	`tc_param` and `tc_param_len`.
	`ACU_TCAP_CF_HAS_HEX_OPCODE`	`tc_op_code` and `tc_op_code_len`.
	`ACU_TCAP_CF_HAS_INT_OPCODE`	`tc_op_code_val`.
	`ACU_TCAP_CF_INVOKE_ID_LOCAL`	`tc_invoke_id` is a local invoke id.
`tc_invoke_id`	Invoke-id identifying the operation.	
`tc_linked_id`	Linked invoke-id from invoke message.	
`tc_op_code`, `tc_op_code_len`	Pointer to, and length of multi-byte operation/result/error code, ITU global, ANSI private.	
`tc_op_code_val`	Integral operation/result/error code, ITU local, ANSI national.	
`tc_param`, `tc_param_len`	Pointer to, and length of the component parameter.	
`tc_rejected_type`	For `LOCAL_REJECT` the original message type.	
`tc_reject_error`	For `LOCAL_REJECT` the reject cause to send.	

`acu_tcap_msg_reply_reject()` can be used to send out a reject component in a Continue message in response to a `LOCAL_REJECT` indication.

The ‘problem code’ from received reject components is put into the `tc_op_code_val` field. Valid values for ITU TCAP are:

```
ACU_TCAP_REJECT_ITU_GENERAL_UNRECOGNIZED_COMPONENT
ACU_TCAP_REJECT_ITU_GENERAL_MISTYPED_COMPONENT
ACU_TCAP_REJECT_ITU_GENERAL_BADLY_STRUCTURED_COMPONENT
ACU_TCAP_REJECT_ITU_INVOKE_DUPLICATE_INVOKE_ID
ACU_TCAP_REJECT_ITU_INVOKE_UNRECOGNIZED_OPERATION
ACU_TCAP_REJECT_ITU_INVOKE_MISTYPED_PARAMETER
ACU_TCAP_REJECT_ITU_INVOKE_RESOURCE_LIMITATION
ACU_TCAP_REJECT_ITU_INVOKE_INITIATING_RELEASE
ACU_TCAP_REJECT_ITU_INVOKE_UNRECOGNIZED_LINKED_ID
ACU_TCAP_REJECT_ITU_INVOKE_LINKED_RESPONSE_UNEXPECTED
ACU_TCAP_REJECT_ITU_INVOKE_UNEXPECTED_LINKED_OPERATION
ACU_TCAP_REJECT_ITU_RESULT_UNRECOGNIZED_INVOKE_ID
ACU_TCAP_REJECT_ITU_RESULT_RETURN_RESULT_UNEXPECTED
ACU_TCAP_REJECT_ITU_RESULT_MISTYPED_PARAMETER
ACU_TCAP_REJECT_ITU_ERROR_UNRECOGNIZED_INVOKE_ID
ACU_TCAP_REJECT_ITU_ERROR_RETURN_ERROR_UNEXPECTED
ACU_TCAP_REJECT_ITU_ERROR_UNRECOGNIZED_ERROR
ACU_TCAP_REJECT_ITU_ERROR_UNEXPECTED_ERROR
ACU_TCAP_REJECT_ITU_ERROR_MISTYPED_PARAMETER
```

And for ANSI TCAP are:

```
ACU_TCAP_REJECT_ANSI_GENERAL_UNRECOGNIZED_COMPONENT_TYPE
ACU_TCAP_REJECT_ANSI_GENERAL_INCORRECT_COMPONENT_PORTION
ACU_TCAP_REJECT_ANSI_GENERAL_BADLY_STRUCTURED_COMPONENT_PORTION
ACU_TCAP_REJECT_ANSI_GENERAL_INCORRECT_COMPONENT_CODING
ACU_TCAP_REJECT_ANSI_INVOKE_DUPLICATE_INVOKE_ID
ACU_TCAP_REJECT_ANSI_INVOKE_UNRECOGNIZED_OPERATION_CODE
ACU_TCAP_REJECT_ANSI_INVOKE_INCORRECT_PARAMETER
ACU_TCAP_REJECT_ANSI_INVOKE_UNRECOGNIZED_CORRELATION_ID
ACU_TCAP_REJECT_ANSI_RESULT_UNASSIGNED_CORRELATION_ID
ACU_TCAP_REJECT_ANSI_RESULT_UNEXPECTED_RETURN_RESULT
ACU_TCAP_REJECT_ANSI_RESULT_INCORRECT_PARAMETER
ACU_TCAP_REJECT_ANSI_ERROR_UNASSIGNED_CORRELATION_ID
ACU_TCAP_REJECT_ANSI_ERROR_UNEXPECTED_RETURN_ERROR
ACU_TCAP_REJECT_ANSI_ERROR_UNRECOGNIZED_ERROR
ACU_TCAP_REJECT_ANSI_ERROR_UNEXPECTED_ERROR
ACU_TCAP_REJECT_ANSI_ERROR_INCORRECT_PARAMETER
ACU_TCAP_REJECT_ANSI_TRANS_UNRECOGNIZED_PACKAGE_TYPE
ACU_TCAP_REJECT_ANSI_TRANS_INCORRECT_TRANSACTION_PORTION
ACU_TCAP_REJECT_ANSI_TRANS_BADLY_STRUCTURED_TRANSACTION_PORTION
ACU_TCAP_REJECT_ANSI_TRANS_UNASSIGNED_RESPONDING_TRANSACTION_ID
ACU_TCAP_REJECT_ANSI_TRANS_PERMISSION_TO_RELEASE
ACU_TCAP_REJECT_ANSI_TRANS_RESOURCE_UNAVAILABLE
```

2.1.9.7 acu_tcap_trans_unblock

```
void acu_tcap_trans_unblock(acu_tcap_trans_t *trans);
```

Purpose

This function removes the block that stops inbound messages for the given transaction from being retrieved from the corresponding ssap queue.

The block exists so that a pool of threads can be used to process messages from the ssap queue without having to worry about multiple threads processing messages from the same transaction. It also allows the application to use a separate thread for each transaction, although this is discouraged because of the resource issues with large numbers of threads.

Parameters

`trans` Address of transaction data area.

2.1.9.8 acu_tcap_trans_block

```
int acu_tcap_trans_block(acu_tcap_trans_t *trans);
```

Purpose

This function sets the block that stops inbound messages for the given transaction from being retrieved from the corresponding ssap queue.

The block is automatically set whenever a message is retrieved for a transaction unless `SINGLE_THREADED=y` is configured.

It may be necessary to manually set the block on a newly created transaction.

Parameters

`trans` Address of transaction data area.

Return value

One if the block was already set, zero otherwise.

2.1.9.9 acu_tcap_ssap_wakeup_msg_get

```
void acu_tcap_ssap_wakeup_msg_get(acu_tcap_ssap_t *ssap);
```

Purpose

This function wakes up all threads sleeping in `acu_tcap_ssap_msg_get()` for the specified ssap. This allows an application to shut down tidily.

If no threads are sleeping then the next call to `.acu_tcap_ssap_msg_get()` will not block.

Parameters

`ssap` Address of ssap data area.

Note **If an application has multiple threads reading from the ssap queue then to ensure all are woken they should call acu_tcap_ssap_wakeup_msg_get() after being woken.**

2.1.9.10 acu_tcap_trans_wakeup_msg_get

```
void acu_tcap_trans_wakeup_msg_get(acu_tcap_trans_t *trans);
```

Purpose

This function wakes up all threads sleeping in `acu_tcap_trans_msg_get()` for the specified transaction.

If no threads are sleeping then the next call to `.acu_tcap_trans_msg_get()` will not block.

Parameters

`trans` Address of the transaction data area.

2.1.10 Operation and timer functions

These functions control the operation state machine and timers described in section 3.2.1.1.3 of Q.774. The timer functions can also be used by the application for any other purpose.

The state engine acts differently for each TCAP class (1 to 4). Setting the class to zero disables the state engine and timeouts, all components will be delivered to the application regardless of the sequence in which they arrive.

When a timer expires, a message with `tm_msg_type` set to `ACU_TCAP_MSG_TIMEOUT` will be queued. It will have a single component that identifies the invoke id of the timed-out operation.

Note **The timer resolution is 1 second. A 1 second timer is guaranteed to sleep for at least 1 second, but may sleep for almost 3 seconds.**

Note **The maximum timeout is 9 hours.**

2.1.10.1 acu_tcap_operation_timer_start

```
int acu_tcap_operation_timer_start(acu_tcap_trans_t *trans, int invoke_id,
     unsigned int tmo_secs);
```

Purpose

This function starts the operation timer for the given invoke-id; if the timer is already running it will be restarted with the new interval.

This can be used by an application to run a timer for its own purposes. The `invoke_id` specified must not be used in an invoke message while the timer is running.

Parameters

`trans`	Transaction data area.
`invoke_id`	Invoke id of the operation.
`tmo_secs`	Required timeout in seconds.

Return value

Zero if successful, `ACU_TCAP_ERROR_MALLOC_FAILURE` if the timer table needs extending and `realloc()` fails.

2.1.10.2 acu_tcap_operation_timer_restore

```
int acu_tcap_operation_timer_restore(acu_tcap_trans_t *trans, int invoke_id,
     unsigned int class, unsigned int tmo_secs);
```

Purpose

This function starts the operation timer for the given invoke-id; the operation state is set to that for an outstanding invoke of the specified class.

This function would normally used on transactions created by `acu_tcap_transaction_restore()` so that results from pending invokes are not rejected..

Parameters

`trans`	Transaction data area.
`invoke_id`	Invoke id of the operation.
`class`	Class of the operation.
`tmo_secs`	Required timeout in seconds.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.10.3 acu_tcap_operation_timer_restart

```
int acu_tcap_operation_timer_restart(acu_tcap_trans_t *trans, int invoke_id,
    unsigned int tmo_secs);
```

Purpose

This function restarts the operation timer for the given invoke-id.
An error will be returned it the timer isn't running (e.g.: if it has just expired).

This function can be used to extend the timeout of a TCAP operation.

Parameters

`trans`	Transaction data area.
`invoke_id`	Invoke id of the operation.
`tmo_secs`	Required timeout in seconds.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.10.4 acu_tcap_operation_cancel

```
int acu_tcap_operation_cancel(acu_tcap_trans_t *trans, int invoke_id);
```

Purpose

This function cancels the operation timer for the given invoke-id. The timer is stopped and the state machine set to the idle state.

Cancelling an operation is a local action; the remote system is not informed. The application level protocol should allow for any messages that are not sent or discarded.

Parameters

`trans`	Transaction data area.
`invoke_id`	Invoke id of the operation.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.11 Connection status functions

The TCAP library connects to the SCCP driver using TCP/IP. It connects asynchronously and will automatically attempt to reconnect if the connection fails for any reason.

Changes in the connections' state are reported by queueing an `ACU_TCAP_MSG_CON_STATE` message onto the ssap message queue. The application must wait until the `IN_SERVICE` state is reported before creating any transactions.

Note **The `IDLE` -> `CONNECTING` and `CONNECTING` -> `CONNECTED` transitions are not reported.**

2.1.11.1 acu_tcap_get_con_state

```
int acu_tcap_get_con_state(acu_tcap_ssap_t *ssap, int con_id,
     const acu_tcap_con_state_t **con_state);
```

Purpose

This function returns information about the current state of one of the TCP/IP connections to SCCP.

Parameters

`ssap` Address of ssap data area.
`con_id` 0 for the connection to 'host a', 1 for that to 'host b'.
`con_state` Pointer filled with the address of the connection state structure.

The `acu_tcap_con_state_t` structure contains the following fields:
`cs_ipaddr` IP address of the connected SCCP (host order).
`cs_tcpport` TCP port number of the connected SCCP.
`cs_state` The current state of the connection, one of:

`ACU_TCAP_CON_STATE_IDLE`	Not configured or connect failed.
`ACU_TCAP_CON_STATE_CONNECTING`	TCP connection being made.
`ACU_TCAP_CON_STATE_CONNECTED`	Initial message handshake in progress.
`ACU_TCAP_CON_STATE_IN_SERVICE`	Available for TCAP traffic.
	If `IN_SERVICE` the following bits can also be set:
`ACU_TCAP_CON_STATE_RX_BLOCKED`	No space in receive ring buffer area.
`ACU_TCAP_CON_STATE_RX_FLOW`	Receive flow controlled off.
`ACU_TCAP_CON_STATE_TX_BLOCKED`	No TCP transmit window.
`ACU_TCAP_CON_STATE_TX_FLOW`	Transmit flow controlled off.

`cs_failure` The reason why the last connection (or connect attempt) failed, one of:

`ACU_TCAP_CON_FAIL_SSAP_DELETED`	ssap deleted.
`ACU_TCAP_CON_FAIL_CON_TIMEOUT`	TCP/IP connect timed out.
`ACU_TCAP_CON_FAIL_CON_REJECTED`	TCP/IP connection rejected.
`ACU_TCAP_CON_FAIL_LOGIN_REJECTED`	Login sequence failed.
`ACU_TCAP_CON_FAIL_INWARD`	Disconnected by SCCP.
`ACU_TCAP_CON_FAIL_KEEPALIVE`	No response to keepalive.
`ACU_TCAP_CON_FAIL_BAD_MESSAGE`	Corrupt message received.

`cs_fail_text` Textual description of `cs_failure`, or one of the following texts when the login fails:

`Bad Request`	Major discrepancy between the versions of the TCAP library and the driver.
`Responder has gone`	The driver is no longer waiting for connections on the requested TCP/IP port. Driver is probably shut down.
`Unknown service`	TCAP isn't configured in the ss7 driver configuration.
`Unknown service parameter`	TCAP isn't configured on the requested pointcode.
`Incorrect password`	The passwords in the application and driver configuration files do not match.
`Rejected by server`	Connection rejected by TCAP driver stub.
`Bad hash in response`	Three-way login handshake failed.

`cs_tx_qlen` The number of outbound tcap messages queued within the library.

Application level acknowledgements are used on the TCP connection in order to avoid blocking the TCP connection itself. Thus the `BLOCKED` states should not happen.

Receive flow control is most likely to occur if the application fails to free receive messages – which have pointers directly into the receive ring buffer area.

If transmit flow control is reported the application should take steps to avoid sending further messages. However all messages sent will be queued by the library.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.11.2 acu_tcap_msg_get_con_state

```
int acu_tcap_msg_get_con_state(acu_tcap_msg_t *msg,
     const acu_tcap_con_state_t **cs_a, const acu_tcap_con_state_t **cs_b)
```

Purpose

This function resolves pointers to the connection state field(s) in messages of type `ACU_MSG_TCAP_CON_STATE`.

This information relates to the state of the connections to SCCP at the time the indication was generated.

Refer to `acu_tcap_get_con_state()` for details of the `acu_tcap_con_state_t` structure.

Parameters

`msg`	Message structure address (from one of the msg_get() functions) .
`cs_a`	Address of parameter where the 'host a' connection state structure address will be written.
`cs_b`	Address of parameter where the 'host b' connection state structure address will be written (where SCCP is configured in 'dual' mode).

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

Note **The addresses written to `cs_a` and `cs_b` point into the message itself.**

2.1.12 Remote SP and SSN status functions

The TCAP library receives status indications from SCCP that show the accessibility of remote entities. The information is saved so that the application can synchronously determine the current status.

The application can also ask to be notified when the status of a remote pointcode or ssn changes. Such changes are reported by queueing an `ACU_TCAP_MSG_SCCP_STATUS` message onto the ssap message queue.

Additionally the application can request to be given all of the raw status events from SCCP by setting the `ACU_TCAP_STATUS_IND` flag when the ssap is created.

2.1.12.1 acu_tcap_get_sccp_status

```
int acu_tcap_get_sccp_status(acu_tcap_ssap_t *ssap, unsigned int pointcode,
     unsigned int ssn, const acu_tcap_sccp_status_t **sccp_status);
```

Purpose

This function returns information about the current state of the pointcode and ssn.

Parameters

`ssap`	Address of ssap data area.
`pointcode`	SS7 pointcode of the remote system.
`ssn`	ssn of remote application.
`sccp_status`	Pointer filled with address of the sccp and user state structure.

The `acu_tcap_sccp_status_t` structure contains the following fields:

`tsp_pc`	Remote pointcode.	
`tsp_ssn`	ssn of remote application.	
`tsp_host`	Either ‘a’ or ‘b’ depending of which SCCP host the information came from.	
`tsp_user_status`	Status of the ssn, one of:	
	`ACU_SCCP_UIS`	User In Service.
	`ACU_SCCP_UOS`	User Out of Service.
`tsp_sp_status`	Status of the signalling point (from MTP3), one of:	
	`ACU_SCCP_SP_PROHIBIT`	Prohibited.
	`ACU_SCCP_SP_ACCESS`	Accessible.
`tsp_sccp_status`	Status of the remote SCCP, one of:	
	`ACU_SCCP_REM_SCCP_PROHIBIT`	Prohibited.
	`ACU_SCCP_REM_SCCP_UNAVAIL`	Unavailable, reason unknown.
	`ACU_SCCP_REM_SCCP_UNEQUIP`	Unequipped.
	`ACU_SCCP_REM_SCCP_INACCESS`	Inaccessible.
	`ACU_SCCP_REM_SCCP_CONGEST`	Congested.
	`ACU_SCCP_REM_SCCP_AVAIL`	Available.
`tsp_tx_cong_cost`	A measure of the level of congestion of the remote node.	

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.12.2 acu_tcap_msg_get_sccp_status

```
int acu_tcap_msg_get_sccp_status(acu_tcap_msg_t *msg,
     const acu_tcap_sccp_status_t **sccp_status);
```

Purpose

This function returns the information about the state of a pointcode and ssn from an ACU_TCAP_MSG_USER_STATUS or ACU_TCAP_MSG_SP_STATUS message.

Parameters

msg	Message data area.
sccp_status	Pointer filled with address of the sccp and user state structure (embedded in the msg).

Return value

Zero if successful, ACU_TCAP_ERROR_xxx on failure.

2.1.12.3 acu_tcap_enable_user_status

```
int acu_tcap_enable_user_status(acu_tcap_ssap_t *ssap,
     unsigned int pointcode, unsigned, int ssn);
```

Purpose

This function enables the receipt of ACU_TCAP_MSG_USER_STATUS messages for the specified pointcode and ssn.

Parameters

ssap	ssap data area.
pointcode	Remote SS7 pointcode from which user status indications are required.
ssn	Associated remote SCCP ssn.

The pointcode and/or ssn may be specified as ~0u in which case indications will be given for all pointcodes/ssns.

Note **User status is only reported if the SS7 stack configuration file contains an SCCP [CONCERNED] section for the pointcode and ssn.**

Return value

Zero if successful, ACU_TCAP_ERROR_xxx on failure.

2.1.12.4 acu_tcap_enable_sp_status

```
int acu_tcap_enable_sp_status(acu_tcap_ssap_t *ssap, unsigned int pointcode);
```

Purpose

This function enables the receipt of ACU_TCAP_MSG_SP_STATUS messages for the specified pointcode.

Parameters

ssap	ssap data area.
pointcode	Remote SS7 pointcode from which user status indications are required.

The pointcode may be specified as ~0u in which case indications will be given for all pointcodes.

Note **The 'unavailable', 'unequipped', 'inaccessible' and 'congested' statuses are only reported if the SS7 stack configuration file contains an SCCP [CONCERNED] section for the pointcode.**

Return value

Zero if successful, ACU_TCAP_ERROR_xxx on failure.

2.1.13 TCAP message events

The message receiving functions allow an application to wait for messages on an ssap or a transaction, however there are cases where an application may need to wait for messages on a group of transactions, or wait for messages from TCAP and events from some other part of the system. The event mechanism described here solves both these problems.

On Microsoft Windows events are implemented using manual-reset events, on Linux systems pipes are used. This allows the application to use `WaitForMultipleObjects` or `poll`/`select` to wait for TCAP messages. Due to scalability problems with both of these it is inappropriate to allocate an event for each transaction. The application can create an event that can be signalled by messages being queued at several TCAP transactions, or queued at the ssap itself.

Note **The transactions must all be on the same ssap**

2.1.13.1 acu_tcap_event_create

```
acu_tcap_event_t *acu_tcap_event_create(acu_tcap_ssap_t *ssap);
```

Purpose

This function creates an event structure.

Parameters

`ssap` ssap data area.

Return value

Address of an initialised event structure. `NULL` if one cannot be allocated or the ssap pointer is invalid.

2.1.13.2 acu_tcap_event_delete

```
void acu_tcap_event_delete(acu_tcap_event_t *event);
```

Purpose

This function unlinks the event from any message queues and then deletes the structure itself.

Parameters

`event` Address of event structure.

Return value

None.

2.1.13.3 acu_tcap_event_wait

```
int acu_tcap_event_wait(acu_tcap_event_t *event, int tmo_ms);
```

Purpose

This function waits for the specified event to be signalled.

It is a simple wrapper for `WaitForSingleObject()` or `poll()`.

Parameters

`event` Address of event data area.
`tmo_ms` Time to wait in milliseconds, 0 => don't wait, -1 => wait for ever.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.13.4 acu_tcap_event_get_os_event

```
acu_tcap_os_event_t acu_tcap_event_get_os_event(acu_tcap_event_t *event);
```

Purpose

This function returns the operating system data item underlying the given event.

The return type is actually `HANDLE` for Windows and `int` for Linux systems.

Parameters

`event` Address of event data area.

Return value

For Windows the `HANDLE` of the windows event.
For Linux the file descriptor number of the read side of a pipe.
If the call is invalid 0 is returned; care is taken to ensure the pipe fd number isn't zero, one or two.

2.1.13.5 acu_tcap_event_clear

```
void acu_tcap_event_clear(acu_tcap_event_t *event);
```

Purpose

This function clears (i.e.: returns to the non-signalled state) the operating system item underlying the given event.

The event is automatically cleared if when `acu_tcap_event_msg_get()` returns the last message or fails because no messages are present.

Parameters

`event` Address of event data area.

Return value

None.

2.1.13.6 acu_tcap_event_ssap_attach

```
int acu_tcap_event_ssap_attach(acu_tcap_event_t *event,
    acu_tcap_ssap_t *ssap);
```

Purpose

This function adds the message queue for `ssap` as a source for the `event`.

Note **The ssap specified must be the same one specified when the event was created.**

Parameters

`event` Address of event data area.
`ssap` Address of corresponding ssap data area.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.13.7 acu_tcap_event_ssap_detach

```
int acu_tcap_event_ssap_detach(acu_tcap_event_t *event,
    acu_tcap_ssap_t *ssap);
```

Purpose

This function removes the message queue for the `ssap` from the sources for `event`. It reverses the effect of `acu_tcap_event_ssap_attach()`

Parameters

`event` Address of event structure.
`ssap` Address of ssap data area.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.13.8 acu_tcap_event_ssap_detach_all

```
int acu_tcap_event_ssap_detach_all(acu_tcap_ssap_t *ssap);
```

Purpose

This function detaches the message queue for the ssap from all events. It is implicitly called if the ssap is deleted.

Parameters

`ssap` ssap data area.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.13.9 acu_tcap_event_trans_attach

```
int acu_tcap_event_trans_attach(acu_tcap_event_t *event,
     acu_tcap_trans_t *trans);
```

Purpose

This function adds the message queue for `trans` as a source for the `event`.

Note **The transaction and event must have been created on the same ssap.**

Parameters

`event` Address of event structures.
`trans` Transaction data area.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.13.10 acu_tcap_event_trans_detach

```
int acu_tcap_event_trans_detach(acu_tcap_avent_t *event,
     acu_tcap_trans_t *trans);
```

Purpose

This function removes the message queue for `trans` from the sources for `event`. It reverses the effect of `acu_tcap_event_trans_attach()`

Parameters

`event` Address of event data area.
`trans` Transaction data area.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.1.13.11 acu_tcap_event_trans_detach_all

```
int acu_tcap_event_trans_detach_all(acu_tcap_trans_t *trans);
```

Purpose

This function detaches the message queue for `trans` from all events. It is implicitly called if the transaction is deleted.

Parameters

`trans` Transaction data area.

Return value

Zero if successful, `ACU_TCAP_ERROR_xxx` on failure.

2.2 ASN.1 Encoder/Decoder functions

The TCAP library contains a simple ASN.1 BER encoder/decoder that is used internally and can be used by applications to process TCAP dialogues and components.

This encoder does not parse the ASN.1 descriptions found in standards documents, but does handle the correct encoding of multi-byte tags, multi-byte length fields, integers and both definite and indefinite length constructed items.

A separate C call is used to encode/decode every field allowing the application complete flexibility in the way that the data is processed.

ASN.1 tag values must be encoded by the `ACU_ASN1_MAKE_TAG() #define`.

The message definition based ASN.1 encoder and decoder (see 2.3) is a higher level interface to these functions.

2.2.1 Header file tcap_asn1_codec.h

This header file defines the constants, structures and function prototypes of the encoder/decoder. It is included by the `tcap_api.h` header file.

2.2.1.1 acu_asn1_buf_t structure

The ASN.1 encoder and decoder use the `acu_asn1_buf_t` structure to control the encoding and decoding of ASN.1 data. The application has to access (but must not write to) some of the fields; the offsets of those fields will not change between releases, maintaining binary compatibility for applications. There are other undocumented fields which could change between releases.

The following structure members can be accessed:

`a1b_buf`	The start of ASN.1 buffer area.
`a1b_end`	The end of the buffer area. i.e.: after the last byte to decode, or the limit of the buffer for the encoder.
`a1b_ptr`	The next byte to be decoded, or the location to write the next encoded byte.
`a1b_flags`	Flags, bitwise or of `ACU_A1BF_xxx` values.
`a1b_error`	The error code for the first error detected since the structure was last initialised.
`a1b_error_offset`	The offset in the buffer (i.e.: `a1b->a1b_ptr - a1b->a1b_buf`) when the error was detected.
`a1b_error_format`	A printf format string that further describes the error.
`a1b_error_val_1`	The first integer parameter to the format string.
`a1b_error_val_2`	The second integer parameter to the format string

After using the encoder, the application should use the `a1b_buf` and `a1b_ptr` members to locate the encoded data bytes.

The TCAP library uses the error fields to log any encode or decode errors the to ssap's logfile.

2.2.1.2 Error codes for ASN.1 codec

The ASN.1 encoder and decoder functions return the following error codes, all of which are small negative integers:

`ACU_ASN1_ERROR_BAD_TAG`	Universal TAG must (not) be constructed.
`ACU_ASN1_ERROR_TOOLONG`	Encoded data would be too long.
`ACU_ASN1_ERROR_TOOCONSTRUCTED`	Too many levels of constructed ASN.1.
`ACU_ASN1_ERROR_BAD_DATA`	Data length invalid for specified tag.
`ACU_ASN1_ERROR_BAD_LENGTH`	Data item longer than data in buffer.
`ACU_ASN1_ERROR_BAD_CODE`	Invalid multi-byte tag value.
`ACU_ASN1_ERROR_NOTCONSTRUCTED`	Not inside a constructed item.
`ACU_ASN1_ERROR_NOMEMORY`	`malloc()` failed.
`ACU_ASN1_ERROR_BAD_MAGIC`	Supplied address isn't a `acu_asn1_buf_t` structure.
`ACU_ASN1_ERROR_TRUNCATED`	User supplied buffer too short.

ACU_ASN1_ERROR_FIELD_MISSING	Mandatory field absent.
ACU_ASN1_ERROR_FIELD_UNEXPECTED	Field not in ASN.1 definition.
ACU_ASN1_ERROR_DEFN_ERROR	Supplied definition doesn't match message.
ACU_ASN1_ERROR_DEFN_MISMATCH	Supplied definition doesn't match values.
ACU_ASN1_ERROR_DEFN_NOT_FOUND	Item not found.

2.2.1.3 ASN.1 tag values

ASN.1 defines four classes of tags: 'Universal', 'Application specific', 'Context specific' and 'Private use', each of which can be qualified as 'constructed' – meaning that the data part is itself ASN.1 encoded. The data format for each of the Universal types is defined by the standard, the format for the other classes will be that of one of the Universal types – but can be 'octetstring' to allow for arbitrary data. The encoding scheme is independent of the class (only the interpretation of the data is defined for Universal). See also Appendix D:

The tag value the application passes to the encoder (and gets back from the decoder) has the 'class', the 'constructed' flag and 24 bits of tag value in a single 32bit quantity. The least significant 8 bits are the first (usually the only) byte written to the buffer, the most significant 24 contain the tag code itself (which is also in the least significant 5 bits for small values).

The following are defined to help the application handle the constants:

ACU_ASN1_UNI(code)	Primitive universal tag with value 'code'.
ACU_ASN1_APP(code)	Primitive application tag with value 'code'.
ACU_ASN1_CTX(code)	Primitive context specific tag with value 'code'.
ACU_ASN1_PRV(code)	Primitive private tag with value 'code'.
ACU_ASN1_CONS_UNI(code)	Constructed universal tag with value 'code'.
ACU_ASN1_CONS_APP(code)	Constructed application tag with value 'code'.
ACU_ASN1_CONS_CTX(code)	Constructed context specific tag with value 'code'.
ACU_ASN1_CONS_PRV(code)	Constructed private tag with value 'code'.
ACU_ASN1_GET_TAG_CODE(tag)	Returns tag code.
ACU_ASN1_GET_TAG_CLASS(tag)	Returns tag class.
ACU_ASN1_GET_TAG_CONSTRUCTED(tag)	Returns non-zero if the tag is 'constructed'.

The following constants are defined, the values match those from X.690:

Tag classes:

ACU_ASN1_TAG_UNIVERSAL	0x00 format defined by the standard.
ACU_ASN1_TAG_APPLICATION	0x40 format defined by the application.
ACU_ASN1_TAG_CONTEXT	0x80 format depends on the location of the tag.
ACU_ASN1_TAG_PRIVATE	0xc0 format defined elsewhere (used by ANSI TCAP).

Tag qualifier:

ACU_ASN1_TAG_CONSTRUCTED	0x20 data is ASN.1 BER encoded.

Universal tag codes:

ACU_ASN1_BOOLEAN	0x01
ACU_ASN1_INT	0x02
ACU_ASN1_BITSTRING	0x03
ACU_ASN1_OCTETSTRING	0x04
ACU_ASN1_NULL	0x05
ACU_ASN1_OBJ_ID	0x06
ACU_ASN1_OBJ_DESC	0x07
ACU_ASN1_EXTERNAL	0x08
ACU_ASN1_REAL	0x09
ACU_ASN1_ENUMERATED	0x0a
ACU_ASN1_SEQ	0x10
ACU_ASN1_SET	0x11

Note **Tag value above 2^24-1 are not supported.**

2.2.2 Common functions

2.2.2.1 acu_asn1_buf_init

```
acu_asn1_buf_t *acu_asn1_buf_init(acu_asn1_buf_t *a1b, const unsigned char
     *buf, unsigned int len, unsigned int flags)
```

Purpose

This function initialises the buffer control structure used by the ASN.1 encoder and decoder.

Parameters

`a1b`	Structure to initialise. If `NULL` the structure will be allocated using `malloc()`.
`buf`	Address of buffer to use. If `NULL` the buffer will be allocated using `malloc()` `buf` is defined `const` so that a const buffer can be passed to the decoder.
`len`	Number of bytes in buffer, initial size if `buf` is `NULL` (when the buffer will be extended as necessary).
`flags`	Bitwise OR of: `ACU_A1BF_REINIT`: re-use existing buffer. `ACU_A1BF_LOG_STERR`: report any errors directly to `stderr`. `ACU_A1BF_DEFINITE_LEN`: use fixed length encoding for all constructed items.

Return value

The address of an initialised `acu_asn1_buf_t` structure, or `NULL` if `malloc()` fails.

Note **Applications should get `acu_asn1_buf_init()` to allocate the structure so that they are not dependant upon the version of the header file used to compile the program.**

2.2.2.2 acu_asn1_buf_free

```
void acu_asn1_buf_free(acu_asn1_buf_t *a1b)
```

Purpose

This function frees any memory allocated to `a1b`, including `a1b` itself.

2.2.2.3 acu_asn1_strerror

```
const char *acu_asn1_strerror(int rval, unsigned int flags);
```

Purpose

This function returns a text string that describes an ASN.1 encoder/decoder error code.

Parameters

`rval`	ASN.1 encoder error number (one of `ACU_ASN1_ERROR_xxx`).
`flags`	`0` => return descriptive text. `1` => return the C name; one of the `"ACU_ASN1_ERROR_xxx"` strings.

Return value

A pointer to a static const string describing the error, unless the error number is unknown in which case the address of a static array filled with the text “error %d unknown” is returned.

The error text strings are defined by the `ACU_ASN1_ERRORS` define in `tcap_asn1_codec.h`.

2.2.2.4 acu_asn1_fmt_errmsg

```
void acu_asn1_fmt_errmsg(acu_asn1_buf_t *a1b, char *buf, int buflen);
```

Purpose

This function writes a description of the first decode (or encode) error from `a1b` into `buf`.

Parameters

`a1b`	Control structure on which error occurred.
`buf`	Buffer to contain error message, will be ‘\0’ terminated.
`buflen`	Size of `buf` in bytes, suggested minimum 160 bytes.

2.2.3 ASN.1 Encoder Functions

The ASN.1 encoder functions all add the given item to the buffer. They correctly encode the `tag` and `length` bytes, and then copy in the user specified data.

These functions return an error on failure, but are very unlikely to fail except due to coding errors or if `malloc()` fails. Instead of checking the result of each call, the program can check whether `a1b->a1b_error` is non-zero after completing the encoding.

The library remembers the number of constructed items, and the start point for a small number (currently 16) so that constructed items can use the definite length format.

Many of the encoding routines are equivalent to calls to `acu_asn1_put_octetstring()` but have different arguments; all can be used for any ASN.1 data type.

The tag values should be generated using one of the `ACU_ASN1_xxx(code)` defines.

Appendix D: contains a brief description of the format of ASN.1 BER encoded data.

2.2.3.1 acu_asn1_put_constructed

```
int acu_asn1_put_constructed(acu_asn1_buf_t *a1b, unsigned int tag);
```

Purpose

This function starts the encoding of a constructed item, writing the ASN.1 tag byte(s) and an indefinite length mark (which may be overwritten with the actual length by `acu_asn1_put_end_constructed`).

The `ACU_ASN1_TAG_CONSTRUCTED` bit is always set in the supplied tag.

Parameters

`a1b`	Control structure for the request.
`tag`	Tag to be encoded.

Return value

Zero if successful, `ACU_ASN1_ERROR_xxx` on failure.

2.2.3.2 acu_asn1_put_end_constructed

```
int acu_asn1_end_constructed(acu_asn1_buf_t *a1b);
```

Purpose

This function terminates the encoding of a constructed item, either replacing the indefinite length written by `acu_asn1_put_constructed()` with the actual length, or writing an indefinite length terminator.

Normally the definite length encoding is used for lengths less than 128, and the indefinite length encoding for longer items.

If the `ACU_A1BF_DEFINITE_LEN` flag is set in `a1b_flags`, and the length is greater than 127, then the data will be copied down the buffer to allow a multi-byte definite length field be written. This gives a shorter encoding for lengths from 128 to 255 bytes.

Note **TCAP applications can use the `enc_def_len` configuration parameter to set this flag.**

Parameters

`a1b`	Control structure for the request.

Return value

Zero if successful, `ACU_ASN1_ERROR_xxx` on failure.

2.2.3.3 acu_asn1_put_end_all_constructed

```
int acu_asn1_end_all_constructed(acu_asn1_buf_t *a1b, int depth);
```

Purpose

This function terminates the encoding of multiple constructed items. It calls `acu_asn1_end_constructed()` until there are `depth` levels of constructed items.

Parameters

`a1b`	Control structure for the request.
`depth`	Required depth of construction.

Return value

Zero if successful, `ACU_ASN1_ERROR_xxx` on failure.

2.2.3.4 acu_asn1_put_int

```
int acu_asn1_put_int(acu_asn1_buf_t *a1b, unsigned int tag, int value);
```

Purpose

This function encodes a signed integer. One, two, three or four bytes may be needed depending on the actual value.

Parameters

`a1b`	Control structure for the request.
`tag`	Tag to be encoded.
`value`	Value to be encoded.

Return value

Zero if successful, `ACU_ASN1_ERROR_xxx` on failure.

2.2.3.5 acu_asn1_put_unsigned

```
int acu_asn1_put_unsigned(acu_asn1_buf_t *a1b, unsigned int tag, unsigned int
    value, unsigned int length);
```

Purpose

This function encodes the least significant bits of an unsigned 32bit integer in the specified number of bytes.

High bits of the supplied value are silently ignored.

Parameters

`a1b`	Control structure for the request.
`tag`	Tag to be encoded.
`value`	Value to be encoded.
`length`	Number of bytes to encode (`0` to `4`).

Return value

Zero if successful, `ACU_ASN1_ERROR_xxx` on failure.

2.2.3.6 acu_asn1_put_octet_8

```
int acu_asn1_put_octet_8(acu_asn1_buf_t *a1b, unsigned int tag, unsigned int
    val1, unsigned int val2);
```

Purpose

This function encodes two 32bit integers as eight bytes.

This function exists in order to encode ANSI transaction-ids.

Parameters

`a1b`	Control structure for the request.
`tag`	Tag to be encoded.
`val1`	First 32 bits of value to be encoded.
`val2`	Second 32 bits of value to be encoded.

Return value

Zero if successful, `ACU_ASN1_ERROR_xxx` on failure.

2.2.3.7 acu_asn1_put_bits32

```
int acu_asn1_put_bits32(acu_asn1_buf_t *a1b, unsigned int tag, unsigned int
    val, unsigned int len);
```

Purpose

This function encodes a bitstring of 1 to 32 bits from an integer value.

Note **In line with the ASN.1 specification, the first bit has value 0x80.**

Parameters

`a1b`	Control structure for the request.
`tag`	Tag to be encoded.
`val`	Data to be encoded.
`len`	Number of bits to be encoded (1 to 32).

Return value

Zero if successful, `ACU_ASN1_ERROR_xxx` on failure.

2.2.3.8 acu_asn1_put_bitstring

```
int acu_asn1_put_bitstring(acu_asn1_buf_t *a1b, unsigned int tag, const void
    *buf, unsigned int len);
```

Purpose

This function encodes the given data as a bitstring.

Parameters

`a1b`	Control structure for the request.
`tag`	Tag to be encoded.
`buf`	Pointer to data to be encoded.
`len`	Number of bits to be encoded.

Return value

Zero if successful, `ACU_ASN1_ERROR_xxx` on failure.

2.2.3.9 acu_asn1_put_octetstring

```
int acu_asn1_put_octetstring(acu_asn1_buf_t *a1b, unsigned int tag, const
    void *buf, unsigned int len);
```

Purpose

This function encodes the given data as an octetstring.

The constructed bit is not cleared from the tag, so this function can be used to add a constructed item from a buffer containing its fields.

Parameters

a1b	Control structure for the request.
tag	Tag to be encoded.
buf	Pointer to data to be encoded.
len	Number of bytes to be encoded.

Return value

Zero if successful, ACU_ASN1_ERROR_xxx on failure.

2.2.3.10 acu_asn1_put_raw_octets

```
int acu_asn1_put_raw_octets(acu_asn1_buf_t *a1b, const void *buf, unsigned
    int len);
```

Purpose

This function copies the given data into the target buffer. The caller is responsible for ensuring that it is valid ASN.1

Parameters

a1b	Control structure for the request.
buf	Pointer to data to be copied.
len	Number of bytes to be copied.

Return value

Zero if successful, ACU_ASN1_ERROR_xxx on failure.

2.2.3.11 acu_asn1_put_space

```
void *acu_asn1_put_space(acu_asn1_buf_t *a1b, unsigned int len);
```

Purpose

This function allocates space in the target buffer, returning a pointer to the space. The caller can then copy the required ASN.1 into the buffer area.

Note **The address returned must not be used after any other encoding function is called.**

Parameters

a1b	Control structure for the request.
len	Number of bytes to be allocate.

Return value

Pointer to the allocated space if successful, NULL on failure.

2.2.3.12 acu_asn1_encode_object_id_str/int

```
int acu_asn1_encode_object_id_str(unsigned char *buf, int buflen, const char
    *id_str);
int acu_asn1_encode_object_id_int(unsigned char *buf, int buflen, ...);
```

Purpose

These functions convert an ASN.1 object identifier to its binary form in the user-supplied buffer.

`acu_asn1_encode_object_id_str()` converts from a string of numbers separated by dots, `acu_asn1_encode_object_id_int()` converts from the list of numbers passed as arguments, terminating on a argument of ~0u.

For example: both:

```
acu_asn1_encode_object_id_str(buf, sizeof buf, "0.0.17.773.1.1.1")
```

and

```
acu_asn1_encode_object_id_int(buf, sizeof buf, 0, 0, 17, 773, 1, 1, 1, ~0u)
```

generate the same 7 bytes `{0x0, 0x11, 0x86, 0x5, 0x1, 0x1, 0x1}` of a TCAP 'Dialogue-as-id' (see table 37/Q.773).

The encoded object identfier does not contain an ASN.1 type and length, these will normally be added by a call to `acu_asn1_put_octetstring()`.

Parameters

`buf`	Buffer area in which to write the encoded object identifier.
`buflen`	Maximum size of encoded object identifier.
`id_str`	String form of object identifier to convert.

Return value

The number of bytes written into `buf`. If the encoded object identifier is longer than `maxlen` then it is silently truncated.

2.2.4 ASN.1 Decoder functions

The ASN.1 decoder functions process an input buffer sequentially (each function advances the read pointer). The maximum depth of constructed items is limited to 32, of which 16 may have definite length. Deeply constructed ASN.1 can be decoded in stages by using `acu_asn1_get_reference()`.

Any ASN.1 formatting errors (e.g.: a length that spans the end of the buffer) are reported by the individual routines. The first such error is saved in the `a1b_error` fields of the `acu_asn1_buf_t` structure.

2.2.4.1 acu_asn1_get_tag_len

```
int acu_asn1_get_tag_len(acu_asn1_buf_t *a1b, int *length);
```

Purpose

This function decodes the header of the next ASN.1 item.

Follow with a call to one of the other `acu_asn1_get_xxx()` functions to obtain the item's value, or call `acu_asn1_get_tag_len()` again to decode the contents of a constructed item.

The returned `tag` value is encoded as if by the `ACU_ASN1_xxx(code)` or `ACU_ASN1_xxx_CONS(code)` defines. The `ACU_ASN1_GET_TAG_xxx(tag)` defines can be used to extract the sub-fields (e.g., for diagnostic prints).

Parameters

`a1b`	Control structure for the request.
`length`	Address of where the item length will be written to. This will be `ACU_ASN1_INDEFINITE_LENGTH` for indefinite length constructed items.

Return value

Zero with length zero for end of constructed item, zero with length -1 for end of buffer, `ACU_ASN1_ERROR_xxx` on failure, otherwise the tag value for the next item.

2.2.4.2 acu_asn1_get_reference

```
int acu_asn1_get_reference(acu_asn1_buf_t *a1b, int data_only, const unsigned
    char **bufptr);
```

Purpose

This function returns the address and length of an item. The item may be primitive or constructed.

Parameters

`a1b`	Control structure for the request.
`data_only`	If non-zero `bufptr` will point to the first data byte, if zero it will point to the first byte of the ASN.1 type field.
`bufptr`	Address of where a pointer to the data will be written.

Return value

Length of the item if successful, `ACU_ASN1_ERROR_xxx` on failure.

2.2.4.3 acu_asn1_get_int

```
int acu_asn1_get_int(acu_asn1_buf_t *a1b, int *value);
```

Purpose

This function obtains a signed integer value. The length of the ASN.1 field must be between 1 and 4.

Parameters

a1b	Control structure for the request.
value	Address of where the data value will be written.

Return value

Zero if successful, ACU_ASN1_ERROR_xxx on failure.

2.2.4.4 acu_asn1_get_unsigned

```
int acu_asn1_get_unsigned(acu_asn1_buf_t *a1b, unsigned int *value);
```

Purpose

This function obtains a zero to four-byte value as an unsigned 32-bit integer.

Parameters

a1b	Control structure for the request.
value	Address of where the data value will be written.

Return value

Zero if successful, ACU_ASN1_ERROR_xxx on failure.

2.2.4.5 acu_asn1_get_octet_8

```
int acu_asn1_get_octet_8(acu_asn1_buf_t_t *a1b, unsigned int *val_1, unsigned
    int *val_2);
```

Purpose

This function obtains an eight-byte value as two 32-bit integers.

Parameters

a1b	Control structure for the request.
val_1	Address of where the first 4 bytes of the data value will be written.
val_2	Address of where the second 4 bytes of the data value will be written.

Return value

Zero if successful, ACU_ASN1_ERROR_xxx on failure.

2.2.4.6 acu_asn1_get_octetstring

```
int acu_asn1_get_octetstring(acu_asn1_buf_t *a1b, void *buf, unsigned int
    buflen);
```

Purpose

This function copies the parameter to the user-supplied buffer.

Parameters

a1b	Control structure for the request.
buf	Address of the buffer where the data will be written.
buflen	Length of the buffer.

Return value

Length of the parameter if successful, ACU_ASN1_ERROR_xxx on failure.

2.2.4.7 acu_asn1_get_bits32

```
int acu_asn1_get_bits32(acu_asn1_buf_t *a1b, unsigned int *value);
```

Purpose

This function obtains the value of an ASN.1 bitfield that contains 1 to 32 bits returning the value as an unsigned integer with zeros for all the absent bits.

Note **In line with the ASN.1 specification, the first bit has value 0x80.**

Parameters

`a1b` Control structure for the request.
`value` Address of where the data value will be written.

Return value

Zero if successful, `ACU_ASN1_ERROR_xxx` on failure.

2.2.4.8 acu_asn1_decode_object_id_str/int

```
int acu_asn1_decode_object_id_str(char *tgt, unsigned int tgtlen, const
     unsigned char *obj, unsigned int objlen);
int acu_asn1_decode_object_id_int(unsigned int *tgt, unsigned int tgtlen,
     const unsigned char *obj, unsigned int objlen);
```

Purpose

These functions convert the binary form of an ASN.1 object identifier to a NUL terminated string or an array of integer values.

Parameters

`tgt` Address of the buffer where the decoded data will be written.
`tgtlen` Number of entries in the target array.
`obj` Address of the object identifier.
`objlen` Length of the object identifier.

Return value

On success `acu_asn1_decode_object_id_str()` returns zero and `acu_asn1_decode_object_id_int()` returns the number of values written to `tgt`. On failure both return one of the `ACU_ASN1_ERROR_xxx` values.

2.3 Message definition based ASN.1 encoder and decoder

The functional encoder and decoder described in section 2.2 are simple and efficient, however they do have drawbacks especially when trying to ensure that the ASN.1 matches that from a protocol standard (e.g. GSM MAP). This is particularly true of the decoder, where the entire message often needs to be validated before any of it is processed, and the sequence of function calls to do this gets long and error-prone.

2.3.1 Codec data structures

The message definition based codec functions use a readonly message definition data structure to define the format of an ASN.1 message. The encode and decode functions are engines that use the definition to convert between a C structure and bytestream ASN.1.

To make this reliable the C structure and message definition must match exactly. Doing this by hand would be very error prone so they are generated from the same source text using pre-processor 'magic'.

Appendix E: gives a short explanation of the C pre-processor features used.

There are some examples in `$ACULAB_ROOT/ss7/sample_code/tcap/asn1`.

2.3.1.1 ASN.1 message definitions

Each ASN.1 message is described by a C pre-processor `#define` containing the sequence of desired fields:

```
#define msg_name(action) \
    action(FIELD_TYPE, (field_name, arguments)) \
    action(FIELD_TYPE, (field_name, arguments)) \
    ...
```

Each line (but notice the line continuations) describes a single ASN.1 field, the start or end of a sequence/sequence of/choice, or is a reference to another message definition.

The *`field_name`* will be used for the name of a C struct/union member and may be printed in error messages. The *`arguments`* usually include the `asn1_tag` (which has `ACU_ASN1_` prepended, so will be `CTX(n)`, `INT`, `OCTETSTRING` etc), and `flags` which should be `0` or `OPTIONAL` (additional flag values may be defined in the future).

The valid *`FIELD_TYPE`*s and their arguments are:

Primitive numeric fields, encode to/from `a1v_value`, encoded if either `a1v_value` or `a1v_length` non-zero.

`INTEGER` `(field_name, asn1_tag, flags)`
: Signed integer, encoded in 1 to 4 bytes (regardless of `a1v_length`) depending on the value.

`UNSIGNED` `(field_name, asn1_tag, flags)`
: Unsigned integer, 0 to 4 bytes.

`BOOLEAN` `(field_name, asn1_tag, flags)`
: Single byte value 0 or 1 (all non-zero values encode and decode as 1).

`NULL` `(field_name, asn1_tag, flags)`
: Zero length item. If `asn1_tag` needs to be `NULL`, use `UNI_NULL` to avoid `NULL` being expanded to `((void *)0)`.

`BITS32` `(field_name, asn1_tag, bit_width, flags)`
: Bit field with a default `bit_width` (0 to 32) bits. If `a1v_length` is greater than 32 the encoder uses `a1v_length - 32` bits. If `a1v_value` has high bits set, then the width will be increased (to 8, 16, 24 or 32) in order to encode the full value.

Buffers, encoded from `a1v_data` if it is not `NULL`.

`OCTETS` `(field_name, asn1_tag, flags)`
: Binary data encoded with the specified tag.

`RAW` `(field_name, asn1_tag, flags)`
: An ASN.1 constructed item with the specified tag (the data excludes the tag).

`ANY` `(field_name, flags)`

A single (primitive or constructed) ASN.1 item with any tag value (the data includes the tag).

Non-primitive types, the *field_name* in the END line must match.

SEQ (*field_name*, asn1_tag, flags)
END_SEQ (*field_name*, end_flags)

This pair enclose the fields of a constructed item. Set end_flags to EXTENDABLE to ignore additional fields (use when the ASN.1 definition ends with an ellipsis).

SEQ_OF (*field_name*, asn1_tag, flags)
END_SEQ_OF (*field_name*, max_reps, 0)

This pair defines and array with max_reps elements to hold the data for a 'SEQUENCE SIZE (1..max_reps) OF'. The enclosed item must be a single ASN.1 entity, either primitive, a sequence, or a choice.

CHOICE (*field_name*, flags)
END_CHOICE (*field_name*, 0)

This pair define the fields of a 'choice'. Only one of the enclosed fields can exist in the actual ASN.1.

References to previously defined message definitions.

REF (*field_name*, *ref_msg_name*, flags)

A reference to another ASN.1 message definition is included at the current location. The referenced item can be a single field, a sequence or a choice.

IMPLICIT (*field_name*, asn1_tag, *ref_msg_name*, flags)

As REF except the item is encoded with the tag specified.

DIR_REF (*field_name*, *ref_msg_name*, flags)
DIR_IMPL (*field_name*, asn1_tag, *ref_msg_name*, flags)

The definition of another ASN.1 message is expanded at the current location. The referenced item must be a single field (not a sequence or choice), and, for DIR_REF, not ANY or BITS32. These remove the C struct added by REF and IMPLICIT.

Primitive fields that are marked OPTIONAL will only be encoded if they contain user-specified values (a1v_data non-NULL for OCTETS, RAW and ANY, otherwise a1v_length or a1v_integer non-zero). Optional sequences and choices will not be encoded unless they contain at least one field that would be encoded if all their fields were marked OPTIONAL.

A message definition must define a single ASN.1 field; it will either have just one field definition, or the first will be a SEQ, SEQ_OF or CHOICE (and the last one the matching end).

As an optimisation the outer SEQ/SEQ_END can be omitted provided the definition is expanded with ACU_A1T_SEQ_*xxx*() instead of ACU_A1T_*xxx*(). This optimisation doesn't change the binary layout of the data structures, but saves a lot of typing.

2.3.1.2 Defining the message definition data

The structure for actual data that describes the message must be defined. Since this is initialised data it would normally be defined in a .c (or .cpp) file.

The name of the data area is that of the message definition with _defn appended. The upper case #define forms of the codec functions append this for you.

Depending on the definition, expand one of the following:

const ACU_A1T_DEFN(*msg_name*);

expands to const acu_a1t_defn_t *msg_name*_defn[] = { ... }; containing the message definition data.

const ACU_A1T_SEQ_DEFN(*msg_name*);

as ACU_A1T_DEFN(*msg_name*) but adds the definitions for the omitted encapsulating SEQ/END_SEQ.

const ACU_A1T_SEQ_EXT_DEFN(*msg_name*);

as ACU_A1T_SEQ_DEFN(*msg_name*) but sets the ACU_A1T_EXTENDABLE flag in the END_SEQ().

If the message definition contains REF or IMPLICIT then the referenced structure needs to be declared earlier in the source file.

`ACU_A1T_EXTERN_DEFN(msg_name);` will generate the correct extern statement.

2.3.1.3 Defining the associated C structure

The C structure for the values of a message is defined by passing the name of the definition to `ACU_A1T_TYPE()` (or `ACU_A1T_SEQ_TYPE()` if the outer `SEQ`/`SEQ_END` were omitted).

`ACU_A1T_TYPE(msg_name)` expands to `struct msg_name { ... }` where the structure members depend on `msg_name`.

Most of the `x(FIELD_TYPE, (arguments))` expand to: `acu_a1t_value_t field_name;` `acu_a1t_value_t` has the following members:

`const unsigned char`	`*a1v_data`	Pointer to data to encode (non-numeric fields) / decoded data (field name for `CHOICE`).
`unsigned int`	`a1v_length`	Length of encoded / decoded data.
`int`	`a1v_integer`	Numeric value, (ASN.1 tag for `CHOICE`).

The exceptions are:

`REF` and `IMPLICIT` expand to: `struct ref_msg_name field_name;`

`SEQ` and `SEQ_OF` expand to: `struct { acu_a1t_value_t field_name_seq;`

The decoder sets `field_name_seq` to reference the entire item.

The encoder will stop processing a `SEQ_OF` after an entry that encodes no actual data.

`END_SEQ` expands to: `} field_name;`.

`END_SEQ_OF` expands to: `} field_name[max_reps];`.

`CHOICE` expands to: `acu_a1t_value_t field_name_choice; union {`

The field `field_name_choice` is the discriminator for the union. The decoder sets the `a1v_integer` field to the ASN.1 tag and the `a1v_data` to the name of the selected field. The encoder will use the `a1v_data` or, if `NULL`, the `a1v_integer` field to determine what to encode.

`END_CHOICE` expands to: `} field_name;`.

The codec functions treat these structures as arrays of `acu_a1t_value_t;` this means that the function argument requires a cast, the upper case #define forms of the codec functions contain the required cast.

`ACU_A1T_TYPE(msg_name)` and `ACU_A1T_SEQ_TYPE(msg_name)` only differ in that the latter adds an additional member `acu_a1t_value_t field_name_seq;` at the start of the structure.

The nested struct and union types (for `SEQ`, `SEQ_OF` and `CHOICE`) are normally unnamed, specifying `#define ACU_A1T_NAMED_STRUCTS` before including the asn1 header file causes them to be named (allowing pointer types be defined).

The union member (added by `END_CHOICE`) is named. For C++ (or C if the compiler supports anonymous unions) the union can be made anonymous by specifying `#define ACU_A1T_ANON_UNION` before including the asn1 header file.

The `ACU_A1T_TYPE(msg_name)` expansion would typically follow the definition of `msg_name` in the application's `.h` file.

2.3.1.4 Example definition

A simple example will help explain things:

Q.773 defines a TCAP OPERATION (effectively) as:

```
OPERATION ::= CHOICE {
    localValue INTEGER,
    globalValue OBJECT_IDENTIFIER }
```

and Invoke as:

```
Invoke ::= SEQUENCE {
    invokeID      INTEGER,
    linkedID      [0] IMPLICIT INTEGER OPTIONAL,
    operationCode OPERATION,
    parameter     ANY DEFINED BY operationCode OPTIONAL }
```

These can be converted mechanically to:

```
#define TCAP_OperationCode(x) \
    x(CHOICE, (operationCode, 0)) \
        x(INTEGER, (localValue, INT, 0)) \
        x(OCTETS,  (globalValue, OBJ_ID, 0)) \
    x(END_CHOICE, (operationCode, 0))

#define TCAP_Invoke(x) \
    x(INTEGER, (invokeID, INT, 0)) \
    x(INTEGER, (linkedID, CTX(0), OPTIONAL)) \
    x(REF, (operationCode, TCAP_OperationCode, 0)) \
    x(ANY, (parameter, OPTIONAL))
```

The expansions of

```
ACU_A1T_TYPE(TCAP_OperationCode);
ACU_A1T_SEQ_TYPE(TCAP_Invoke);
```

give us the C structs:

```
struct TCAP_OperationCode {
    acu_a1t_value_t            operationCode_choice;
    union {
        acu_a1t_value_t            localValue;
        acu_a1t_value_t            globalValue;
    } operationCode;
};

struct TCAP_Invoke {
    acu_a1t_value_t            TCAP_Invoke_seq;
    acu_a1t_value_t            invokeID;
    acu_a1t_value_t            linkedID;
    struct TCAP_OperationCode  operationCode;
    acu_a1t_value_t            parameter;
};
```

which can easily be filled and inspected.

`acu_asn1_trace_data()` can be used to print out the C structure hierarchy.

The application will also need to expand `ACU_A1T_DEFN(TCAP_OperationCode);` and `ACU_A1T_SEQ_DEFN(TCAP_Invoke);` but can ignore the contents.

2.3.2 API functions

These functions interpret the message definition (treating it like a program) using the other parameters as data. The structure containing the values to be encoded/decoded is defined by the application using `ACU_A1T_TYPE()` (or `ACU_A1T_SEQ_TYPE()`) but is treated internally as an array of `acu_a1t_defn_t` this usually requires a cast on the API calls. The structure's size is also passed in order to perform consistency tests.

An upper case version of these functions is provided (as a `#define`) that contains the required casts, any `sizeof` requests and appends `_defn` to the name of the definition.

2.3.2.1 acu_asn1_encode_data

```
int acu_asn1_encode_data(acu_asn1_buf_t *a1b, const acu_a1t_defn_t *defn,
     const acu_a1t_value_t *value, int value_len);
int ACU_ASN1_ENCODE_DATA(acu_asn1_buf_t *a1b, const acu_a1t_defn_t *defn,
     const struct xxx *value);
```

Purpose

This function encodes the information from `value` into the buffer associated with the `a1b` structure using the message definition in `defn`.

The `a1b` buffer must be in a valid state for the encode functions (see 2.2.3) to be called.

Parameters

`a1b`	Control structure for the request, encoded data written at `a1b->a1b_ptr`.
`defn`	Definition of ASN.1 message, generated by `ACU_A1T_DEFN()` or `ACU_A1T_SEQ_DEFN()`.
`value`	Pointer to structure containing values to encode, the passed structure must be of the type defined by expanding `ACU_A1T_TYPE()` or `ACU_A1T_SEQ_TYPE()` on the same define that generated `defn`.
`value_len`	`sizeof *value`, used to verfiy that `value` and `defn` match.

Return value

Zero on success; the `a1b->a1b_ptr` is advanced past encoded data. On failure one of the `ACU_ASN1_ERROR_xxx` values.

If The function reports an error use `acu_asn1_fmt_errmsg()` (see 2.2.2.4) to get a printable explanation of the error.

2.3.2.2 acu_asn1_decode_data

```
int acu_asn1_decode_data(acu_asn1_buf_t *a1b, const acu_a1t_defn_t *defn,
     acu_a1t_value_t *value, int value_len);
int ACU_ASN1_DECODE_DATA(acu_asn1_buf_t *a1b, const acu_a1t_defn_t *defn,
     struct xxx *value);
```

Purpose

This function decodes from the `a1b` structure into `value` using the message definition in `defn`.

Normally preceded by a call to `acu_asn1_buf_init()` that specifies the bounds of a receive message, but may also be used to decode something that has just been encoded.

Parameters

`a1b`	Control structure for the request. Data from `a1b->a1b_buf` to either `a1b->a1b_ptr` or `a1b->a1b_end` is decoded.
`defn`	Definition of ASN.1 message, generated by `ACU_A1T_DEFN()` or `ACU_A1T_SEQ_DEFN()`.
`value`	Pointer to structure to contain the decoded values, the passed structure must be of the type defined by expanding `ACU_A1T_TYPE()` or `ACU_A1T_SEQ_TYPE()` on the same data as `defn`.
`value_len`	`sizeof *value`, used to verfiy that `value` and `defn` match.

Return value

Zero on success. On failure one of the ACU_ASN1_ERROR_xxx values.

If the function reports an error use acu_asn1_fmt_errmsg() (see 2.2.2.4) to get a printable explanation of the error.

2.3.2.3 acu_asn1_trace_data

```
char *acu_asn1_trace_data(const acu_a1t_defn_t *defn,
     const acu_a1t_value_t *value, int value_len, const char *hdr,
     unsigned int flags);
char *ACU_ASN1_TRACE_DATA(const acu_a1t_defn_t *defn,
     const struct xxx *value, const char *hdr, unsigned int flags);
```

Purpose

This function generates a textual representation of the ASN.1 defined by defn, with the numeric values from value.

This function is useful for checking that the definitions are correct and that the application is correctly coded as well as for tracing sent and received messages.

Parameters

defn	Definition of ASN.1 message, generated by ACU_A1T_DEFN() or ACU_A1T_SEQ_DEFN().
value	Pointer to structure containing the values to trace, the passed structure must be of the type defined by expanding ACU_A1T_TYPE() or ACU_A1T_SEQ_TYPE() on the same data as defn.
value_len	sizeof *value, used to verfiy that value and defn match.
hdr	Text added to the start or every trace line, may be NULL.
flags	Bitwise 'or' of the following:

ACU_A1BF_TRACE_HEX	Full hexdump (not just first 16 bytes).
ACU_A1BF_TRACE_UNSET	Trace fields that would not be encoded (no value or unselected fields of choice).
ACU_A1BF_TRACE_SEQS	Trace all start of SEQ/SEQ_OF markers.
ACU_A1BF_TRACE_ENDS	Trace 'end' markers.
ACU_A1BF_TRACE_NO_REF	Don't trace REF/IMPLICIT, enables TRACE_SEQS.
ACU_A1BF_TRACE_RAW	Trace some values in hex (for debugging the library).

Setting ACU_A1BF_TRACE_UNSET with an all zero byte value buffer will show the C struct/union names – useful for working out the field names themselves.

Setting ACU_A1BF_TRACE_NO_REF removes the traces for the C structures (adding in those for SEQ) and gives a trace that is close to the original definition of the ASN.1. These are the names that can be searched for with acu_asn1_find_defn().

Return value

Address of a malloced buffer containing the trace data, might be NULL if malloc() fails. The caller must call acu_asn1_free_trace_data() to free the buffer.

Any error message from the encoder (used to generate the trace text) will be appended to the output.

2.3.2.4 acu_asn1_free_trace_data

```
void acu_asn1_free_trace_data(char *trace_data);
```

Purpose

This frees the buffer allocated by acu_asn1_trace_data().

Note **On windows, directly calling `free(trace_data)` will corrupt the `malloc` arena.**

Parameters

trace_data	Buffer to free.

Return value

None.

2.3.2.5 acu_asn1_find_defn

```
int acu_asn1_find_defn(const acu_a1t_defn_t *defn,
     const acu_a1t_value_t *value, const char *name, unsigned int ndx ,
     const acu_a1t_defn_t **found_defn, const acu_a1t_value_t **found_value);
int ACU_ASN1_FIND_DEFN(const acu_a1t_defn_t *defn, const struct xxx *value,
     const char *name, unsigned int ndx , const acu_a1t_defn_t **found_defn,
     const acu_a1t_value_t **found_value);
```

Purpose

This function searches through the definition of an ASN.1 `SEQ`, `SEQ_OF` or `CHOICE` looking for a field with the requested name.

If the requested name is that of the definition itself, then the end of the definition will be located. This can be used to determine the size of the `value` structure. For `SEQ_OF` this will find the end of the `ndx`'th element, set `ndx` to `~0u` to find the end of the last one.

Locating fields by name is rather less efficient than indexing the C structure, but can be useful in scripted and diagnostic tools.

The `value` parameter will typically need a cast since the function treats the C struct that the application provides as an array.

Note **The `value` parameter isn't dereferenced so could be `NULL` if the intention is to find the offset of a field or the size of the required data item.**

Parameters

`defn`	Definition of ASN.1 message, generated by `ACU_A1T_DEFN()` or `ACU_A1T_SEQ_DEFN()`.
`value`	Pointer to structure containing the associated values, the passed structure must be of the type defined by expanding `ACU_A1T_TYPE()` or `ACU_A1T_SEQ_TYPE()` on the same data as `defn`.
`name`	Field to search for, if `NULL` the definition that `found_value` references will be located (not necessarily very useful).
`ndx`	If the located item is a `SEQ_OF` this will index the sequence, otherwise ignored.
`found_defn`	On success will contain the address of definition information for the parameter. May be `NULL` is the information isn't wanted.
`found_value`	On success will contain the address of value information for the parameter. May be `NULL` is the information isn't wanted.

Return value

Zero on success. On failure one of the `ACU_ASN1_ERROR_xxx` values.

If successful the `found_value` field can be used to read or write the value of a primitive ASN.1 field. The `found_value` and `found_defn` fields can be passed to `acu_asn1_find_defn()` to search inside a constructed item.

2.4 Thread support functions

Support functions are provided for multi-threaded applications. They provide an operating independent interface to threads and thread synchronization functions.

Some of the functions are actually `#defines` within the header file `tcap_synch.h`. Because of this, the function arguments may be evaluated more than once.

Additional error information may be available in an operating system dependant manner (e.g.: by inspecting `errno`).

These functions are used within the TCAP library itself. They are exposed by its interface, and portable applications may decide to use them internally.

On Linux systems the functions use the pthread library routines.

Note **Do not cancel threads that are using the TCAP library.**

2.4.1 Mutex functions

Mutexes are used to protect data areas from concurrent access by more than one thread.

The mutex functions are non-recursive under Linux. Under Windows an error message will be output to `stderr` if a mutex is acquired recursively.

On Windows systems mutexes are implemented using the critical-section functions so that acquiring an uncontested mutex does not require a system call.

2.4.1.1 acu_tcap_mutex_create

```
int acu_tcap_mutex_create(acu_tcap_mutex_t *mutex);
```

Purpose

This function initialises the mutex, allocating any operating system resources needed.

Parameters

`mutex` Address of the mutex to initialise.

Return value

Zero on success, `-1` on failure.

2.4.1.2 acu_tcap_mutex_delete

```
void acu_tcap_mutex_delete(acu_tcap_mutex_t *mutex);
```

Purpose

This function frees all the operating system resources associated with the mutex. The mutex must not be locked when it is deleted.

Parameters

`mutex` Address of the mutex delete.

2.4.1.3 acu_tcap_mutex_lock

```
int acu_tcap_mutex_lock(acu_tcap_mutex_t *mutex);
```

Purpose

This function locks the mutex. If the mutex is already locked the thread will block until the mutex is unlocked.

Parameters

`mutex` Address of the mutex to lock.

Return value

Zero on success, `-1` on failure.

2.4.1.4 acu_tcap_mutex_trylock

```
int acu_tcap_mutex_trylock(acu_tcap_mutex_t *mutex);
```

Purpose

This function attempts to lock the mutex. If the mutex is already locked then it will return immediately with a non-zero return value.

Parameters

`mutex` Address of the mutex to lock.

Return value

Zero on success, `-1` on failure.

2.4.1.5 acu_tcap_mutex_unlock

```
void acu_tcap_mutex_unlock(acu_tcap_mutex_t *mutex);
```

Purpose

This function unlocks the mutex. A mutex can only be unlocked by the thread that locked it

Parameters

`mutex` Address of mutex to unlock.

2.4.2 Condition variable functions

Condition variables allow one thread to wait until signalled by a different thread. To avoid timing windows all accesses to a condition variable must be protected by the same mutex.

Under Windows, a condition variable is implemented using two manual reset events that are used alternately, with the last thread to exit resetting the event. This avoids any problems associated with `PulseEvent()` and kernel mode APC. It also allows the mutex to be implemented using the critical section functions – avoiding a system call when the mutex is available.

2.4.2.1 acu_tcap_condvar_create

```
int acu_tcap_condvar_create(acu_tcap_cond_t *condvar);
```

Purpose

This function initialises the condition variable, allocating any operating system resources needed.

Parameters

`condvar` Address of the condition variable to initialise.

Return value

Zero on success, `-1` on failure.

2.4.2.2 acu_tcap_condvar_delete

```
void acu_tcap_condvar_delete(acu_tcap_cond_t *condvar);
```

Purpose

This function frees the operating system resources allocated to the condition variable. No threads must be waiting for a condition variable when it is deleted.

Parameters

`condvar` Condition variable to delete.

2.4.2.3 acu_tcap_condvar_wait

```
int acu_tcap_condvar_wait(acu_tcap_cond_t *condvar, acu_tcap_mutex_t *mutex);
```

Purpose

This function waits for the condition variable to be signalled. The `mutex` is released atomically with the wait and re-acquired before the function returns.

Parameters

`condvar`	Condition variable to wait for.
`mutex`	Mutex associated with this condvar.

Return value

Zero on success, `-1` on failure.

2.4.2.4 acu_tcap_condvar_wait_tmo

```
int acu_tcap_condvar_wait_tmo(acu_tcap_cond_t *condvar,
     acu_tcap_mutex_t *mutex, int millisecs);
```

Purpose

This function waits for the condition variable to be signalled. If the condition variable isn't signalled within the specified timeout it will return `-1`.

Note **Since the `pthread_cond_timedwait()` takes an absolute timeout the Linux code has to add the current time to the timeout. The pthread implementation almost certainly subtracts it before sleeping!**

Parameters

`condvar`	Condition variable to wait for.
`millisecs`	The maximum time to wait in milliseconds.
`mutex`	Mutex associated with this condvar.

Return value

Zero on success, `-1` on failure or if the wait times out.

2.4.2.5 acu_tcap_condvar_broadcast

```
int acu_tcap_condvar_broadcast(acu_tcap_cond_t *condvar);
```

Purpose

This function signals the condition variable. All threads blocked in `acu_tcap_condvar_wait()` or `acu_tcap_condvar_wait_tmo()` on the specified condition variable are woken up.

The calling thread must hold the mutex associated with the condvar.

Parameters

`condvar`	Address of the condition variable to signal.

Return value

Zero on success, `-1` on failure.

2.4.3 Thread functions

2.4.3.1 acu_tcap_thread_create

```
int acu_tcap_thread_create(acu_tcap_thrd_id_t *id,
     ACU_TCAP_THREAD_FN((*fn), arg), void *fn_arg);
```

Purpose

This function creates a new thread to run the caller supplied function. The thread function can be defined portably as:

```
static ACU_TCAP_THREAD_FN(fn, arg)
{
    ...
    acu_tcap_thread_exit(1, 0);
    return 0;
}
```

Parameters

`id`	Data area to hold thread identification.
`fn`	Function to call in the new thread.
`fn_arg`	Argument to pass `fn`.

Return value

Zero on success, `-1` on failure.

2.4.3.2 acu_tcap_thread_exit

```
void acu_tcap_thread_exit(int detach, unsigned int rval);
```

Purpose

This function causes the current thread to terminate itself.

Parameters

`detach`	If non-zero the thread will exit and free all associated system resources. If zero `acu_tcap_thread_join()` must be called to free the resources.
`rval`	Return value to pass to the caller of `acu_tcap_thread_join()`.

If a thread function returns (instead of calling `acu_tcap_thread_exit`) then it is not detached and `acu_tcap_thread_join` must be called to free the operating system resources.

2.4.3.3 acu_tcap_thread_join

```
void acu_tcap_thread_join(acu_tcap_thrd_id_t *id, unsigned int *rval);
```

Purpose

This function waits for the specified thread to terminate, saves the thread return code, and frees all the system resources associated with the thread.

Parameters

`id`	Thread identification data for the thread (from `acu_tcap_thread_create`) .
`rval`	Pointer to a where the thread return code will be written.

2.4.3.4 acu_tcap_thread_id

```
int acu_tcap_thread_id(void);
```

Purpose

This function returns an operating system supplied identifier for the current thread.

Return value

The operating system identifier for the current thread. This has the same value as the `att_thrd_id` field of the `acu_tcap_thrd_id_t` structure.

2.4.4 Thread Pool functions

Thread pools allow an application to execute a function in a different thread without the overhead of creating a thread, and restricting the total number of threads by queuing the request until a thread becomes available.

There is no requirement to use these functions; an application can handle threads itself.

2.4.4.1 acu_tcap_thread_pool_create

```
acu_tcap_thrd_pool_t *acu_tcap_thread_pool_create(unsigned int min_threads,
    unsigned int max_threads, unsigned int max_queued_jobs);
```

Purpose

This function creates a thread pool.

Parameters

`min_threads`	Minimum (initial) number of threads to create.
`max_threads`	Maximum number of threads.
`max_queued_jobs`	Maximum permitted length of pending job list.

Return value

The address of the pool; or `NULL` if the pool cannot be created.

The `min_threads` parameter allows a thread pool to be populated with the indicated number of threads before `acu_tcap_thread_pool_create()` returns. If `min_threads` is zero, the thread pool is initially empty.

The `max_threads` parameter allows a thread pool to be constrained to no more than the indicated number of threads. If `max_threads` is large, the availability of system resources will also constrain thread pool growth.

The default behaviour of `acu_tcap_thread_pool_submit()` is to add the user-supplied job to a list of pending jobs if an idle thread is not available. The `max_queued_jobs` parameter limits the maximum length of this pending job list.

2.4.4.2 acu_tcap_thread_pool_destroy

```
int acu_tcap_thread_pool_destroy(acu_tcap_thrd_pool_t *pool);
```

Purpose

This function destroys a thread pool.

Parameters

`pool`	The pool address.

Return value

Zero if successful, `ACU_TCAP_ERROR_BAD_THREAD_POOL` on failure.

The call to `acu_tcap_thread_pool_destroy()` blocks until any active user-supplied jobs have returned.

2.4.4.3 acu_tcap_thread_pool_num_active

```
int acu_tcap_thread_pool_num_active(acu_tcap_thrd_pool_t *pool);
```

Purpose

This function returns the number of threads running user-supplied jobs.

Parameters

`pool`	The pool address.

Return value

Zero if successful, `ACU_TCAP_ERROR_BAD_THREAD_POOL` on failure.

2.4.4.4 acu_tcap_thread_pool_num_idle

```
int acu_tcap_thread_pool_num_idle(acu_tcap_thrd_pool_t *pool);
```

Purpose

This function returns the number of idle threads in the pool.

Parameters

`pool` The pool address.

Return value

Zero if successful, `ACU_TCAP_ERROR_BAD_THREAD_POOL` on failure.

2.4.4.5 acu_tcap_thread_pool_num_jobs

```
int acu_tcap_thread_pool_num_jobs(acu_tcap_thrd_pool_t *pool);
```

Purpose

This function returns the number of jobs on the pool's list of pending jobs.

Parameters

`pool` The pool address.

Return value

Zero if successful, `ACU_TCAP_ERROR_BAD_THREAD_POOL` on failure.

2.4.4.6 acu_tcap_thread_pool_submit

```
int acu_tcap_thread_pool_submit(acu_tcap_thrd_pool_t *pool,
     void (*fn)(void *), void *arg, unsigned int flags);
```

Purpose

This function submits a user-supplied job to a thread pool.

If no idle threads are available (and the number of threads in the pool is below the limit set during pool creation), some new threads will be added. If an attempt at pool growth fails to produce an idle thread capable of handling the user's job, the job will be added to the pool's list of pending jobs. The job list will be serviced in FIFO order as existing threads complete their current jobs.

The above default behaviour may be modified by inclusion of the flags described below.

Parameters

`pool`	The pool address.
`fn`	Function to be called by a thread from the thread pool.
`arg`	Argument to pass to `fn`.
`flags`	A bitwise OR of zero or more of:
`ACU_TCAP_THREAD_POOL_F_BLOCKING`	If no idle threads are available, the call to `acu_tcap_thread_pool_submit()` will block until a thread becomes available.
`ACU_TCAP_THREAD_POOL_F_NO_JOB`	When no idle threads are available (and any attempt at pool growth has failed), the function and argument specified will not be added to the pool's list of outstanding jobs.

Return value

Zero if successful or one of the following on failure:

`ACU_TCAP_ERROR_BAD_THREAD_POOL`
`ACU_TCAP_ERROR_THREAD_POOL_NO_FREE_THREADS`
`ACU_TCAP_ERROR_MALLOC_FAIL`

Appendix A: Building TCAP applications

A.1 Linux

The TCAP API header file includes all the necessary system headers.
Compile with `-D_REENTRANT`.
Link with `-lpthread -Wl,--enable-new-dtags`.
Link with `-Wl,-rpath,$ACULAB_ROOT/lib64` to get the location of the libraries embedded in the application image (`$ACULAB_ROOT` here must be expanded at program link time).

A.2 Windows

To obtain the correct definitions the symbol `_WINSOCKAPI_` must be defined before `windows.h` is included. One way to achieve this is to specify `-D_WINSOCKAPI_=` on the compiler command line.

Since the TCAP library itself creates threads, the program must be compiled as a threaded program. ie: build with `-MT` (or `-MTd`) not `-ML`.

The application must also include `windows.h` and `winsock2.h` before the TCAP API header file.

Appendix B: tcap_api.h

B.1 Error Codes

The error codes returned by the TCAP library functions are small negative integers. API functions may return any of the error codes below, not just those identified in the section for the API function itself.

In most cases more detailed information is written to the logfile.

`ACU_TCAP_ERROR_SUCCESS`
: Call succeeded (guaranteed to be zero).

`ACU_TCAP_ERROR_NO_COMPONENT`
: No more components in received message.

`ACU_TCAP_ERROR_TIMEDOUT`
: Request timed out.

`ACU_TCAP_ERROR_NO_MESSAGE`
: There are no messages on the specified queue.

`ACU_TCAP_ERROR_NO_INFORMATION_AVAILABLE`
: Requested information isn't available.

`ACU_TCAP_ERROR_MALLOC_FAIL`
: The library failed to allocate memory for a data item.
Check the application for memory leaks.

`ACU_TCAP_ERROR_NO_THREADS`
: The TCAP library failed to create a thread.
Check that the application isn't using more threads than the operating system can support.

`ACU_TCAP_ERROR_BAD_TRANSACTION`
: The `acu_tcap_trans_t` parameter doesn't reference a valid transaction data area.

`ACU_TCAP_ERROR_BAD_SSAP`
: The `acu_tcap_ssap_t` parameter doesn't reference a valid ssap data area.

`ACU_TCAP_ERROR_BAD_MESSAGE`
: The `acu_tcap_msg_t` parameter doesn't reference a valid message data area.

`ACU_TCAP_ERROR_BAD_EVENT`
: The `acu_tcap_event_t` parameter doesn't reference a valid event data area.

Note **The above four errors are likely to be caused by the application using a stale pointer.**

`ACU_TCAP_ERROR_BAD_OS_EVENT`
: An internal function tried to use an invalid operating system event or file descriptor.

`ACU_TCAP_ERROR_WRONG_MSG_TYPE`
: The message buffer isn't the correct type for the called function.

`ACU_TCAP_ERROR_ALREADY_CONNECTED`
: The ssap is already connected to (or is trying to connect to) SCCP.

`ACU_TCAP_ERROR_NO_LOCAL_SSN`
: No local SCCP sub-system number has been set.

`ACU_TCAP_ERROR_NO_LOCAL_POINTCODE`
: No local MTP3 pointcode has been set.

`ACU_TCAP_ERROR_UNKNOWN_CONFIG_PARAM`
: Configuration parameter not known.

`ACU_TCAP_ERROR_INVALID_CONFIG_VALUE`
: Invalid, or out of range, configuration parameter value.

`ACU_TCAP_ERROR_CANNOT_OPEN_CONFIG_FILE`
: The configuration file cannot be opened.

`ACU_TCAP_ERROR_BAD_TRANSACTION_STATE`
: The transaction isn't in the correct state for the request.

`ACU_TCAP_ERROR_INVOKE_ID_OUTSTATE`
: The operation state engine rejected the request because the specified invoke-id is not in the correct state.

`ACU_TCAP_ERROR_INVOKE_ID_IDLE`
: The operation state engine rejected the request because the specified invoke-id is

not in use.

`ACU_TCAP_ERROR_ASN1_ENCODING`
: An application supplied buffer isn't valid ASN.1, or an error from the ASN.1 encoding functions.

`ACU_TCAP_ERROR_INVALID_PROBLEM_TYPE`
: Requested reject problem doesn't refer to a valid problem type.

`ACU_TCAP_ERROR_INVALID_DIALOGUE_USERINFO`
: Supplied dialogue userinfo isn't a valid ASN.1 external data item.

`ACU_TCAP_ERROR_DIALOGUE_NOT_ALLOWED`
: The transaction state doesn't allow a dialogue portion to be added to the current message.

`ACU_TCAP_ERROR_NO_REMOTE_TRANS_ID`
: The message requires a remote transaction-id, but none is available. E.g.: trying to send a CONTINUE message on a newly created transaction.

`ACU_TCAP_ERROR_INVALID_MSG_TYPE`
: The specified message type is invalid for the TCAP variant.

`ACU_TCAP_ERROR_BUILD_SEQUENCE`
: The application is attempting to build a TCAP message in an order other than that of the final message. E.g.: Trying to define the dialogue portion after a component has been added.

`ACU_TCAP_ERROR_INVALID_DIALOGUE_USER_SYNTAX`
: The received dialogue portion of an abort message isn't an ASN.1 external data item, or is one of the TCAP pdus.

`ACU_TCAP_ERROR_BAD_ASN1`
: Received message is invalidly encoded ASN.1.

`ACU_TCAP_ERROR_BAD_PROTOCOL_VERSION`
: None of the proposed protocol versions are supported.

`ACU_TCAP_ERROR_BAD_TCAP_MSG`
: The received message isn't encoded according to the protocol rules.

`ACU_TCAP_ERROR_BAD_LOCAL_TID`
: The received message doesn't contain a valid local transaction identifier. All local transaction identifiers are 4 bytes long.

`ACU_TCAP_ERROR_BAD_REMOTE_TID`
: The received message either doesn't contain a remote transaction identifier, contains one when it should not, or contains a different one from that already saved for the local transaction.

`ACU_TCAP_ERROR_UNKNOWN_MESSAGE_TYPE`
: The received message isn't a valid TCAP message.

`ACU_TCAP_ERROR_THREAD_POOL_NO_FREE_THREADS`
: No free threads, or thread pool job queue full.

`ACU_TCAP_ERROR_BAD_THREAD_POOL`
: The `acu_tcap_thrd_pool_t` parameter doesn't reference a valid thread pool data area.

Note **Do not confuse the above error codes with call control error codes that have the same numeric values.**

B.2 SCCP addresses

The `acu_sccp_addr_t` structure has the following fields:

`sa_flags`	bitwise 'or' of the following:	
	`ACU_SCCP_SA_FLAGS_ROUTE_SSN`	route on SSN (even if global title present)
	`ACU_SCCP_SA_FLAGS_RAW_GT`	raw global title (unknown `sa_gti`)
`sa_valid`	indicates which address elements contain valid data, bitwise 'or of:	
	`ACU_SCCP_SA_VALID_GTI`	
	`ACU_SCCP_SA_VALID_SSN`	
	`ACU_SCCP_SA_VALID_PC`	
	`ACU_SCCP_SA_VALID_RL_PC`	
	`ACU_SCCP_SA_VALID_TT`	
	`ACU_SCCP_SA_VALID_NP`	
	`ACU_SCCP_SA_VALID_ES`	
	`ACU_SCCP_SA_VALID_NAI`	
`sa_gti`	global title indicator (4 bits)	
`sa_ssn`	subsystem number	
`sa_pc`	SS7 signalling pointcode for/from SCCP address buffer	
`sa_rl_pc`	SS7 signalling pointcode from MTP3 routing label	
`sa_tt`	translation type (8 bits)	
`sa_np`	numbering plan (4 bits)	
`sa_es`	encoding scheme (4 bits)	
`sa_nai`	nature of address indicator (7 bits)	
`sa_gt.sag_num`	number of digits in `sa_gt.sag_digits`	
`sa_gt.sag_digits[]`	global title address information, two digits per byte	

The global title indicator placed in outbound messages depends on which of the `ss_tt`, `sa_np`, `sa_es` and `sa_nai` fields are marked as valid, not on the value of `sa_gti`.

The `sa_gt.sag_num` field contains the number of digits (not bytes) in the global title. The application need not care about the odd/even field of the encoded global title.

The `sa_rl_pc` field contains the pointcode from the MTP3 routing label of received messages, it has no effect on outward messages.

When routing using global titles, if the sa_pc field is set then the SCCP driver will not perform global title translation and will send the message to that point code, if the sa_pc is not set then global title translation is performed.

The SCCP protocol constrains the valid combinations of TT, NP, ES and NAI. NP and ES must always be specified together. NAI is not valid for ANSI SCCP, and, for ITU and China SCCP, must be specified on its own or with TT, NP and ES.

The first digit of the global title is encoded in the least significant 4 bits of `sa_gt.sag_digits[0]` and the second digit in the most significant 4 bits. This matches the protocol encoding, but is reversed from a normal hexdump of the address buffer.

Appendix C: System limits

The following limits are inherent in the design of the TCAP product; however other constraints (e.g. lack of memory) may apply first:

Dimension	Limit	Notes
Connections to an SCCP system	4094	Also constrained by available server-side resources.
Transactions per library ssap	983040	Costs a few kb per transaction.
Operations per transaction	256	No additional cost per operation.
Active timers per ssap	None	Timers are held in binary heap. Start and stop are O(log active_timers).

Appendix D: ASN.1 BER encoding

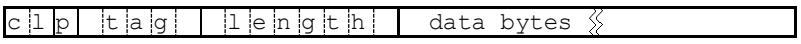
This section contains a brief description of the ASN.1 BER (Basic Encoding Scheme) used for TCAP and associated protocols. For a complete definition refer to X.690 (formerly X.209).

BER encodes data in a byte aligned 'type', 'length', 'value' (TLV) manner, X.690 uses the term 'identifier' for the type and 'contents' for the value. The 'length' is always exclusive – i.e.: the 'identifier' and 'length' bytes are excluded from the specified length

The 'identifier' field can specify that the 'contents' is further BER encoded data – i.e.: is a 'constructed' item. The 'identifier' and 'length' fields are usually a single byte, but the encoding allows larger values to be described using multiple bytes. Constructed items can be marked as having an 'indefinite length', in which case a terminating data item is used.

D.1 Basic encoding rules

Data items with small tags and short lengths are encoded as:



Where:

`cl`	Class of tag: 00 Universal, data type and encoding is defined by X.680 and X.690. 01 Application wide [APPLICATION n]. 10 Context specific [n]. 11 Private use [PRIVATE n] (Used by ANSI TCAP). The encoding rules do not depend on the class.
`p`	0 if primitive, 1 if constructed.
`tag`	Identifier for data (zero to 30).
`length`	Number of data bytes (zero to 127).

Tags larger than 30 are encoded by setting the `tag` bits of the initial byte to 31 and following it with extra bytes. Each additional byte contains 7 bits of the tag value (most significant bits in the first extra byte), all but the last extra byte having its most significant bit set. So a context-specific primitive field with tag of 31 is encoded as `0x9f 0x1f`, tag 127 as `0x9f 0x7f`, and tag 128 as `0x9f 0x81 0x00` etc.

Lengths larger than 127 are encoded by setting the most significant bit of the length byte to a 1 and putting the number of bytes required to encode the length into the lower 7 bits. The length itself then follows with 8 bits per byte and the most significant byte first. So a length of 128 is encoded as `0x81 0x80`, 255 as `0x81 0xff`, and 256 as `0x82 0x01 0x00` etc.

Constructed items can be encoded with an 'indefinite length'. This is useful if the overall length isn't known when tag for the item is written. This is done by specifying `0x80` for the length byte and using two zero bytes to terminate the constructed item. The TCAP library ASN.1 encoding functions normally use the indefinite length form for all constructed items that exceed 127 bytes. The indefinite form uses one extra byte for lengths 128 to 255.

Note **TCAP requires that definite length fields be encoded in their shortest form.**

A data item might be defined as:

```
result [0] INTEGER,
```

Which is encoded 'context specific' 'constructed' 0, followed by the 'universal' coding for an integer. Giving `0xa0 0x03 0x02 0x01 0x2a` when result is 42 (or `0xa0 0x80 0x02 0x01 0x2a 0x00 0x00` if the indefinite length form is used).

More usually it would be defined as:

```
result [0] IMPLICIT INTEGER,
```

Which is encoded 'context specific' 'primitive' 0, followed by the data of the integer coding. i.e.: `0x80 0x01 0x2a`.

D.2 Universal tags

The common universal tags are:

0		Reserved for encoding indefinite length terminators.
1	Boolean	One byte, 0 => false, other values => true.
2	Integer	2s compliment signed integer most significant byte first, encoded in the minimum number of bytes.
3	Bitstring	First byte indicates the number of unused bits in the last byte [0..7]. Latter bytes contain bits with the 0x80 bit used first.
4	Octetstring	Any sequence of bytes.
5	Null	Length always zero, no data bytes.
6	Object Identifier	See below.
8	External data	See below.
10	Enumerated	Encoded as an integer.
16	Sequence	A constructed item where the data items have to appear in the specified order.
17	Set	A constructed item where the data items can appear in any order and may be repeated.

X.690 allows octetstring and bitstring to be constructed (i.e.: made up of several concatenated parts). However the TCAP specification requires that they be primitive.

D.2.1 Object Identifiers

Object identifiers are globally assigned hierarchic identifiers used for data structures and protocols. In ASN.1 definitions they are typically specified as (for example):

```
{ itu-t recommendations q 773 modules (2) messages (1) version2 (2) }
```

Each field is converted to a number using the rules defined in Annex B of X.680. The latter identifiers have the numeric value in parenthesis. The first identifier is one of `itu-t` (0), `iso` (1) or `joint-iso-itu-t` (2). `itu-t` used to be `ccitt`.

If the first identifier is `itu-t`, the second is one of `recommendation` (0), `question` (1), `administration` (2), `network-operator` (3), or `identified-organization` (4). For recommendations the next identifier is based on the letter (with `a` => 1 and `z` => 26) and the following one the number of the relevant document. Corporate bodies may have an object tree subsidiary to the identified-organization identifier.

If the first identifier is `iso`, the second is one of `standard` (0), `member-body` (2) or `identified-organization` (3).

The above example is thus the series of decimal numeric values 0, 0, 17, 773, 2, 1, 2. The first two are converted to a single byte by multiplying the first by 40 and adding in the second. Values greater than 127 are converted to multiple bytes by putting 7 bits into each byte and setting the most significant bit of all but the last byte. So the data bytes that encode the above example are `0x00 0x11 0x86 0x05 0x02 0x01 0x02`.

D.2.2 External data

External data items are used to refer to ASN.1 definitions in other documents.
X.690 section 8.18 defines the external data type as:

```
[UNIVERSAL 8] IMPLICIT SEQUENCE {
     direct-reference          OBJECT IDENTIFIER OPTIONAL,
     indirect-reference        INTEGER OPTIONAL,
     data-value-descriptor     ObjectDescriptor OPTIONAL,
     encoding                  CHOICE {
        single-ASN1-type            [0] ABSTRACT-SYNTAX.&Type,
        octet-aligned               [1] IMPLICIT OCTET STRING,
        arbitrary                   [2] IMPLICIT BIT STRING } }
```

In ITU TCAP they usually contain a direct reference to an object identifier that refers to a standards document, and use the single-ASN.1-type encoding to encapsulate the relevant data.

Appendix E: C Pre-processor explained

The header files for the TCAP API make use of some little-used features of the C pre-processor. This has been done in order to avoid error-prone replication of information, and to ensure that sets of data, that would normally have to be defined separately, are always kept in step. A simple example is defining the explanatory texts for error codes in the same place as the error code itself. The message definitions of section 2.2.4 make heavy use of this.

Consider what happens when `#define foo(x) x(args)` is expanded: `foo(bar)` clearly becomes `bar(args)`. If we also have `#define bar(args)` then `bar()` is expanded AFTER `foo()` allowing us to generate any text including `args`. So we have passed the name of one `#define` as a parameter to a different `#define`.

If we replace the definition of `foo` with `#define foo(x) x(args1) x(args2)` then `foo(bar)` is equivalent to `bar(args1) bar(args2)`. This is useful because `foo(baz)` expands to `baz(args1) baz(args2)` allowing us to feed the same set of arguments to more than one `#define`.

A simple example:

```
#define lights(x) x(red) x(orange) x(green)
#define xx(colour) LIGHT_##colour,
enum { lights(xx) NUM_LIGHTS };
#undef xx
#define xx(colour) #colour,
static const char light_names[] = { lights(xx) };
#undef xx
```

This expands to:

```
enum { LIGHT_red, LIGHT_orange, LIGHT_green, NUM_LIGHTS };
static const char light_names[] = { "red", "orange", "green", };
```

(We needed to add `NUM_LIGHTS` because a trailing comma isn't valid in a C++ enum.)

Remember that `#` causes the parameter to be converted to a string and that `##` causes parameters to be concatenated.

In this case the `enum` definition would probably just follow the definition of `lights` (in the header file), whereas the `light_names` array definition would more likely be in some C source file associated with error messages and/or tracing.

The advantage of this is that, if we ever have to add another colour, adding it to the `#define` automatically updates both definitions.

Compile time errors in these expansions can be difficult to locate, not helped by the very long lines the expansions generate. Processing the pre-processor output through `sed` to break the long lines can help somewhat, a suitable command is in the released `tcap_asn1_codec.h` file.

Contact us

Phone

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

Email

Info@aculab.com

Sales@aculab.com

Support@aculab.com

Socials

Certificate number IS 722024
ISO 27001:2013

Certificate number FS722030
ISO 9001:2015