Aculab V6 switch API guide

Revision 6.7.0

PROPRIETARY INFORMATION

Document Revision

Rev	Date	By	Detail
6.0	06.12.02	DJL	Interim release
6.1.0	04.06.03	DJL	Evaluation trials
6.1.0	15.01.04	DJL	Initial Controlled V6 release
6.2.0	14.07.04	DJL	Review updates
6.2.2	07.09.04	DJL	Beta release
6.2.2	15.09.04	DJL	Full release
6.2.3	02.11.04	DJL	Clarification added regarding IP streams and timeslots
6.3.0	08.11.04	DJL	Update for new 16 port cPCI card
6.3.1	25.01.05	DJL	Additional calls to support new 16 port cPCI features
6.3.2	09.02.05	DJL	Notes added for specific 16 port E1/T1 cPCI functions
6.3.3	11.04.05	DJL	`sw_set_output()` update for 16 port card
6.3.4	19.04.05	DJL	Updated to reflect latest release
6.4.0	07.10.05	DJL	Updates for V6.4 release
6.4.1	03.03.06	DJL	Correction to example scripts
6.4.2	07.07.06	DJL	Updates following header file review
6.4.3	16.10.06	DJL	Addition of stream information for Prosody X cPCI
6.4.4	14.12.06	DJL	Addition of new companding APIs and further `ERR_SW_NO_PATH` guidance.
6.4.5	17.08.07	PP	Addition of Prosody X PCIe card and `sw_abort_api_calls()`
6.4.6	19.05.09	PP	Note about Prosody X NETREF in section 3.25. References to "Stratum 4 Enhanced" compatible changeover in section 5.1.1. Corrections to notes column in table in A.3 and other minor corrections.
6.5.0	01.02.11	DF/ EBJ/ PP	Removed references to EOL products. Updated to corporate fonts and removal of hyperlinks. Description of failsafe mode.
6.5.1	08.02.11	PP	Addition of Prosody X rev 3 cards.

Rev	Date	By	Detail
6.5.2	01.11.11	PP	Addition of Prosody X 1U. Stream numbers and clock modes for this product added to Appendix A.
6.6.0	04.12.15	PP	Change “Prosody X 1U” to “Prosody X 1U Enterprise” and add brief reference to Prosody X HA. Remove references to PX PCI cards, PX cPCI cards, PX PCIe rev1 cards, H.110 bus, `CT_NETREF_2` and hot-swap.
6.7.0	21.09.16	PP	`sw_reset_switch()` - reference to companding added.

CONTENTS

1 Introduction

This document describes the generic switching and clock control API presented by the Aculab driver to application programs, and is applicable for use with the following Aculab products:

- Prosody X PCIe rev 3 card
- Prosody X 1U Enterprise
- Prosody X 1U High Availability

The same API is supported across diverse operating systems.

The Prosody X 1U High Availability chassis contains either one or two Prosody X PCIe rev 3 cards. The switching and clocking available on these cards is the same as on standard Prosody X PCIe rev 3 cards.

1.1 Terminology

In this document, the generic switching and clock control API presented by the Aculab switch driver is referred to as the switch API. Reference is also made to `idle_net_ts()` and `port_init()` routines in the Aculab Call Control library. These routines are not formally part of the switch API but are intimately related to it.

Aculab cards usually have one or more network ports to allow the cards to be connected to a telecom network, referred to in this document as the network.

A *stream* refers to a time division multiplexed (TDM) signal consisting of a number of timeslots. Each timeslot carries a 64Kbps-speech path (also known as a B channel or a bearer channel), which might carry, for example, speech data relating to a single phone call.

The term *expansion bus* is used to refer to the time division multiplexed telecom buses into which Aculab cards can be integrated. Aculab cards currently support the following expansion bus:

- H.100 bus (also known as CT Bus or H-Bus) – PCIe rev 3 cards only

Prosody X 1U Enterprise does not have an expansion bus.

Each Aculab card is equipped with a digital switch matrix that allows data to be switched from a timeslot in one stream to another timeslot in the same stream or another stream.

Streams from card network ports, on-card resources such as DSP modules and the expansion buses, are connected to the digital switch matrix.

Each card is also equipped with a clock generation circuit used to synchronize switching of data between timeslots on streams, and between the card and the network. The clock generation circuit is driven from a clock reference source that could be the local oscillator on the card, a network port or an expansion bus.

1.2 Switching

Aculab cards can switch data between timeslots on network ports, on-card resources, and associated expansion buses, for example, H.100 (H-Bus).

Each card type supported by the switch driver has a different set of streams connected to its digital switch matrix according to the number of network ports it has, the expansion bus types supported, and the on-card resources fitted. Each stream is assigned a logical stream number. For details of the stream numbers used on each type of card, please refer to Appendix A.

The switch driver API may be used to control the digital switch matrix in order to allow the switching of data between timeslots on different streams, this is termed “making a connection”. For example, a timeslot from a network port stream could be switched to a timeslot on an H-Bus stream. The digital switch matrix may also be set up to make multiple connections from a single timeslot to multiple destination timeslots simultaneously.

The switch driver API may be used to set up the digital switch matrix to output constant 8-bit patterns on timeslots, and sample 8-bit values from timeslots.

When switching data between network ports and/or resources located on the same card, a single connection may be made between the source of the speech data and its required destination. This is termed local switching. Alternatively, two connections can be made, the first in order to switch data from the source up to a free timeslot on a card expansion bus, and the second to switch the data down from the expansion bus to its required destination. This is termed “distributed switching.”

1.3 Clock control

Each Aculab card is equipped with a clock generation circuit, which is used to synchronize the switching of timeslot data between the network, on-card resources such as DSPs, and the card’s expansion buses.

The clock generation circuit may be set up to obtain clock timing information from a number of sources including:

- A signal at a network port
- A local oscillator
- An expansion bus primary or secondary clock signal

If data is to be switched between an Aculab card and the network, it is essential that the clock generation circuit is synchronized to network timing, otherwise “slip” errors may occur. Exceptionally, rather than derive timing information from the network, a card may be required to provide timing to the network (from its local oscillator), for example, this would be the case if a card is running the network end of a signalling protocol in a back to back test configuration.

A card may be synchronized to the network clock by setting the clock generation circuit to derive timing from one of the network ports on the card. The card could alternatively synchronize to network timing from an expansion bus whose clock master is another card using a network port as reference.

If data is to be switched between two Aculab cards via an expansion bus, it is essential that their clock generation circuits be synchronized to the expansion bus clock. This is achieved either through acting as the bus clock master or through ‘clock slaving’ off the bus.

The switch driver API may be used to control the reference source for the clock generation circuit and control whether the card acts as clock master or clock slave on an expansion bus.

1.4 Expansion buses

1.4.1 H.100 bus

The H.100 bus, sometimes referred to as the CT Bus, is a time division multiplexed bus consisting of 32 streams. Each stream carries 128 unidirectional 64Kbps-speech paths (timeslots).

The 32 streams are named D0, D1, ..., D31, and the timeslots within these streams are identified using numbers 0 to 127.

Aculab cards attached to the H.100 bus may switch data onto the H.100 bus from network port timeslots and on-card resources, or vice versa.

It is the responsibility of the application to manage the use of the H.100 bus and to assign H.100 bus timeslots as required. Bus timeslots are used to carry speech data between cards, or for making distributed switch connections. H.100 bus timeslots are not used when making local switch connections.

No more than one card at a time may output data onto a particular H.100 bus stream timeslot. For example, if a card is outputting data to timeslot 6 of stream D0, then no other card can be permitted to output onto steam D0 timeslot 6. If this were to occur in error, then it would be referred to as bus contention.

Multiple cards may of course switch data from the same H.100 bus stream timeslot.

In order for data to be switched successfully between cards on the H.100 bus, all the cards on the bus must be synchronized to the H.100 bus clocks. One card must be set up to be H.100 primary clock master, a second card may optionally be set up to be a secondary clock master, all remaining cards must be set up to be H.100 bus clock slaves. See section 5 for more information on H.100 bus clocking.

The first and last cards on an H.100 bus ribbon cable should have H.100 bus terminations enabled. Bus terminations may be configured using the Aculab Configuration Tool (ACT), or by manually editing a card's configuration file (see section 5.2.1), or by using the API call `sw_set_card_h100_termination()`.

1.4.2 Monitoring two party conversations - DSP/Prosody required

In order for an expansion bus to carry speech data from a two party conversation, two expansion bus timeslots are required, one each for the signal transmitted to each peer party.

It is not possible to combine the two signals (e.g. for monitoring by a 3rd party) merely by switching both signals to a single expansion bus timeslot. Attempting to do this from a single card would fail, as the first connection made would be replaced by the second. Attempting to do this by switching from two or more cards to the same expansion bus timeslot would result in bus contention.

If an application needs to output the combined input speech path and the output speech path of a phone call onto a single timeslot, the two signals must be merged using an algorithm running on an Aculab DSP or Prosody module.

2 API types, header files and libraries

In order that applications using the switch driver API may be compiled with diverse compilers on different operating systems, the use of some basic C data types such as `int` and `long` are avoided in parameter block structure definitions for the API calls. Instead, a portable basic integer type is defined in the header "`acu_type.h`" and this type is used in the switch driver parameter block definitions (i.e. `ACU_INT` is used instead of `int`). When applications are compiled, it is essential that the size of parameter block structures is the same as the size that the driver is expecting. This is enforced by the switch library, which checks the `size` field in the parameter block structure. The `INIT_ACU_STRUCT` macro can be used to initialise the `size` field. It will also set all the remaining fields in the parameter block structure to zero.

All applications built to use Aculab drivers must include the header file:

```
acu_type.h
```

Applications using the switch driver must also include the following header file:

```
sw_lib.h
```

If the application includes call control processing, then the appropriate call control libraries should also be linked into the application.

For some operating systems or compilers, the header files may need to be edited. Various other compile time pre-processor definitions and compiler options may be required.

2.1 Operating systems

2.1.1 Windows

Applications that use the Switch API must be linked against the following DLL:

```
sw_lib.dll
```

2.1.2 Linux

Applications that use the Switch API must be linked against the following shared library:

```
libacu_sw.so
```

2.2 Opening cards for use with the switch driver

Before an application can make Switch API calls to a card, the card must first be opened for use. To open a card for general use, the `acu_open_card()` function is used. This function requires a card serial number to identify the card to open. Card serial numbers can be obtained from the `acu_get_system_snapshot()` API call. Once a serial number is known, it can be used as in the following example:

```
ACU_CHAR* serial_no = "1234567";
ACU_OPEN_CARD_PARMS open_card_parms;
ACU_ERR result;

INIT_ACU_STRUCT(&open_card_parms);

strncpy(open_card_parms.serial_no, serial_no, ACU_MAX_SERIAL);

result = acu_open_card(&open_card_parms);

if (result != 0)
{
  printf("Failed opening card %s with error %d\n", serial_no, result);
}
```

`acu_open_card()` returns a unique identifier for the card that can be used in a number of API calls. The same card id can be used with the Switch, Call, and Prosody APIs. In order to use the Switch API with the newly opened card, a further API call is used - `acu_open_switch()`. This function takes the card ID returned by `acu_open_card()` and opens the switch driver for that card. After this, the card ID can be used, as the `card_id` parameter in all Switch API calls. The `acu_open_switch()` function is used as in the following example:

```
ACU_CARD_ID card_id; /* a previously opened card */
ACU_OPEN_SWITCH_PARMS open_switch_parms;
ACU_ERR result;

INIT_ACU_STRUCT(&open_switch_parms);

open_switch_parms.card_id = card_id;

result = acu_open_switch(&open_switch_parms);

if (result != 0)
{
  printf("Error %d opening switch API\n", result);
  exit(EXIT_FAILURE);
}
```

When the application has finished using the Switch API with a card, it must close the switch driver for that card. This is performed using the `acu_close_switch()` function. As this function closes the switch driver, subsequent Switch API calls using the same card ID will fail. This function does not however close the card ID for the purposes of any other Aculab APIs. The following sample code demonstrates the use of `acu_close_switch()`:

```
ACU_CARD_ID card_id; /* id of a previously opened card */
ACU_CLOSE_SWITCH_PARMS close_switch_parms;
ACU_ERR result;

INIT_ACU_STRUCT(&close_switch_parms);

close_switch_parms.card_id = card_id;

result = acu_close_switch(&close_switch_parms);
```

3 API call summary and descriptions

The calls that together make up the switch driver API are listed in the table below:

Switch API call	Description
`sw_ver_switch()`	Determines the version of a given switch driver.
`sw_component_version()`	Determines the version of a given component.
`sw_mode_switch()`	Determines which expansion buses a card may switch data onto/off-from.
`sw_card_info()`	Returns card information for the card whose switching is controlled by the switch driver `card_id`.
`sw_set_card_notification_queue()`	Used to register a queue that will receive switch notification events (such as clock fallback events) for the specified card.
`sw_get_card_notfication()`	The switch driver queues events such as clock fallbacks. This function is used to collect those events.
`sw_get_card_notfication_wait_object()`	This function retrieves an operating system specific wait event that can be used to wait for switch driver events.
`sw_config_timeslot_companding()`	A-law/mu-law to mu-law/A-law configuration, used to select A-law/mu-law to mu-law/A-law conversion on network timeslot streams.
`sw_query_timeslot_companding()`	Query A-law/mu-law to mu-law/A-law conversion, determines the companding conversion settings of a given card timeslot.
`sw_config_companding()`	A-law/mu-law to mu-law/A-law configuration, used to select A-law/mu-law to mu-law/A-law conversion on network port streams.
`sw_query_companding()`	Query A-law/mu-law to mu-law/A-law conversion, determines the companding conversion settings of a given card.
`sw_get_dsp_stream_info()`	Maps DSP position and port indexes to TDM streams and timeslots.
`sw_set_card_h100_termination()`	Enables or disables the H-Bus bus termination.
`sw_set_output()`	Make a connection; break a connection or output pattern on timeslot.

Switch API call	Description
`sw_query_output()`	Determine the source of switch connection.
`sw_sample_input()` `sw_sample_input0()`	Obtain 8-bit sample from given timeslot.
`sw_reset_switch()`	Reset card switch devices to disable all current connections.
`sw_tristate_switch()`	Tri-state card from expansion bus.
`sw_clock_control()`	Change clock generation circuit reference source and/or expansion bus clock master/slave mode.
`sw_query_clock_control()`	Determine last clock mode set for card.
`sw_h100_config_board_clock()`	Change clock generation circuit reference source and/or H-Bus bus clock master/slave mode.
`sw_h100_config_netref_clock()`	Configure fallback reference clock for H-Bus bus.
`sw_h100_query_board_clock()`	Determine last H-Bus clock mode set for card and status of H-Bus bus clocks.
`sw_h100_query_netref_clock()`	Determine fallback reference clock for H-Bus bus.
`sw_track_api_calls()`	This function may be used to track API calls made to a switch driver.
`sw_abort_api_calls()`	Abort pending API calls.

Routines that are implemented in the call library and referenced in this document are:

Call API routine	Description
`port_init()`	Used to write an idle pattern onto all timeslots on a port. For CAS protocols will also connect each timeslot to the signalling DSP.
`idle_net_ts()`	Output signalling specific signal to exchange when network port timeslot is idle.

The individual switch driver API calls are now specified in more detail:

Most calls take a card ID, which is returned by the `acu_open_card()` function. Before the switch API can be used with a card, the Switch driver must be opened using the `acu_open_switch()` function.

Miscellaneous functions

3.1 sw_ver_switch() - Get switch driver version

This function will return version information for the switch driver indicated by *card_id*.

Synopsis

```
int sw_ver_switch( ACU_CARD_ID card_id, struct swver_parms* vparms);

   typedef struct swver_parms
   {
     ACU_INT          size;          /* IN */
     ACU_INT          major;         /* OUT */
     ACU_INT          minor;         /* OUT */
     ACU_INT          step;          /* OUT */
     ACU_INT          custom;        /* OUT */
     ACU_INT          quality;       /* OUT */
     ACU_INT          buildno0;      /* OUT */
     ACU_INT          buildno1;      /* OUT */
   } SWVER_PARMS;
```

The `sw_ver_switch()` function takes a pointer *vparms* , to a structure `SWVER_PARMS`. The structure must be initialised before invoking the function (see section 2).

Input parameters

card_id

The required switch driver, which must be a valid card ID as returned by `acu_open_card()`

size

The `size` field should be set to the size of the input structure in bytes. This may be achieved using `INIT_ACU_STRUCT`

Return values

major, ***minor*** and ***step***

On return, the parameters `major`, `minor` and `step` will be set to the values X, Y, and Z respectively of the three-element version number for the driver X.Y.Z. For example, 6.2.11

custom

The `custom` parameter will be set to a non-zero value if the driver build is a custom special build of the driver for a specific customer.

quality

The `quality` parameter will be set to one of the following character values:

Character	Release quality
'I'	Released version
'F'	Field trial version
'S'	Customer special version
'B'	Beta quality version
'D'	Development (or alpha quality) version

buildno0 and ***buildno1***

The `buildno0` and `buildno1` parameters are for Aculab use only.

On successful completion a value of zero is returned; otherwise one of the following negative values will be returned indicating the type of error:

`ERR_SW_INVALID_SWITCH`	no switch driver corresponding to `card_id`
`ERR_SW_DEVICE_ERROR`	no switch drivers installed, switch driver initialisation failed or device `I/O` error occurred
`ERR_SW_COMPONENT_MISMATCH`	version number mismatch between software components.

3.2 sw_mode_switch() - Get switch driver mode

This function will return expansion bus capability information for the switch driver indicated by *card_id*.

Synopsis

```
int sw_mode_switch ( ACU_CARD_ID card_id, struct swmode_parms* mparms);

   typedef struct swmode_parms
   {
     ACU_INT          size;          /* IN */
     ACU_INT          ct_buses;      /* OUT */
   } SWMODE_PARMS;
```

The `sw_mode_switch()` function takes a pointer *mparms* , to a structure `SWMODE_PARMS`. The structure must be initialised before invoking the function (see section 2).

Input parameters

card_id

The required switch driver, which must be a valid card ID as returned by `acu_open_card()`

size

The `size` field should be set to the size of the input structure in bytes. This may be achieved using `INIT_ACU_STRUCT`

Return values

ct_buses

On return, the parameter `ct_buses` will be set to a value, which has bits set according to which expansion buses are supported by the driver for the card corresponding to indicated switch driver. The following bits are defined:

Bit	Expansion bus
`SWMODE_CTBUS_H100`	H.100 Bus
`SWMODE_CTBUS_H100_TERM`	H.100 Bus with terminated bus.
`SWMODE_CTBUS_NONE`	No CT bus available

On successful completion a value of zero is returned; otherwise one of the following negative values will be returned indicating the type of error:

`ERR_SW_INVALID_SWITCH` no switch driver corresponding to `card_id`

`ERR_SW_DEVICE_ERROR` no switch drivers installed, switch driver initialisation failed or device `I/O` error occurred

3.3 sw_card_info() - Retrieve card details

This function will return card information for the *card_id* whose switching is controlled by the switch driver.

Synopsis

```
int sw_card_info ( ACU_CARD_ID card_id, struct swcard_info_parms* iparms);

   typedef struct swcard_info_parms
   {
     ACU_INT          size;                          /* IN */
     ACU_INT          card_type;                     /* OUT */
     ACU_INT          card_present;                  /* OUT */
     ACU_INT          max_capacity;                  /* OUT */
     ACU_INT          additional_data[4];            /* OUT */
     ACU_ULONG        physical_address;              /* OUT */
     ACU_ULONG        io_address_or_pcidev;          /* OUT */
     ACU_ULONG        physical_irq;                  /* OUT */
     char             serial_no[kSWMaxSerialNoText]; /* OUT */
   } SWCARD_INFO_PARMS;
```

The `sw_card_info()` function takes a pointer *iparms* , to a structure `SWCARD_INFO_PARMS`. The structure must be initialised before invoking the function (see section 2).

Input parameters

card_id

The required switch driver, which must be a valid card ID as returned by `acu_open_card()`

size

The `size` field should be set to the size of the input structure in bytes. This may be achieved using `INIT_ACU_STRUCT`

Return values

card_type

This parameter indicates the type of card:

Card type value	Description
SW_PROSODY_X_PCIE_R3_CARD	Prosody X PCIe rev 3 card
SW_PROSODY_X_1U_CARD	Prosody X 1U Enterprise

card_present

This parameter is set to 1 for all cards.

max_capacity

This parameter indicates the maximum full duplex switching capacity of the card from on-card resources to the expansion bus in its configured mode.

additional_data

This parameter is reserved for future use.

physical_address , ***io_address_or_pcidev*** , ***physical_irq***

These parameters are obsolete.

serial_no

This parameter is a zero terminated ASCII string indicating the card serial number.

On successful completion a value of zero is returned; otherwise one of the following negative values will be returned indicating the type of error:

`ERR_SW_INVALID_SWITCH`	no switch driver corresponding to `card_id`.
`ERR_SW_DEVICE_ERROR`	no switch drivers installed, switch driver initialisation failed or device `I/O` error occurred.

3.4 sw_set_card_notification_queue()

This function is used to register a queue that will receive switch notification events, for example, clock fallback events, for the specified card.

Synopsis

```
ACU_ERR sw_set_card_notification_queue(ACU_QUEUE_PARMS* queue_parms);

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

The `sw_set_card_notification_queue()` function takes a pointer *queue_parms* , to a structure `ACU_QUEUE_PARMS`. The structure must be initialised before invoking the function (see section 2).

The `ACU_QUEUE_PARMS` structure is defined in header file `acu_type.h`.

Input parameters

size

The `size` field should be set to the size of the input structure in bytes. This may be achieved using `INIT_ACU_STRUCT`

resource_id

This field must be set to the card ID of the card.

queue_id

Set this field to the ID of the queue that is to be used for this card's switch notifications. This ID must have previously been allocated using `acu_allocate_event_queue()`.

Return values

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

3.5 sw_get_card_notification()

The switch driver queues events such as clock fallbacks. This function is used to collect such events.

Synopsis

```
ACU_INT sw_get_card_notification(ACU_CARD_ID card_id, struct
                                 sw_card_notification_parms* notifyp);

   typedef struct sw_card_notification_parms
   {
     ACU_ULONG        size;          /* IN */
     ACU_INT          event;         /* OUT */
   } SW_CARD_NOTIFICATION_PARMS;
```

The `sw_get_card_notification()` function takes a pointer *`notifyp`*, to a structure `SW_CARD_NOTIFICATION_PARMS`. The structure must be initialised before invoking the function (see section 2).

Input parameters

`card_id`

Is the ID of the card to check for events, which must be a valid card ID as returned by `acu_open_card()`.

`size`

The `size` field should be set to the size of the input structure in bytes. This may be achieved using `INIT_ACU_STRUCT`

Return values

`event`

Will be one of the following values:

#define	Description
`SW_EV_NO_EVENT`	No event
`SW_EV_PRIMARY_REF_NETWORK_PORT`	Primary network port change of state. Reported by an H-Bus primary clock master only
`SW_EV_CTBUS_PRIMARY_LOS`	H-Bus primary clock lost. Reported only by an H-Bus clock slave with fallback enabled.
`SW_EV_SECONDARY_REF_NETWORK_PORT`	Change in state on a network port driving `CT_NETREF1`

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

Example usage

See the example shown in `sw_get_notification_wait_object()` below.

3.6 sw_get_notification_wait_object()

This function retrieves an operating system specific wait event that can be used to wait for switch driver events.

Synopsis

```
ACU_INT sw_get_notification_wait_object(ACU_CARD_ID card_id, SW_WAIT_OBJECT_PARMS*
                                        wo_parms);

  typedef struct sw_wait_object_parms
  {
    ACU_ULONG             size;              /* IN */
    ACU_WAIT_OBJECT       wait_object;       /* OUT */
  } SW_WAIT_OBJECT_PARMS;
```

The `sw_get_notification_wait_object()` function takes a pointer *`wo_parms`*, to a structure `SW_WAIT_OBJECT_PARMS`. The structure must be initialised before invoking the function (see section 2).

Input parameters

`card_id`

Is the ID of the card the event will apply to, which must be a valid card ID as returned by `acu_open_card()`

`size`

The `size` field should be set to the size of the input structure in bytes. This may be achieved using `INIT_ACU_STRUCT`

Return values

`wait_object`

Will contain a platform specific wait event that can be used with functions such as `poll()` or `WaitForMultipleObjects()`.

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

Example usage

```
ACU_CARD_ID card_id; /* the id of a card obtained previously */
SW_CARD_NOTIFICATION_PARMS event_parms;
SW_WAIT_OBJECT_PARMS wo_parms;
ACU_ERR error;

INIT_ACU_STRUCT(&wo_parms);
INIT_ACU_STRUCT(&event_parms);

error = sw_get_notification_wait_object(card_id, &wo_parms);

if (error != 0)
{
  printf("Failed getting wait event with error %d\n", error);
  exit(-1);
}

if (WaitForSingleObject(wo_parms.wait_object, INFINITE) == WAIT_OBJECT_0)
{
  error = sw_get_card_notification(card_id, &event_parms);
}
```

3.7 sw_set_card_h100_termination() - Enable/disable H-Bus termination

Enables or disables the H-Bus termination on Prosody X PCIe rev 3 cards.

Synopsis

```
int sw_set_card_h100_info(ACU_CARD_ID card_id, int h100termination);
```

Input parameters

card_id

Selects the required switch driver and must be a valid card id returned by `acu_open_card()`.

h100termination

Enables or disables the H-Bus termination for the specified card ID.

h100termination	
Value	Description
0	H-Bus termination disabled (default)
1	H-Bus termination enabled

Return values

On successful completion a value of zero is returned; otherwise one of the following negative values will be returned indicating the type of error:

`ERR_SW_INVALID_SWITCH`	No switch driver corresponding to `card_id`.
`ERR_SW_DEVICE_ERROR`	An error was returned from a device driver called by this driver.

3.8 sw_config_ timeslot_companding() - A-law/mu-law to mu-law/A-law configuration

This function is used to select A-law to mu-law or mu-law to A-law conversion on individual timeslots on network port streams. By default companding conversion is disabled.

Synopsis

```
int sw_config_timeslot_companding(ACU_CARD_ID card_id,
                                  timeslot_companding_parms* compandp);

   typedef struct timeslot_companding_parms
   {
     ACU_INT      size;          /* IN */
     ACU_INT      stream;        /* IN */
     ACU_INT      timeslot;      /* IN */
     ACU_INT      rx_mode;       /* IN */
     ACU_INT      tx_mode;       /* IN */
   } TIMESLOT_COMPANDING_PARMS;
```

The `sw_config_timeslot_companding()` function takes a pointer *compandp*, to a structure `TIMESLOT_COMPANDING_PARMS`. The structure must be initialised before invoking the function (see section 2).

Input parameters

card_id

The required switch driver, which must be a valid card ID as returned by `acu_open_card()`

size

The `size` field should be set to the size of the input structure in bytes. This may be achieved using `INIT_ACU_STRUCT`

stream

Set to the value for the required stream, this can be any stream number between 32 and 47.

timeslot

Set to the value of the required timeslot (0-31).

tx_mode and ***rx_mode***

`rx_mode` selects the conversion mode of a network port receive stream.
`tx_mode` selects the conversion mode of a network port transmit stream.
`tx_mode` and `rx_mode` should each be set to one of the following values:

```
COMPANDING_DISABLED         0
COMPANDING_A_TO_MU_LAW      2
COMPANDING_MU_TO_A_LAW      3
```

In addition, on original Prosody X cards, `rx_mode` may be set to one of the following values:

```
COMPANDING_T1_IDLE          6
COMPANDING_E1_IDLE          7
COMPANDING_E1_SILENCE       8
COMPANDING_T1_SILENCE       9
```

Selecting one of these will force a fixed pattern to appear on a PMX receive stream timeslot.

This provides a method for switching idle or silence patterns onto H-Bus timeslots, which may be used to work around a limitation of the switch device (see “Switching restrictions for original (pre-rev 3) Prosody X cards” for further details).

Return values

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

3.9 sw_query_ timeslot_companding() - Query A-law/mu-law to mu-law/A-law conversion

This function is used to determine the companding conversion settings of a given port.

Synopsis

```
int sw_query_timeslot_companding(ACU_CARD_ID card_id, timeslot_companding_parms*
                                 compandp);

   typedef struct timeslot_companding_parms
   {
     ACU_INT      size;          /* IN */
     ACU_INT      stream;        /* IN */
     ACU_INT      timeslot;      /* IN */
     ACU_INT      rx_mode;       /* OUT */
     ACU_INT      tx_mode;       /* OUT */
   } TIMESLOT_COMPANDING_PARMS;
```

The `sw_query_timeslot_companding()` function takes a pointer *compandp*, to a structure `TIMESLOT_COMPANDING_PARMS`. The structure must be initialised before invoking the function (see section 2).

Input parameters

card_id

The required switch driver, which must be a valid card ID as returned by `acu_open_card()`

size

The `size` field should be set to the size of the input structure in bytes. This may be achieved using `INIT_ACU_STRUCT`

stream

Set to the value for the required stream, this can be any stream number between 32 and 47.

timeslot

Set to the value of the required timeslot (0-31).

Return values

tx_mode and ***rx_mode***

On return, these parameters will be set to values indicating the conversion modes operating on the requested network port. These may be one of the following values:

```
COMPANDING_DISABLED         0
COMPANDING_A_TO_MU_LAW      2
COMPANDING_MU_TO_A_LAW      3
```

In addition, on original Prosody X cards, `rx_mode` may be set to one of the following values:

```
COMPANDING_T1_IDLE          6
COMPANDING_E1_IDLE          7
COMPANDING_E1_SILENCE       8
COMPANDING_T1_SILENCE       9
```

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

3.10 sw_config_companding() - A-law/mu-law to mu-law/A-law configuration

This function is used to select A-law to mu-law or mu-law to A-law conversion on network port streams. By default the companding conversion is disabled.

Synopsis

```
int sw_config_companding(ACU_CARD_ID card_id, companding_parms* compandp);

   typedef struct companding_parms
   {
     ACU_INT      size;          /* IN */
     ACU_INT      stream;        /* IN */
     ACU_INT      rx_mode;       /* IN */
     ACU_INT      tx_mode;       /* IN */
   } COMPANDING_PARMS;
```

The `sw_config_companding()` function takes a pointer *compandp*, to a structure `COMPANDING_PARMS`. The structure must be initialised before invoking the function (see section 2).

Input parameters

card_id

The required switch driver, which must be a valid card ID as returned by `acu_open_card()`

size

The `size` field should be set to the size of the input structure in bytes. This may be achieved using `INIT_ACU_STRUCT`

stream

Set to the value for the required stream, this can be any stream number between 32 and 47.

tx_mode and ***rx_mode***

`rx_mode` selects the conversion mode of a network port receive stream.
`tx_mode` selects the conversion mode of a network port transmit stream.
`tx_mode` and `rx_mode` should be set to one of the following values:

```
COMPANDING_DISABLED         0
COMPANDING_A_TO_MU_LAW      2
COMPANDING_MU_TO_A_LAW      3
```

Return values

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

3.11 sw_query_companding() - Query A-law/mu-law to mu-law/A-law conversion

This function is used to determine the companding conversion settings of a given port.

Synopsis

```
int sw_query_companding(ACU_CARD_ID card_id, companding_parms* compandp);

   typedef struct companding_parms
   {
     ACU_INT      size;          /* IN */
     ACU_INT      stream;        /* IN */
     ACU_INT      rx_mode;       /* OUT */
     ACU_INT      tx_mode;       /* OUT */
   } COMPANDING_PARMS;
```

The `sw_query_companding()` function takes a pointer *compandp*, to a structure `COMPANDING_PARMS`. The structure must be initialised before invoking the function (see section 2).

Input parameters

card_id

The required switch driver, which must be a valid card ID as returned by `acu_open_card()`

size

The `size` field should be set to the size of the input structure in bytes. This may be achieved using `INIT_ACU_STRUCT`

stream

Set to the value for the required stream, this can be any stream number between 32 and 47.

Return values

tx_mode and ***rx_mode***

On return, these parameters will be set to values indicating the conversion modes operating on the requested network port. These may be one of the following values:

```
COMPANDING_DISABLED     0
COMPANDING_A_TO_MU_LAW  2
COMPANDING_MU_TO_A_LAW  3
```

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

3.12 sw_component_version() – Get switch driver component version

This function will return version information for the switch driver component as indicated by *component_id*.

Synopsis

```
int sw_component_version(ACU_CARD_ID card_id, struct component_version_parms*
                         vparms);

   typedef struct component_version_parms
   {
     ACU_INT      size;              /* IN */
     ACU_INT      component_id;      /* IN */
     ACU_INT      major;             /* OUT */
     ACU_INT      minor;             /* OUT */
     ACU_INT      step;              /* OUT */
     ACU_INT      custom;            /* OUT */
     ACU_INT      quality;           /* OUT */
     ACU_INT      reserved0;         /* OUT */
     ACU_INT      reserved1;         /* OUT */
   } COMPONENT_VERSION_PARMS;
```

`sw_component_version()` takes a pointer *vparms*, to a structure `COMPONENT_VERSION_PARMS`. The structure must be initialised before invoking the function (see section 2).

Input parameters

card_id

The required switch driver, which must be a valid card ID as returned by `acu_open_card()`

size

The `size` field should be set to the size of the input structure in bytes. This may be achieved using `INIT_ACU_STRUCT`

component_id

The switch driver component version information to be returned:

0 – will return the version of either the t8110.ko or tdmsw.ko component

1 – will return the version of the PXSCS component

Return values

major, ***minor*** and ***step***

On return, the parameters *major*, *minor* and *step* will be set to the values X, Y, and Z respectively of the three-element version number for the driver X.Y.Z. For example, 6.2.11

custom

The *custom* parameter will be set to a non-zero value if the driver build is a custom special build of the driver for a specific customer.

quality

The *quality* parameter will be set to one of the following character values:

Character	Release standard
'I'	Released version
'F'	Field trial version
'S'	Customer special version
'B'	Beta quality version
'D'	Development (or alpha quality) version

reserved0 and ***reserved1***

The reserved0 and reserved1 parameters are for Aculab use only.

On successful completion a value of zero is returned; otherwise, one of the following negative values will be returned indicating the type of error:

ERR_SW_INVALID_SWITCH	no switch driver corresponding to card_id
ERR_SW_DEVICE_ERROR	no switch drivers installed, switch driver initialisation failed or device I/O error occurred
ERR_SW_COMPONENT_MISMATCH	version number mismatch between software components.

3.13 sw_get_dsp_stream_info()

This function returns the corresponding TDM stream number and stream length for a given DSP type, position, and serial port.

Synopsis

```
int sw_get_dsp_stream_info(ACU_CARD_ID card_id, DSP_STREAM_INFO_PARMS*infoparms);

   typedef struct info_parms
   {
     ACU_INT      size;                  /* IN */
     ACU_INT      dsp_type;              /* IN */
     ACU_INT      dsp_position_ix;       /* IN */
     ACU_INT      dsp_serial_port_ix;    /* IN */
     ACU_INT      switching_stream;      /* OUT */
     ACU_INT      no_ts_in_stream;       /* OUT */
   } DSP_STREAM_INFO_PARMS;
```

The function `sw_get_dsp_stream_info()` takes a pointer, *`infoparms`*, to a structure `DSP_STREAM_INFO_PARMS`. The structure must be initialised before invoking the function (see section 2).

Input parameters

`card_id`

Selects the required switch driver, which must be a valid card id as returned by `acu_open_card()`.

`size`

The `size` field should be set to the size of the input structure in bytes. This may be achieved using `INIT_ACU_STRUCT`

`dsp_type`

The following DSP type designations are defined:

DSP type designation	
Constant	Description
`kSW_DSP_TYPE_SIGNALLING`	DSP used for CAS or SS7 signalling such as PMX 8101/3
`kSW_DSP_TYPE_PROSODY`	DSP used for Prosody such as 8122

`dsp_position_ix`

Ranges from zero to `kSW_MAX_DSP_POSITION_IX` and enumerates the physical position of the DSP of a given type on the baseboard, or on modules attached to the baseboard.

`dsp_serial_port_ix`

Indicates the serial port for which switch stream information is required.

Return values

`switching_stream`

The switch stream number corresponding to the selected DSP serial port index.

`no_ts_in_stream`

The number of timeslots for the port.

On successful completion a value of zero is returned; otherwise one of the following negative values will be returned indicating the type of error:

`ERR_SW_NO_SUCH_DSP`	DSP of the given type is not fitted to the indicated position
`ERR_SW_NO_SUCH_DSP_PORT`	indicated serial port does not exist on DSP
`ERR_SW_INVALID_PARAMETER`	invalid `dsp_type` specified

3.14 sw_abort_api_calls()

This function forces the switch library to terminate pending switch API calls.

Synopsis

```
int sw_abort_api_calls(ACU_CARD_ID card_id);
```

Input parameters

card_id

Selects the required switch driver, which must be a valid card id as returned by `acu_open_card()`.

Description

This function should be used under the following conditions. If a resource manager `ACU_SYS_EVT_CARD_REMOVED` event occurs (indicating, for example, failure of the network connection to a Prosody X card), applications that have previously invoked `acu_open_switch()` should call `sw_abort_api_calls()`. This will cause all pending switch API calls on the removed card to complete with the return code `ERR_SW_API_CALL_ABORTED`.

Switching Functions

3.15 sw_set_output () - Control switch matrix

This function is used to make and break connections between streams and timeslots.

Synopsis

```
int sw_set_output( ACU_CARD_ID card_id, stuct output_parms* oparms);

   typedef struct output_parms
   {
     ACU_INT      size;     /* IN */
     ACU_INT      ost;      /* IN */
     ACU_INT      ots;      /* IN */
     ACU_INT      mode;     /* IN */
     ACU_INT      ist;      /* IN */
     ACU_INT      its;      /* IN */
     ACU_INT      pattern;  /* IN */
   } OUTPUT_PARMS;
```

The `sw_set_output()` function takes a pointer *oparms* , to a structure `OUTPUT_PARMS`. The structure must be initialised before invoking the function (see section 2).

Description

The Aculab switch driver provides local switching (where source and sink are not connected via the H-bus but must be on the same card) and distributed switching (where source and sink are connected via the H-bus and may either be on the same card or on different cards). Both local switching and distributed switching are available on Aculab Prosody X PCIe rev 3 cards.

Input parameters

card_id

Is a valid card ID as returned by `acu_open_card()`

size

The `size` field should be set to the size of the input structure in bytes. This may be achieved using `INIT_ACU_STRUCT`

The behaviour of the following stream, timeslot and pattern parameters are subject to the selected `mode` option:

ost	output stream number
ots	output timeslot number
ist	input stream number
its	input timeslot number
pattern	the value to be written to the output timeslot
mode	the mode of operation of the output timeslot

The function has three modes of operation:

`CONNECT_MODE`

In this mode the input timeslot in the input stream is connected to the output timeslot on the output stream. The output stream and timeslot are provided by `ost` and `ots` respectively, input stream and timeslot are provided by `ist` and `its` respectively, `pattern` is not used and may assume any value.

`PATTERN_MODE`

In this mode the pattern provided is written in the output timeslot of the output stream. The

output stream and timeslot are provided by `ost` and `ots` respectively and `pattern` contains the value to be written to the time slot. The input stream and timeslot `ist` and `its` are not used and may assume any value.

`DISABLE_MODE`

In this mode the output timeslot on the output stream is tri-stated. The output stream and timeslot are provided by `ost` and `ots` respectively. The input stream and timeslot and pattern - `ist`, `its` and `pattern` - are not used and may assume any value.

Return values

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

`ERR_SW_NO_PATH` indicates a problem was encountered making the requested connection.

3.16 sw_query_output() - Query switch matrix

This function returns the current mode and connection of a given output stream and timeslot as configured by the `sw_set_output()` function.

Synopsis

```
int sw_query_output( ACU_CARD_ID card_id, stuct output_parms* queryp);

   typedef struct output_parms
   {
     ACU_INT      size;     /* IN */
     ACU_INT      ost;      /* IN */
     ACU_INT      ots;      /* IN */
     ACU_INT      mode;     /* OUT */
     ACU_INT      ist;      /* OUT */
     ACU_INT      its;      /* OUT */
     ACU_INT      pattern;  /* OUT */
   } OUTPUT_PARMS;
```

The `sw_query_output()` function takes a pointer *queryp*, to a structure `OUTPUT_PARMS`. The structure must be initialised before invoking the function (see section 2).

Input parameters

card_id

The required switch driver, which must be a valid card ID as returned by `acu_open_card()`

size

The `size` field should be set to the size of the input structure in bytes. This may be achieved using `INIT_ACU_STRUCT`

ost and ***ots***

The input parameters `ost` and `ots` define the output stream and timeslot respectively, on which the query is to take place.

Return values

mode, ***ist***, ***its*** and ***pattern***

`mode`, `ist`, `its` and `pattern` will be set up to indicate the source of any data being switched to the specified output timeslot.

On successful completion a value of zero is returned; otherwise one of the following negative values will be returned indicating the type of error:

`ERR_SW_INVALID_SWITCH`	no switch driver corresponding to `card_id`
`ERR_SW_INVALID_STREAM`	invalid `ost` stream number specified
`ERR_SW_INVALID_TIMESLOT`	invalid `ots` timeslot number specified
`ERR_SW_DEVICE_ERROR`	no switch drivers installed, switch driver initialisation failed or device `I/O` error occurred

3.17 sw_sample_input(), sw_sample_input0() - Sample timeslot

This function samples an octet from the indicated timeslot. The behaviour of the legacy API call `sw_sample_input0()` is identical to `sw_sample_input()`. See Appendix B for further information.

Synopsis

```
int sw_sample_input( ACU_CARD_ID card_id, struct sample_parms* samplep);

int sw_sample_input0( ACU_CARD_ID card_id, struct sample_parms* samplep);

   typedef struct sample_parms
   {
     ACU_INT      size;     /* IN */
     ACU_INT      ist;      /* IN */
     ACU_INT      its;      /* IN */
     char         sample;   /* OUT */
   } SAMPLE_PARMS;
```

The `sw_sample_input()` function takes a pointer *samplep* , to a structure `SAMPLE_PARMS`. The structure must be initialised before invoking the function (see section 2).

Input parameters

card_id

The required switch driver, which must be a valid card ID as returned by `acu_open_card()`

size

The *size* field should be set to the size of the input structure in bytes. This may be achieved using `INIT_ACU_STRUCT`

ist and ***its***

The input parameters `ist` and `its` define the input stream and timeslot respectively on which the sample is to be taken.

Return values

sample

Contains an 8-bit sample of the data that is currently asserted on the input timeslot.

On successful completion a value of zero is returned; otherwise one of the following negative values will be returned indicating the type of error:

`ERR_SW_INVALID_SWITCH`	no switch driver corresponding to `card_id`
`ERR_SW_INVALID_STREAM`	invalid `ist` stream number specified
`ERR_SW_INVALID_TIMESLOT`	invalid its timeslot number specified
`ERR_SW_NO_RESOURCES`	no resources to sample this input
`ERR_SW_PATH_BLOCKED`	cannot sample this input without breaking a current connection
`ERR_SW_DEVICE_ERROR`	no switch drivers installed, switch driver initialisation failed or device `I/O` error occurred

3.18 sw_tristate_switch() - Tristate switch matrix

This function is used to enable or disable the whole switch matrix.

Synopsis

```
int sw_tristate_switch( ACU_CARD_ID card_id, int tristate);
```

Input parameters

card_id

The required switch driver, which must be a valid card ID as returned by `acu_open_card()`

tristate

Enables or disables the switch matrix:

`tristate = 0` : switch matrix enabled

`tristate = 1` : switch matrix disabled (tristated)

Return values

On successful completion a value of zero is returned; otherwise one of the following negative values will be returned indicating the type of error:

`ERR_SW_INVALID_SWITCH`	no switch driver corresponding to `card_id`
`ERR_SW_DEVICE_ERROR`	no switch drivers installed, switch driver initialisation failed or device `I/O` error occurred

3.19 sw_reset_switch() - Reset switch matrix

This function will reset the switch matrix to the idle state. All timeslot outputs on all streams will be disabled on each expansion bus. All companding conversion settings will be reset to `COMPANDING_DISABLED`. The state of the clock will not be altered by this function.

Synopsis

```
int sw_reset_switch( ACU_CARD_ID card_id);
```

Input parameters

card_id

The required switch driver, which must be a valid card ID as returned by `acu_open_card()`

Return values

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

`ERR_SW_INVALID_SWITCH`	no switch driver corresponding to `card_id`
`ERR_SW_DEVICE_ERROR`	no switch drivers installed, switch driver initialisation failed or device `I/O` error occurred

Clock Control Functions

3.20 sw_clock_control() - Set clock reference

This function controls the configuration of the reference source for the clock generation circuit, and the card clock master/slave mode on expansion buses.

Synopsis

```
int sw_clock_control( ACU_CARD_ID card_id, int clockmode );
```

Description

When a switch driver is configured to operate a card in H-Bus mode, it may be more appropriate to use the H-Bus specific clock configuration API call `sw_h100_config_board_clock()`. This gives the application the ability to set up the full range of possible H-Bus clocking scenarios rather than the simple master/slave type scenario configurable through this call.

Input parameters

card_id

The required switch driver, which must be a valid card ID as returned by `acu_open_card()`

clockmode

This argument provides a designator for the clock. Not all clock modes are valid for all card types or card operation modes. See Appendix A for permitted values for the various card types. See section 5 for further information on clocking.

Return values

On successful completion a value of zero is returned; otherwise one of the following negative values will be returned indicating the type of error:

`ERR_SW_INVALID_SWITCH`	no switch driver corresponding to `card_id`
`ERR_SW_INVALID_CLOCK_PARM`	clock mode invalid for card or card mode
`ERR_SW_DEVICE_ERROR`	no switch drivers installed, switch driver initialisation failed or device `I/O` error occurred

3.21 sw_query_clock_control() - Query clock reference

This function determines the last configured clock set up for a card.

When a switch driver is configured to operate a card in H-Bus mode, it may be more appropriate to use the H-Bus specific clock query API call `sw_h100_query_board_clock()`. This gives the application more information about how the H-Bus bus clocking is currently set up.

Synopsis

```
int sw_query_clock_control( ACU_CARD_ID card_id, struct query_clkmode_parms*
                            queryp);

   typedef struct query_clkmode_parms
   {
     ACU_INT      size;                    /* IN */
     ACU_INT      last_clock_mode;         /* OUT */
     ACU_INT      sysinit_clock_mode;      /* OUT */
   } QUERY_CLKMODE_PARMS;
```

The `sw_query_clock_control()` function takes a pointer *queryp*, to a structure `QUERY_CLKMODE_PARMS`. The structure must be initialised before invoking the function (see section 2).

Input parameters

card_id

The required switch driver, which must be a valid card ID as returned by `acu_open_card()`

size

The `size` field should be set to the size of the input structure in bytes. This may be achieved using `INIT_ACU_STRUCT`

Return values

last_clock_mode

Will be set to the last clock mode that the clock generation circuit was set up with.

sysinit_clock_mode

The value returned by this field is no longer used and should be ignored

On successful completion a value of zero is returned; otherwise one of the following negative values will be returned indicating the type of error:

`ERR_SW_INVALID_SWITCH`	no switch driver corresponding to `card_id`
`ERR_SW_DEVICE_ERROR`	no switch drivers installed, switch driver initialisation failed or device `I/O` error occurred

3.22 sw_h100_config_board_clock() - Set up H-Bus clocking

Used to set up card clocking parameters, See section 5 for more information on H-Bus clocking issues.

Synopsis

```
int sw_h100_config_board_clock(ACU_CARD_ID card_id, struct
                                h100_config_board_clock_parms* clockp);

  typedef struct h100_config_board_clock_parms
  {
    ACU_INT     size;                   /* IN */
    ACU_INT     clock_source;           /* IN */
    ACU_INT     network;                /* IN */
    ACU_INT     h100_clock_mode;        /* IN */
    ACU_INT     auto_fall_back;         /* IN */
    ACU_INT     netref_clock_speed;     /* IN */
  } H100_CONFIG_BOARD_CLOCK_PARMS;
```

Input parameters

The `sw_h100_config_board_clock()` function takes a pointer *`clockp`*, to a structure `H100_CONFIG_BOARD_CLOCK_PARMS`. The structure must be initialised before invoking the function (see section 2).

`card_id`

The required switch driver, which must be a valid card ID as returned by `acu_open_card()`

`size`

The `size` field should be set to the size of the input structure in bytes. This may be achieved using `INIT_ACU_STRUCT`

`clock_source`

This parameter indicates reference source for the card clock generation circuit and should be set to one of the following values:

Clock source	Value
`H100_SOURCE_INTERNAL`	1
`H100_SOURCE_NETWORK`	3
`H100_SOURCE_H100_A`	8
`H100_SOURCE_H100_B`	9
`H100_SOURCE_NETREF`	10

`network`

If `clock_source` is set to `H100_SOURCE_NETWORK`, then the parameter `network` should be set to a value that indicates which network port should be used as the reference source. `network` may also need to be configured if `clock_source` is set to `H100_SOURCE_H100_A` or `H100_SOURCE_H100_B`, and auto fall back is enabled, see below.

`h100_clock_mode`

Determines whether the card is an H-Bus primary or secondary clock master, or an H-Bus clock slave, and should be set to one of the following values:

H.100 clock mode	Value
H100_SLAVE	0
H100_MASTER_A	1
H100_MASTER_B	2

auto_fall_back

Determines the card's clocking behaviour when an H-Bus clock failure occurs. It should be set to H100_FALLBACK_DISABLED if no clock fallback action is required. If a clock fallback action is required, it should be set to H100_FALLBACK_ENABLED with one or more of the following optional qualifiers added to it:

Auto fallback qualifier	Value
H100_FALLBACK_DISABLED	0
H100_FALLBACK_ENABLED	1
H100_AUTO_RETURN	16
H100_CHANGEOVER_TO_NETWORK	32
H100_CHANGEOVER_TO_NETREF	64

The interpretation of this parameter depends on the cards current clock mode as follows:

Card configured as H-Bus primary clock master:

If the card is operating as a H-Bus bus primary clock master with the h100_clock_mode parameter set to either H100_MASTER_A or H100_MASTER_B, and the clock_source parameter set to H100_SOURCE_NETWORK, then setting the parameter auto_fall_back to H100_FALLBACK_ENABLED will cause the H-Bus CT_NETREF signal to automatically be used as a fallback clock reference.

If auto_fall_back is qualified by H100_AUTO_RETURN, then when the original reference source once more becomes available, the card will revert to using it as clock reference.

If auto_fall_back is set to H100_FALLBACK_DISABLED, a default configuration is used. This does not guarantee valid H-Bus primary clocks if the primary master network reference is lost. It does however ensure that the H-Bus primary clocks will be restored when the primary master network reference recovers.

Card configured as H-Bus secondary clock master:

If the card is operating as a H-Bus bus secondary clock master with the h100_clock_mode parameter set to either H100_MASTER_A or H100_MASTER_B, and the clock_source parameter set to H100_SOURCE_H100_A or H100_SOURCE_H100_B, then setting the parameter auto_fall_back to H100_FALLBACK_ENABLED will cause the card to automatically be promoted to become the new primary clock master driving the alternate set of H-Bus clock signals if the primary clock master fails.

If auto_fall_back is set to H100_FALLBACK_DISABLED, no automatic promotion will occur.

The qualifier H100_CHANGEOVER_TO_NETWORK indicates that a network port (specified by the network parameter) is to be used as reference source by a promoted clock master, the qualifier H100_CHANGEOVER_TO_NETREF indicates that the reference source CT_NETREF is to be used instead. Specifying both qualifiers indicates the network port is to be used and that fallback to CT_NETREF is to be enabled should the network port reference source subsequently fail.

Card configured as H-Bus clock slave:

If the card is operating as a H-Bus clock slave with the `h100_clock_mode` parameter set to `H100_SLAVE`, then setting the parameter `auto_fall_back` to `H100_FALLBACK_ENABLED` will cause the card to automatically fallback to the alternate clocks driven by the secondary clock master if the primary clock master fails. If `auto_fall_back` is set to `H100_FALLBACK_DISABLED`, no automatic fallback will occur.

netref_clock_speed

The parameter `netref_clock_speed` indicates the clock rate that the H.100 `CT_NETREF` fallback clock line is running at, and should be set to one of the following values:

NETREF clock speed	Value
`H100_NETREF_8KHZ` (8 kHz)	0
`H100_NETREF_1544MHZ` (1.544 MHz)	1
`H100_NETREF_2048MHZ` (2.048 MHz)	2

Return values

On successful completion a value of zero is returned; otherwise one of the following negative values will be returned indicating the type of error:

`ERR_SW_DEVICE_ERROR`	no switch drivers installed, switch driver initialisation failed or device `I/O` error occurred.
`ERR_SW_INVALID_SWITCH`	no switch driver corresponding to `card_id`.

3.23 sw_h100_config_netref_clock() - Set up H-Bus fallback clock

Used to set up fallback clocking parameters, see section 5 for more information on H-Bus bus clocking issues.

Synopsis

```
int sw_h100_config_netref_clock(ACU_CARD_ID card_id, struct
                                h100_netref_clock_parms* fallbackp);

   typedef struct h100_netref_clock_parms
   {
     ACU_INT     size;                  /* IN */
     ACU_INT     network;               /* IN */
     ACU_INT     netref_clock_mode;     /* IN */
     ACU_INT     netref_clock_speed;    /* IN */
   } H100_NETREF_CLOCK_PARMS;
```

The `sw_h100_config_netref_clock()` function takes a pointer *`fallbackp`* , to a structure `H100_NETREF_CLOCK_PARMS`. The structure must be initialised before invoking the function (see section 2).

Input parameters

`card_id`

The required switch driver, which must be a valid card ID as returned by `acu_open_card()`

`size`

The `size` field should be set to the size of the input structure in bytes. This may be achieved using `INIT_ACU_STRUCT`

`network`

If `netref_clock_mode` is set to `H100_ENABLE_NETREF`, then the parameter *`network`* should be set to a value that indicates which network port should be used as the reference source for the `CT_NETREF` clock signal.

`netref_clock_mode`

NETREF clock mode	Value
`H100_DISABLE_NETREF`	0
`H100_ENABLE_NETREF`	1

`netref_clock_speed`

If the parameter `netref_clock_mode` is set to `H100_ENABLE_NETREF`, then the H-Bus `CT_NETREF` clock signal will be driven at the rate indicated by the parameter `netref_clock_speed`, which must take one of the following values:

NETREF clock speed	Value
`H100_NETREF_8KHZ` (8 kHz)	0
`H100_NETREF_1544MHZ` (1.544 MHz)	1
`H100_NETREF_2048MHZ` (2.048 MHz)	2

The reference clock source for the generated `CT_NETREF` will be the network port indicated by the `network` parameter.

The type of firmware (E1 or T1) loaded on the network port selected as the `CT_NETREF`

reference clock source determines the possible speeds that CT_NETREF may be driven at. If the port indicated by the network parameter is loaded with E1 firmware, then CT_NETREF may be driven at either 8kHz or 2.048MHz; if the port indicated by the network parameter is loaded with T1 firmware, then CT_NETREF may be driven at either 8kHz or 1.544MHz. Firmware must be loaded on to the network port to be used as the CT_NETREF source before sw_h100_config_netref_clock() is invoked.

If the card is already acting as an H-Bus primary clock master, the fallback network port selected in this call must be distinct from the network port selected as the primary master clock reference network port.

Return values

On successful completion a value of zero is returned; otherwise one of the following negative values will be returned indicating the type of error:

ERR_SW_INVALID_SWITCH	no switch driver corresponding to card_id
ERR_SW_DEVICE_ERROR	no switch drivers installed, switch driver initialisation failed or device I/O error occurred

3.24 sw_h100_query_board_clock() - Query H-Bus clock mode

Function is used to determine H-Bus clock settings for a given card, and status of H-Bus clocks, see section 5 for more information on H-Bus clocking issues.

Synopsis

```
int sw_h100_query_board_clock (ACU_CARD_ID card_id, struct
                               h100_query_board_clock_parms* queryp);

  typedef struct h100_query_board_clock_parms
  {
    ACU_INT     size;                   /* IN */
    ACU_INT     clock_source;           /* OUT */
    ACU_INT     network;                /* OUT */
    ACU_INT     h100_clock_mode;        /* OUT */
    ACU_INT     auto_fall_back;         /* OUT */
    ACU_INT     fall_back_occurred;     /* OUT */
    ACU_INT     h100_a_clock_status;    /* OUT */
    ACU_INT     h100_b_clock_status;    /* OUT */
    ACU_INT     netref_a_clock_status;  /* OUT */
    ACU_INT     netref_b_clock_status;  /* OUT */
  } H100_QUERY_BOARD_CLOCK_PARMS;
```

The `sw_h100_query_board_clock()` function takes a pointer *`queryp`* , to a structure `H100_QUERY_BOARD_CLOCK_PARMS`. The structure must be initialised before invoking the function (see section 2).

Input parameters

`card_id`

The required switch driver, which must be a valid card ID as returned by `acu_open_card()`

`size`

The `size` field should be set to the size of the input structure in bytes. This may be achieved using `INIT_ACU_STRUCT`

Return values

`clock_source`**, *`network`*****, *`h100_clock_mode`*****,** and ***`auto_fall_back`***

On return the output parameters, `clock_source`, `network`, `h100_clock_mode`, and `auto_fall_back` will normally be set to the last values set up by a call to `sw_h100_config_board_clock()`. If clock settings have changed, clock fallback may have occurred

`h100_a_clock_status`**, *`h100_b_clock_status`*** and ***`netref_a_clock_status`***

The output parameters `h100_a_clock_status`, `h100_b_clock_status` and `netref_a_clock_status` will be set to one of the following values:

Clock status	Value
`H100_CLOCK_STATUS_GOOD`	0
`H100_CLOCK_STATUS_BAD`	1
`H100_CLOCK_STATUS_UNKNOWN`	2

Prosody X PCIe rev 3 cards can determine `CT_NETREF` status when the `CT_NETREF` clock frequency is 8kHz, 1.544MHz, or 2.048MHz provided that the `CT_NETREF` clock rate for the H-bus has been previously set using either `sw_h100_config_netref_clock()` or `sw_h100_config_board_clock()`.

fallback_occurred

The `fallback_occurred` parameter will be set to a non-zero value if a primary master has fallen back and is using `CT_NETREF` as a clock source.

On successful completion a value of zero is returned; otherwise one of the following negative values will be returned indicating the type of error:

`ERR_SW_INVALID_SWITCH`	no switch driver corresponding to `card_id`
`ERR_SW_DEVICE_ERROR`	no switch drivers installed, switch driver initialisation failed or device `I/O` error occurred

3.25 sw_h100_query_netref_clock() - Query H-Bus fallback clock

Synopsis

```
int sw_h100_query_netref_clock(ACU_CARD_ID card_id, struct
                                h100_netref_clock_parms* queryp);

   typedef struct h100_netref_clock_parms
   {
     ACU_INT     size;                  /* IN */
     ACU_INT     network;               /* OUT */
     ACU_INT     netref_clock_mode;     /* OUT */
     ACU_INT     netref_clock_speed;    /* OUT */
   } H100_NETREF_CLOCK_PARMS;
```

The `sw_h100_query_netref_clock()` function takes a pointer *queryp*, to a structure `H100_NETREF_CLOCK_PARMS`. The structure must be initialised before invoking the function (see section 2).

Input parameters

card_id

The required switch driver, which must be a valid card ID as returned by `acu_open_card()`

size

The `size` field should be set to the size of the input structure in bytes. This may be achieved using `INIT_ACU_STRUCT`

Return values

network, netref_clock_mode and ***netref_clock_speed***

On return the output parameters `network`, `netref_clock_mode` and `netref_clock_speed` will reflect the current card set up with respect to H-Bus `CT_NETREF` generation, values for these output parameters are as described for the `sw_h100_config_netref_clock()` API call.

After a card has been configured to generate the H.100 `CT_NETREF` signal from one of its ports, the H.100 `CT_NETREF` clock signal will only be generated when a good signal is present at the port. At other times, when no signal is present at the port, the `CT_NETREF` signal will not be generated. The returned `netref_clock_mode` allows an application to determine if `CT_NETREF` is currently being generated. If a card has previously been configured to generate `CT_NETREF` from a network port and `CT_NETREF` is not currently being generated (due to the absence of a valid signal at the port) then `netref_clock_mode` will be set to `H100_DISABLE_NETREF`.

On successful completion a value of zero is returned; otherwise one of the following negative values will be returned indicating the type of error:

`ERR_SW_INVALID_SWITCH`	no switch driver corresponding to `card_id`
`ERR_SW_DEVICE_ERROR`	no switch drivers installed, switch driver initialisation failed or device `I/O` error occurred

Diagnostic and trace functions

3.26 sw_track_api_calls() - Track API calls

For applications running on multi-tasking operating systems, this function may be used to track API calls made to a switch driver.

Typically this call would be used in a diagnostic application such as when `swcmd` is running as a separate process to the application whose switch API calls are being tracked.

An alternative to using this API call is to use the Aculab `trace_mode` tool to generate a log of switch API calls.

Synopsis

```
int sw_track_api_calls (SW_TRACK_API_PARMS track_api );


   typedef struct sw_track_api_parms
   {
     ACU_INT          size;          /* IN */
     INT              tracking_on;   /* IN */
     INT              count;         /* IN */
     ACU_CARD_ID*     cards;         /* IN */
   } SW_TRACK_API_PARMS;
```

The `sw_track_api_calls()` function takes a pointer *`track_api`*, to a structure `SW_TRACK_API_PARMS`. The structure must be initialised before invoking the function (see section 2).

Input parameters

size

The `size` field should be set to the size of the input structure in bytes. This may be achieved using `INIT_ACU_STRUCT`

tracking_on

Normally the switch driver does not track application API calls. If this function is called with `tracking_on` set to:

`kSWDiagTrackCmdTrackAPIWithTimestamp` - all further API calls made to the switch driver, up to an internal buffering limit, will be recorded. Information will be written to `stdout`.

`kSWDiagTrackCmdTrackingOff` - tracking is turned off.

count

The number of cards included in `cards`.

cards

The `cards` field is a pointer to an array of `count` card ids. It is up to the application to allocate memory for this array and ensure it is de-allocated once it is finished with.

Returns

On successful completion a value of zero is returned; otherwise one of the following negative values will be returned indicating the type of error:

`ERR_SW_DEVICE_ERROR`	no switch drivers installed, switch driver initialisation failed or device `I/O` error occurred

4 Switching considerations

4.1 General principles

It should be noted that there are two categories of stream timeslot that can be switched to; these are bus timeslots and non-bus timeslots:

- a bus timeslot references the location of a single instance of a bearer channel on a bus, thus specifying this timeslot as an input or an output to a connection references the same bearer channel
- a non-bus timeslot references a pair of a bearer channels, one input and one output (for instance, a network port timeslot has a bearer channel for the speech signal received and one for the speech signal sent), thus a different bearer channel instance is selected depending on whether the timeslot is specified as an input or an output

Applications normally use one of two strategies to manage switching activity:

- Creating connections on demand and breaking them when they are no longer needed
- Creating a set of “nailed up connections” at application start up time, which exist for whole lifetime of application

Sometimes, for example, when using 3rd party cards with Aculab cards, using a mixture of the two strategies may be appropriate.

The architecture of switch devices is such that any given output timeslot can only be driven from one specific input timeslot. If a new switch connection is made to an output timeslot that is already in use, then the new input timeslot will replace the existing input timeslot.

The application can configure system clocking when it starts up or it can leave the job to the automatic system configuration mechanism (either configured using the Aculab configuration tool or by manually editing the system configuration files).

For an application to configure system clocking itself, it should use the `sw_h100_config_board_clock()` function to set the clock master and clock slaves. Use the `sw_reset_switch()` function to ensure that all existing switch connections are broken. The `port_init()` function can be used to write an idle pattern to each timeslot on a port. It is important to bear in mind that using these functions while another telephony application is running can disrupt that application's activities.

When an application makes connections to network port timeslots, attention should be paid as to when it is safe to do so.

- For CAS protocols, where signalling is performed using the bearer channel, no switch connection should be made until the call reaches its CONNECTED state. Then when the switch connection has been finished, and before the call is disconnected, the Call Control API routine `idle_net_ts()` should be invoked.
- For ISDN protocols, switch connections can be made at an earlier stage if required, but `idle_net_ts()` should still be invoked once the connection has been finished with. This use of `idle_net_ts()` replaces a `DISABLE_MODE sw_set_output()` call.

When implementing an on-demand switching strategy, applications must have a scheme for managing the use of timeslots on the expansion bus. They must make sure that they always disable outputs to the expansion bus once a set of switch connections is finished with, otherwise bus contention between cards attempting to drive the same expansion bus timeslot could occur.

When using nailed up connections, it should be noted that if a connection has no driven input (say from another card on the expansion bus) the connection could pick up a signal from an adjacent timeslot. If this poses a problem, for example, because the nailed up connection is an input to a conference, then the application can arrange for a card to output to this timeslot in `PATTERN_MODE` until the time that the input can be driven, at which point the pattern output can be replaced by the required connection.

The same pickup phenomenon can occur for outputs to network port timeslots if the output for a network port is set to `DISABLE_MODE` while a phone call is still connected. Use `sw_set_output()` with *`mode`* set to `PATTERN_MODE` to output a constant A-law (0xD5) or Mu-law (0xFF) silence octet.

4.2 H.100 switching

Aculab cards have enough switching capacity for all port and Prosody module timeslots to have full duplex connections to the bus.

Local switching between on-card resources is also possible.

There are no conventions on stream usage for switching data between network and resource cards.

For the H-Bus there are 32 streams with 128 timeslots each. Streams are referenced with the same stream number for both input and output.

Below is an example code fragment showing how four connections may be made to make a connection over the H-Bus between two network port timeslots on two different Aculab cards (`card_id_a` and `card_id_b`).

```
OUTPUT_PARMS halfParms;

halfparms.ost      =0;
halfparms.ots      =0;
halfparms.mode     =CONNECT_MODE
halfparms.ist      =32;
halfparms.its      =24;
sw_set_output(card_id_a, &halfParms);

halfparms.ost      =33;
halfparms.ots      =14;
halfparms.mode     =CONNECT_MODE
halfparms.ist      =0;
halfparms.its      =0;
sw_set_output(card_id_b, &halfParms);

halfparms.ost      =0;
halfparms.ots      =64;
halfparms.mode     =CONNECT_MODE
halfparms.ist      =33;
halfparms.its      =14;
sw_set_output(card_id_b, &halfParms);

halfparms.ost      =32;
halfparms.ots      =24;
halfparms.mode     =CONNECT_MODE
halfparms.ist      =0;
halfparms.its      =64;
sw_set_output(card_id_a, &halfParms);
```

5 Clocking considerations

5.1 Clocking of cards and expansion buses

When cards interconnected by an expansion bus exchange data across that bus, it is essential that all the cards on the bus are correctly synchronized (so that the switching of data to/from expansion bus timeslots by different cards is synchronized). This synchronization is achieved by configuring expansion bus clocking.

Current Aculab cards may be integrated into the following expansion bus type:

- H.100 bus

This bus type requires exactly one card to act as the clock master on the bus. All other cards must act as clock slaves (an H-Bus secondary clock master slaves off the primary H-Bus clocks and so can be classed as a clock slave here).

If a card acts as clock master on a bus, the mastered bus clock is generated from the card's clock generation circuit. This clock generation circuit may be configured to obtain timing information from a number of reference sources including for example:

- Network port signal
- Local oscillator
- A primary clock or secondary clock on a expansion bus

5.1.1 Failsafe mode

When a Prosody X card is configured to use an external clock source (i.e. a network port or an H-bus clock) and that source has failed, then if fallback has not been configured (or has also failed), the card will resort to using a local clock source. The card is then referred to as being in failsafe clock mode. The following events will force a card into failsafe mode:

- A card configured as an H-bus slave, with fallback disabled, loses its H-bus primary clock reference.
- A card configured as an H-bus master, with fallback disabled, loses its H-bus or network port clock source.
- A card configured as an H-bus master, with fallback enabled, loses both its primary and its secondary (fallback) references.

While a card is in failsafe mode it will not be synchronized to any other cards on the H-bus and will not be clocked from an external network reference. The card will exit failsafe mode and start using its configured primary clock reference as soon as the primary reference is restored.

The `swcmd` utility (see Appendix D) may be used to query the H.100 clock mode of a card (`swcmd –e`, or `swcmd -i <serial no.>`). These commands will print out the current clock setting of a card and also indicate when a card is in failsafe mode, as shown below.

```
swcmd –i <123456>
123456: SLAVE:    H100-A-clocks A=BAD     B=BAD     NR1=UNKNOWN NR2=UNKNOWN
123456: Using failsafe clock - not synchronized to H.100 bus
```

In this case, the card is configured as an H.100 'A' clocks slave, but the 'A' clocks are "`BAD`", so the card is operating in failsafe clock mode

```
swcmd -i 218170
218170: A-MASTER: Network(1)   A=GOOD    B=BAD     NR1=BAD     NR2=BAD
218170: Using failsafe clock - not synchronized to H.100 bus
```

Here the card is configured as an H.100 'A' clocks master, but there is no valid clock

reference on network port 1. The ‘A’ clocks are “GOOD” as the card is driving the H.100 bus ‘A’ clocks using the failsafe clock reference.

5.1.2 H.100 bus clocking

The ECTF document “H.100 Hardware Compatibility Specification: CT Bus” contains further information on the H.100 bus signals referred to below.

The H.100 bus has three sets of signals used for synchronising data transfer across the bus and distributing network timing among multiple network cards, these are the ‘A’ clocks, the ‘B’ clocks and the CT_NETREF fallback clock signal.

The use of these clock signals by a card on the bus depends on the clock-master/clock-slave role that the card has being assigned on the H.100 bus during system initialisation and later by the application API calls.

Initial H.100 bus clocking will be set up during switch driver initialisation, (see the section “Controlling clocking set up during system initialisation”).

Single clock master/multiple clock slaves configuration

In the simplest case, one card would be designated as H.100 primary clock master and drive the H.100 ‘A’ clocks, and all other cards would be designated clock slaves and take their timing from the H.100 ‘A’ clocks. The H.100 ‘B’ clocks would not be used.

The following code fragment shows how such a clocking scenario may be set up using the H.100 switch driver API clock control calls. Here the first card is set up to be H.100 primary clock master driving ‘A’ clocks using its first network port as a timing reference source. A second card is set up to be an H.100 bus clock slave synchronized to H.100 ‘A’ clocks.

```
H100_CONFIG_BOARD_CLOCK_PARMS h100MasterClocks;
H100_CONFIG_BOARD_CLOCK_PARMS h100SlaveClocks;

h100MasterClocks.clock_source      = H100_SOURCE_NETWORK;
h100MasterClocks.network           = 1;
h100MasterClocks.h100_clock_mode   = H100_MASTER_A;
h100MasterClocks.auto_fall_back    = H100_FALLBACK_DISABLED;

sw_h100_config_board_clock(card_id_a, &h100MasterClocks);

h100SlaveClocks.clock_source       = H100_SOURCE_H100_A;
h100SlaveClocks.h100_clock_mode    = H100_SLAVE;
h100SlaveClocks.auto_fall_back     = H100_FALLBACK_DISABLED;

sw_h100_config_board_clock(card_id_b, &h100SlaveClocks);
```

Handling trunk failures and clock fallback

In the simple system clocking scenario described in the previous section, if the trunk connected to network port being used as a reference source by the H.100 bus primary clock master were to go out of service, the H.100 bus timing would go into holdover mode and eventually become de-synchronized from the network. This situation can be avoided if an alternative trunk, synchronized to the same network, is connected to another network port on any of the cards on the H.100 bus. This alternative trunk can be nominated, through the switch driver API call `sw_h100_config_netref_clock()`, to be used as a “fallback” reference source for the card acting as H.100 primary clock master. Once nominated, timing from the fallback trunk will be used to drive the H.100 CT_NETREF clock signal. As soon as the primary clock master detects that it can no longer obtain network timing information from its usual trunk, it will fallback to use timing from this CT_NETREF signal. Primary master fallback is a “Stratum 4 Enhanced” compatible changeover and will not cause any disruption to the H.100 primary clock.

Only one card at any time may drive the H.100 CT_NETREF signal. It may be driven at one of three possible clock speeds 8kHz, 1.544MHz or 2.048MHz. The rate it is driven at is immaterial but all the cards on the H.100 bus must be configured to use it at the same rate. If CT_NETREF is driven from a network port loaded with E1 firmware, it can be driven at either 8kHz or 2.048MHz. If CT_NETREF is driven from network port loaded with T1 firmware, it can be driven at either 8kHz or1.544 MHz.

The code fragment that follows shows how an alternative trunk may be set up to provide a fallback clock reference for the primary clock master.

The primary clock master must have its auto-fallback capability enabled and the CT_NETREF speed must be specified for all cards. In this example the nominated alternative trunk is attached to a network port on a card acting as an H.100 clock slave. It is equally possible to nominate an alternative trunk connected to a primary/secondary clock master card.

```
H100_CONFIG_BOARD_CLOCK_PARMS h100MasterClocks;
H100_CONFIG_BOARD_CLOCK_PARMS h100SlaveClocks;
H100_NETREF_CLOCK_PARMS       h100FallbackClocks;

h100MasterClocks.clock_source         = H100_SOURCE_NETWORK;
h100MasterClocks.network              = 1;
h100MasterClocks.h100_clock_mode      = H100_MASTER_A;
h100MasterClocks.auto_fall_back       = H100_FALLBACK_ENABLED;
h100MasterClocks.netref_clock_speed   = H100_NETREF_8KHZ;

sw_h100_config_board_clock(card_id_a, &h100MasterClocks);

h100SlaveClocks.clock_source          = H100_SOURCE_H100_A;
h100SlaveClocks.h100_clock_mode       = H100_SLAVE;
h100SlaveClocks.auto_fall_back        = H100_FALLBACK_DISABLED;
h100SlaveClocks.netref_clock_speed    = H100_NETREF_8KHZ;

sw_h100_config_board_clock(card_id_b, &h100SlaveClocks);

h100FallbackClocks.network            = 1;
h100FallbackClocks.netref_clock_mode  = H100_ENABLE_NETREF;
h100FallbackClocks.netref_clock_speed = H100_NETREF_8KHZ;

sw_h100_config_netref_clock(card_id_b, &h100FallbackClocks);
```

An application can determine if the primary master card has fallen back to the alternative trunk by using the switch driver API call sw_h100_query_board_clock(). This call can also be used with other cards to determine the status of various H.100 bus clock signals – for example to see if CT_NETFREF is being driven:

```
H100_QUERY_BOARD_CLOCK_PARMS h100ClocksStatus;

sw_h100_query_board_clock(card_id_a, &h100ClocksStatus);

if (h100ClocksStatus.fall_back_occurred)
{
  /* Default trunk for primary clock master has gone out of service or
   * failed in some other way.
   * Primary master clocks being driven from CT_NETREF.
   */
}

if (h100ClocksStatus. netref_a_clock_status != H100_CLOCK_STATUS_GOOD)
{
  /*
   * Fallback trunk must also have failed.
   * Perform some action - e.g. alert operator
   */
}
else
{
  /* Primary clock master has fallen back to alternate trunk
   * Perform some action - e.g. alert operator
   */
}
```

The application can use a call control driver API call to obtain layer 1 statistics for a given network port, this information may be used to obtain more information about a trunk failure. It may also be used to detect when the failed trunk comes back into service.

The primary clock master can be configured to return automatically to using the original trunk as a reference source when timing information can once more be recovered from it. To enable this feature, an auto-return bit is OR'ed with the fallback enable bit during the initial primary clock master clock set up.

```
H100_CONFIG_BOARD_CLOCK_PARMS h100MasterClocks;

h100MasterClocks.clock_source       = H100_SOURCE_NETWORK;
h100MasterClocks.network            = 1;
h100MasterClocks.h100_clock_mode    = H100_MASTER_A;
h100MasterClocks.auto_fall_back     = (H100_FALLBACK_ENABLED+H100_AUTO_RETURN);
h100MasterClocks.netref_clock_speed = H100_NETREF_8KHZ;

sw_h100_config_board_clock(card_id_a, &h100MasterClocks);
```

Fallback has occurred and auto-return was not enabled, then the application must make new a call to `sw_h100_config_board_clock()` in order to return the primary master clock reference to the original trunk (with the same set of parameters as before).

Handling failure of primary clock master

As mentioned previously, the H.100 bus has two sets of parallel clocks that slave cards can synchronize to, the 'A' clocks and the 'B' clocks. For simple system configurations, an application can restrict itself to just using the 'A' clocks. If a system has a requirement to handle, with minimum disruption, situations when the card acting as primary clock master fails - possibly due to hardware failure - then both sets ('A' and 'B') clocks must be configured using switch driver API calls. In this case, the two sets of clocks are symmetric with one set being driven by a primary clock master and the other set by another card acting as secondary clock master.

Normally secondary clock timing will be locked to the primary master clocks. However, if the primary clock master card fails, then the secondary clock master card will change over its reference source to one of its network ports (or to `CT_NETREF`). All the H.100 slave cards will change over from slaving off the primary master clocks to slaving off the secondary master clocks. When such a change over occurs, the secondary clock master is promoted

to become the new primary clock master. Use of a completely different set of clock signals during and after change over averts any danger of clock contention between the original failed primary clock master and the newly promoted primary clock master. The clock changeover on slave cards resulting from a primary master hardware failure is not required to be a “Stratum 4 Enhanced” compatible changeover, and it may cause brief disruption to the clocks on the slave cards.

On initial system configuration, the primary clock master would normally drive the H.100 ‘A’ clocks and the secondary clock master would normally drive the H.100 ‘B’ clocks (from timing derived from ‘A’ clocks). If the primary clock master were then to fail, then the secondary clock master would be promoted to become the new primary clock master. (Thus the roles of the ‘A’ clocks and ‘B’ clocks would be reversed and if a new secondary clock master were to be configured subsequently, it would be configured to drive the ‘A’ clocks.)

The following code fragment shows how a multi-card system may be set up with the first card being configured to be primary clock master driving ‘A’ clocks. The second card is configured to be secondary clock master driving ‘B’ clocks from timing it derives from ‘A’ clocks.

```
H100_CONFIG_BOARD_CLOCK_PARMS h100PriMasterClocks;
H100_CONFIG_BOARD_CLOCK_PARMS h100SecMasterClocks;
H100_CONFIG_BOARD_CLOCK_PARMS h100SlaveClocks;

h100PriMasterClocks.clock_source      = H100_SOURCE_NETWORK;
h100PriMasterClocks.network           = 1;
h100PriMasterClocks.h100_clock_mode   = H100_MASTER_A;
h100PriMasterClocks.auto_fall_back    = H100_FALLBACK_DISABLED;

sw_h100_config_board_clock(card_id_a, &h100PriMasterClocks);

h100SecMasterClocks.clock_source      = H100_SOURCE_H100_A;
h100SecMasterClocks.h100_clock_mode   = H100_MASTER_B;
h100SecMasterClocks.auto_fall_back    = H100_FALLBACK_DISABLED;

sw_h100_config_board_clock(card_id_b, &h100SecMasterClocks);
```

It will take up to 1 second for secondary ‘B’ clocks to become phase aligned with the ‘A’ clocks following the call to `sw_h100_config_board_clock()` on `card_id_b`.

In the above example, the secondary clock master has not been configured to make an automatic change over to another clock reference in the event of primary clock master failure. Normally the secondary clock master would be required to make such an automatic change over. The following code fragment shows how the secondary clock master could be pre-configured to automatically changeover to become a primary clock master whose reference source would be its first network port.

```
h100SecMasterClocks.clock_source      = H100_SOURCE_H100_A;
h100SecMasterClocks.h100_clock_mode   = H100_MASTER_B;
h100SecMasterClocks.auto_fall_back    = (H100_FALLBACK_ENABLED +
                                         H100_CHANGEOVER_TO_NETWORK);
h100SecMasterClocks.network           = 1;

sw_h100_config_board_clock(card_id_b, &h100SecMasterClocks);
```

The following code fragment shows how the secondary clock master could be pre-configured to change over to become a primary clock master whose reference source would be `CT_NETREF` (driven from another card):

```
h100SecMasterClocks.clock_source      = H100_SOURCE_H100_A;
h100SecMasterClocks.h100_clock_mode   = H100_MASTER_B;
h100SecMasterClocks.auto_fall_back    = (H100_FALLBACK_ENABLED +
                                         H100_CHANGEOVER_TO_NETREF);
h100SecMasterClocks.network           = 1;

sw_h100_config_board_clock(card_id_b, &h100SecMasterClocks);
```

A third card, operating as an H.100 'A' clocks slave, may be set up to automatically switch over to the 'B' clocks following primary clock master card failure. The slave card clocking would be configured as shown in the following code fragment:

```
h100SlaveClocks.clock_source          = H100_SOURCE_H100_A;
h100SlaveClocks.h100_clock_mode       = H100_SLAVE;
h100SlaveClocks.auto_fall_back        = H100_FALLBACK_ENABLED;

sw_h100_config_board_clock(card_id_c, &h100SlaveClocks);
```

When a card is configured to operate in `H100_SLAVE` mode, the auto_fall_back parameter is used to enable clock slave swap over from 'A' clocks to 'B' clocks (not used to control fallback to CT_NETREF).

If, for some exceptional reason, no automatic slave clock change over is required, then the `auto_fall_back` parameter should be set to `H100_FALLBACK_DISABLED`.

The status of the 'A' clocks and 'B' clocks, and the primary/secondary master status of a card can be determined at any time using the switch driver API call `sw_h100_query_board_clock()`. (If a secondary clock master has become promoted to a primary clock master, then the returned `clock_source` parameter will reflect this by returning a value that is not the original set of H.100 bus clocks specified, instead it will be set to specify the alternate set).

As previously described, the secondary clock master is pre-configured with a reference source to be used if it becomes promoted to primary clock master. If this reference source is a trunk attached to a network port on the secondary master, then it is possible to also pre-configure a fallback to `CT_NETREF` should that trunk fail after the card has been promoted to primary master:

```
h100SecMasterClocks.clock_source      = H100_SOURCE_H100_A;
h100SecMasterClocks.h100_clock_mode   = H100_MASTER_B;
h100SecMasterClocks.auto_fall_back    = (H100_FALLBACK_ENABLED +
                                         H100_CHANGEOVER_TO_NETWORK +
                                         H100_CHANGEOVER_TO_NETREF);
h100SecMasterClocks.network           = 1;

sw_h100_config_board_clock(card_id_b, &h100SecMasterClocks);
```

Primary H.100 clock master changeover

Sometimes it may be necessary to changeover the role of H.100 bus primary clock master from one card to another. Care must be taken during such a changeover so that:

- cards do not become de-synchronized from H.100 bus
- cards do not cause contention on the H.100 bus
- discontinuities of clocks on cards with DSP devices do not occur. Discontinuities can cause DSPs to lose synchronization with the card.

Such a changeover can be achieved by the use of primary and secondary clocks and appropriate programming of secondary master clock fallback. The example below assumes that the primary master is driving the 'A' clocks:

- set up the card that is to become the new clock master (it needs a network port) as secondary clock master (driving ‘B’ clocks), slaving from primary clocks (‘A’ clocks) with fallback to network port enabled
- wait 1 second for secondary clocks to become phase aligned with primary clocks
- for all other slave cards in system, change clock reference from primary clocks to secondary clocks
- change primary master card to become ‘B’ clocks slave, the secondary master will at this point automatically fallback, changing its clock source to the pre-configured network port (because ‘A’ clocks will have disappeared) and become the new H.100 bus primary master (driving ‘B’ clocks)

In the above example, if secondary master fallback had not been used to change over to the new primary clock reference, and the secondary reference clock source had been changed by a direct call to `sw_h100_config_board_clock()`, then for a short period of time the original clock master would not be synchronized with other cards on H.100 bus. This would not cause a problem provided that there were no switch connections to or from the H.100 bus on the original clock master during the changeover. However, if there were such connections, then the card’s desynchronized state might cause H.100 bus contention to occur until its clock reference was changed to ‘B’ clocks slave.

5.2 Controlling clocking set up during system initialisation

The configuration of a card to act as bus clock master and of other cards to operate as clock slaves can occur during any of the following:

- system boot up time through the automatic configuration mechanism
- use of the Aculab Configuration Tool
- application run time through explicit switch API calls (`sw_h100_config_board_clock()`, `sw_clock_control()`)

Once a card has been configured as a bus clock master or bus clock slave it will remain configured in the same way until a switch API call is made to change clock configuration.

5.2.1 Editing files used by the automatic configuration mechanism

The clocking settings for cards in the system are automatically applied at start up by a tool called `config`. This tool uses configuration files to determine the appropriate setting for each card in the system. These configuration files can be automatically generated using the Aculab Configuration Tool. It is also possible to edit these files manually.

Configuration files are found in a directory called `ACULAB_ROOT\Cfg`. Each configuration filename incorporates the serial number of its associated card. Hence, a card with a serial number of `123456` will be configured using a file called `123456.cfg`.

The section in the configuration file that controls the clocking configuration starts with the line `"[Switch]"` and ends with the line `"[EndSwitch]"`. For example, the section may look like this:

```
[Switch]
CtBusTermination=TRUE
CtBus=SWMODE_CTBUS_H100
Source=H100_SOURCE_INTERNAL
Network=0
H100Mode=H100_MASTER_A
AutoFallBack=H100_FALLBACK_DISABLED
NetRefClockSpeed=H100_NETREF_8KHZ
[EndSwitch]
```

Each line in the section consists of a case sensitive field name followed by an equals sign and then the value assigned to that field.

H.100 configuration

Source - controls the source of the card's clock. This can be one of the following:

`H100_SOURCE_INTERNAL` - generate a clock on board and provide it to the CT bus

`H100_SOURCE_NETWORK` - use the port specified in the network field as the clock reference

`H100_SOURCE_H100_A` - take the clock from the H.100 clock master A

`H100_SOURCE_H100_B` - take the clock from the H.100 clock master B

Network - controls the network port that is used as a clock reference if `Source` is set to `H100_SOURCE_NETWORK`. The network port index is the physical index of the port on the card. Remember that for the Switch API, ports are numbered from 1.

H100Mode - controls whether the card is an H.100 master or an H.100 clock slave. The possible values for this field are:

`H100_SLAVE` - card will be a slave on the H.100 bus (`Source` must be one of `H100_SOURCE_H100_A` or `H100_SOURCE_H100_B`)

`H100_MASTER_A` - card will drive the 'A' clocks on the H.100 bus (`Source` must be one of `H100_SOURCE_INTERNAL`, `H100_SOURCE_NETWORK`, or `H100_SOURCE_H100_B`).

`H100_MASTER_B` - card will drive the 'B' clocks on the H.100 bus (`Source` must be one of `H100_SOURCE_INTERNAL` , `H100_SOURCE_NETWORK`, or `H100_SOURCE_H100_A`).

AutoFallBack - This field determines the card's behavior when an H.100 clock failure occurs. The values in this field depend on the `H100Mode` field. Examples are:

```
H100_FALLBACK_DISABLED
H100_FALLBACK_ENABLED
H100_FALLBACK_ENABLED | H100_AUTO_RETURN
H100_FALLBACK_ENABLED | H100_CHANGEOVER_TO_NETWORK
```

See the description of the `auto_fall_back` field in the documentation for the `sw_h100_config_board_clock()` function for an explanation of how these fields apply.

NetRefClockSpeed - This field is used to tell the card the speed of the `CT_NETREF` fallback clock. See the description of the `netref_clock_speed` field in the documentation for the `sw_h100_config_board_clock()` function for further details.

CtBusTermination - This sets H.100 bus termination. Possible values are:

`TRUE` - the card should terminate the H.100 bus

`FALSE` - the card should not terminate the H.100 bus

After editing the configuration file, the new settings are applied by running the `config` tool:

```
config  <card serial number>
```

6 Troubleshooting

If you think your application has made all the correct switch connections to an expansion bus but for some reason you think no speech path has been established, verify the following:

- the switch connections really have been made and still exist (use the `sw_query_output()` call to verify each switch connection you think your application has made is still there)
- the expansion bus clocking has been set up correctly (use the `sw_query_clock_control()` call to verify each card's clock set up, remember firmware download or restart can sometimes effect clocking)
- the expansion bus clock master has not been abruptly changed from one card to another (this may cause some resource and network cards to fail to switch data properly)
- your application has not inadvertently caused bus contention to occur (two cards outputting to the same expansion bus timeslot). When a connection up to the expansion bus from a card is no longer needed, the application should apply `sw_set_output()` with mode set to `DISABLE_MODE` to that expansion bus timeslot
- the cable connecting cards to the expansion bus is not faulty and is not excessively long or twisted.
- your application was compiled so that the size of the switch driver API parameter structures were the same as that expected by the switch driver library for your target operating system

You can check if the expansion bus is operating correctly by switching a constant pattern onto an expansion bus timeslot using `sw_set_output()` in `PATTERN_MODE` and using `sw_sample_input()` to read back the pattern from a different card.

If the quality of the signal on the speech path seems distorted, verify:

- appropriate firmware is being used for the Aculab card (clicking sounds on a speech path may originate from clock slips, possibly a non-CRC variant of signaling system firmware should be used instead of a CRC variant, or vice-versa)
- the Aculab card is appropriately set up to be synchronized with the network clock or, exceptionally, to provide timing information to the network
- the expansion bus clocking has been set up correctly
- your application has not inadvertently caused bus contention to occur (two cards outputting to the same expansion bus timeslot). When a connection up to the expansion bus from a card is no longer needed, the application should invoke `sw_set_output()` with mode set to `DISABLE_MODE` for that expansion bus timeslot
- the signal has the appropriate encoding for the network or speech processing resource (A-law or μ-law)
- check expansion bus has been terminated appropriately
- if expansion bus is being used at or near its loading limit, locate the card that is the bus clock master half way along the H.100 bus ribbon cable

You can check how expansion bus clocking has been set up using the switch driver API

call `sw_query_clock_control()`.

If an expansion bus timeslot is not driven by any card, do not assume the sampled values will be equivalent to a silent (DC) signal whose eight bit samples are 0xff. In fact, such a non-driven expansion bus timeslot may pickup a signal from an adjacent timeslot. The same is true for local resource and network port timeslots, which are not driven (outputs set to `DISABLE_MODE`). See section 4.1 for more details.

The switch driver reports an error `ERR_SW_NO_RESOURCES` when your application tries to make a connection to/from an expansion bus timeslot. Check your application has disabled each previous expansion bus connection after it was no longer required using a call to `sw_set_output()` with mode set to `DISABLE_MODE`. There are limits to the number of connections that can be made up to and down from the expansion bus for certain types of card.

In a multi-tasking operating system you can run, concurrently with your application, a diagnostic application that makes calls to `sw_track_api_calls()` to observe switch driver calls made by your application.

The `swcmd` utility described in Appendix D may be used as a troubleshooting tool.

Appendix A: Aculab card stream numbering and clock settings

A.1 Prosody X PCIe rev 3 card stream usage

Stream names	Available channels	Stream numbers	Details	Notes
H.100 bus streams D0-D31	0-127	0-31	H.100 bus streams D0…D31	1
Network Ports 1-8	E1: 1-15,17-31 T1: 0-22,23	32-39	Port 1-8 bearer channels	2,3
Signalling DSP0 A1-A4	0-31	48-51		4
Signalling DSP0 B1-B4	0-31	52-55		4
Signalling DSP1 A1-A4	0-31	56-59		4
Signalling DSP1 B1-B4	0-31	60-63		4
TiNG1 0-1	0-127	64-65		5
TiNG2 0-1	0-127	66-67		5
TiNG3 0-1	0-127	68-69		5
TiNG4 0-1	0-127	70-71		5

1. Prosody X PCIe rev 3 cards can only be interconnected via an H.100 bus.
2. Use of timeslot 23 depends on the type of signalling used by the T1 firmware.
3. The number of network ports available depends on the card variant.

   Maximum number of network ports permitted on each card type is as follows:

Card type	Maximum ports	Stream numbers
Prosody X PCIe rev 3	8	32-39

4. Prosody X PCIe rev 3 cards provide signalling capability equivalent to 4 signalling DSPs.

5. The number of TiNG DSPs available depends on the card variant

Maximum number of TiNG DSPs available on each card type is as follows

Card type	Maximum TiNG DSPs	Stream numbers
Prosody X PCIe rev 3	4	64-71

A.2 Prosody X PCIe rev 3 card clock settings

Prosody X PCIe rev 3 cards can only be configured to operate the CTBus in H.100 mode.

CT Bus clock control is best achieved via the `sw_h100_config_board_clock()` call. If, however, an application makes calls to `sw_clock_control()`, the following clock modes are applicable. Note that in switch API calls, network port numbering starts from 1.

Reference source	sw_clock_control() parameter	Value
Network ports 1-3	CLOCK_REF_NET1	0x0003
	CLOCK_REF_NET2	0x0004
	CLOCK_REF_NET3	0x000c
	CLOCK_REF_NET4	0x000d
Network ports 4-7	CLOCK_REF_NET5	0x0005
	CLOCK_REF_NET6	0x000a
	CLOCK_REF_NET7	0x000b
	CLOCK_REF_NET8	0x000e
H.100 bus clock	CLOCK_REF_H100	0x0007
Local oscillator	CLOCK_REF_LOCAL	0x0002

The card will act as H.100 bus 'A' clocks master in all of these clock modes except `CLOCK_REF_H100`, where the card is configured as an H.100 bus clock slave.

A.3 Prosody X 1U Enterprise stream usage

The following streams are available on the Prosody X 1U:

Stream names	Available channels	Stream numbers	Details	Notes
Network Ports 1-4	E1: 1-15,17-31 T1: 0-22,23	32-35	Port 1-4 bearer channels	
Signalling DSP0 A1-A4	0-31	48-51		
TiNG1 0	0-127	64		1
TiNG2 0	0-127	66		1

1. Unlike Prosody X PCIe rev 3 cards, each TiNG DSP only has a single stream of 128 timeslots.

A.4 Prosody X 1U Enterprise clock settings

Clock control is best achieved via the `sw_h100_config_board_clock()` call. If, however, an application makes calls to `sw_clock_control()`, the following clock modes are applicable. Note that in switch API calls, network port numbering starts from 1.

Reference source	sw_clock_control() parameter	Value

Network ports 1-3	`CLOCK_REF_NET1`
	`CLOCK_REF_NET2`
	`CLOCK_REF_NET3`
	`CLOCK_REF_NET4`
		`0x0003`
		`0x0004`
		`0x000c`
		`0x000d`

Local oscillator	`CLOCK_REF_LOCAL`	`0x0002`

Appendix B: Sampling bearer channels

A bearer channel has a data rate of 64000 bits a second, which are divided up into 8000 samples of 8 bits (an octet) a second. The switch driver API gives an application the ability to determine the instantaneous value of a single sample from a bearer channel. However, it is not possible for an application to record or process all the samples on a bearer channel using `sw_sample_input()`. If an application requires this kind of functionality, the bearer channel should be switched through to some kind of recording resource (such as a Prosody module).

Appendix C: API error codes

As defined in `sw_lib.h`

Error	Value	Meaning
SUCCESS	0	Command executed successfully
ERR_SW_INVALID_COMMAND	-200	Command code is not supported
ERR_SW_DEVICE_ERROR	-202	No switch drivers installed, switch driver initialisation failed or device I/O error occurred
ERR_SW_NO_RESOURCES	-204	Not enough switching paths left
ERR_SW_INVALID_SWITCH	-209	Switch driver selector out of range
ERR_SW_INVALID_STREAM	-210	Stream number in parameter list is out of range
ERR_SW_INVALID_TIMESLOT	-211	Timeslot in parameter list is out of range
ERR_SW_INVALID_CLOCK_PARM	-213	Invalid clock configuration parameter
ERR_SW_INVALID_MODE	-216	Incorrect command mode
ERR_SW_INVALID_MINOR_SWITCH	-217	Minor internal switch error
ERR_SW_INVALID_PARAMETER	-218	General invalid parameter error
ERR_SW_NO_PATH	-220	Refer to notes in description of `sw_set_output()` API call
ERR_SW_PATH_BLOCKED	-226	Switch connection cannot be made
ERR_SW_OS_INTERRUPTED	-227	Event wait interrupted
ERR_SW_OS_INTERRUPTED	-227	Event wait interrupted
ERR_SW_PORT_NOT_LOADED	-229	Firmware is not running on network port
ERR_SW_PORT_RATE_MISMATCH	-230	NETREF rate does not match line rate
ERR_SW_PMX_FPGA	-231	Request not supported by PMX FPGA
ERR_SW_COMPONENT_MISMATCH	-232	Prosody X software component versions do not match (PXSCS and t8110.ko)
ERR_SW_NO_SUCH_DSP	-233	No DSP is fitted in requested position
ERR_SW_NO_SUCH_DSP_PORT	-234	Requested DSP serial port does not exist
ERR_SW_PXSCS_TIMED_OUT	-235	API call has timed out
ERR_SW_PXSCS_UNKNOWN_API_CALL	-236	New PX_Sw_Driver package may be required
ERR_SW_T8110_UNKNOWN_IOCTL	-237	New Prosody_IP_Firmware may need to be flashed
ERR_SW_PXSCS_COMMS_FAILED	-238	Problem communicating with pxscs on Prosody X card
ERR_SW_API_CALL_ABORTED	-239	An application has called `sw_abort_api_calls()`

Error	Value	Meaning
ERR_SW_SIZE_PARAMETER	-240	`size` field in struct used in API call is less than the `size` expected by the switch library

Appendix D: Using swcmd

Introduction

`swcmd` is a command line utility/diagnostic tool which may be used to exercise the functionality of the Aculab switch driver API for Aculab cards, allowing display and alteration of card clocking and switching modes, and diagnosis of switch path integrity problems.

Different builds of the `swcmd` executable exist for each operating system supported by the Aculab switch API.

Under multi-tasking operating systems, such as Windows and Unix, `swcmd` may be run concurrently with an application. It can be used to verify which switch connections are made; to determine the current clocking mode of each card, and even to output a log of switch API calls made by the application.

Command line syntax

The `swcmd` command line must specify one or more flags, each flag followed by a space-separated list of parameters, for example:

```
swcmd –t 12345 2 –t 67890 1 –v
```

Which equates to:

```
swcmd <flag> <serial no> <clkmode> <flag> <serial no> <clkmode> <verbose>
```

Numeric values may be specified as decimal (default) or as C style hex values (e.g. 0x72).

The following parameter types occur in parameter lists:

`<serial no>` - card serial number (e.g. `12345`)
`<ost>` - output stream specifier
`<ots>` - output timeslot specifier
`<ist>` - input stream specifier
`<its>` - input timeslot specifier

The following flags are supported:

<Flag>	Parameters	Use
`-v`	*none*	Run `swcmd` tool in verbose output mode. Print out driver version and card information. Invokes `sw_ver_switch()` and `sw_card_info()`
`-t`	`<serial no>` `<clkmode>`	Set card clock circuit reference and/or expansion bus clock mastering/slaving mode using `<clkmode>` for card `<serial no>`. Invokes `sw_clock_control()`.
`-ti`	*none*	Set clock mode interactively
`-e`	*none*	Display system clocking for all cards.
`-c`	`<serial no>` `<ost> <ots>` `<ist> <its>`	Make switch connection on card `<serial no>` switching data from `<ist,its>` to `<ost,ots>`
`-o`	`<serial no>` `<ost> <ots>` `<pat>`	Output constant pattern `<pat>` on card `<serial no>` on `<ost,ots>`
`-d`	`<serial no>` `<ost> <ots>`	Disable switch output <`ost,ots`> on card `<serial no>`

<Flag>	Parameters	Use
-q	<serial no> <ost> <ots>	Determine source of data being switched to <ost,ots> by card <serial no>. Invokes sw_query_output().
-y	<serial no>	Report the number of H-bus timeslots driven by card.
-a	<serial no> <ist> <its>	Obtain 8-bit sample of data currently being received by card <serial no> on <ist,its>. Invokes sw_sample_input().
-w	<predicted sample>	When used with -l, allows swcmd to loop, sampling data until value not equal to <predicted sample> is obtained.
-r	<serial no>	Invokes sw_reset_switch()for card <serial no>
-k	*none*	Monitor switch driver API calls being made by an application to standard output, repeatedly invokes sw_track_api_calls().
-2	*none*	Turn switch driver API call tracking on.
-0	*none*	Turn switch driver API call tracking off.
-?	*none*	Output swcmd command line syntax.
-h	<serial no> <src> <net> <mode> <afb> <ncs>	Configure H.100 clocks. Invokes sw_h100_config_board_clock() for card <serial no>
-hi	*none*	Configure H.100 clocks interactively
-i	<serial no>	Query H.100 clocks. Invokes sw_h100_query_board_clock() for card <serial no>
-j	<serial no> <net> <mode> <ncs>	Configure CT_NETREF. Invokes sw_h100_config_netref_clock() for card <serial no>
-ji	*none*	Configure CT_NETREF interactively
-u	<serial no>	Query H.100 CT_NETREF. Invokes sw_h100_query_netref_clock() for card <serial no>
-n	<repeat-count>	Iterate swcmd command – see below.
-l	*none*	Loop forever invoking the switch driver calls specified by other swcmd flags.
-f	<serial no> <stream> <rx_mode> <tx_mode>	Sets the companding mode on a network port stream.
-ft	<serial no> <stream> <timeslot> <rx_mode> <tx_mode>	Sets the companding mode on a network port timeslot.
-g	<serial no> <stream>	Determines which companding mode has been set on a network port stream
-gt	<serial no> <stream> <timeslot>	Determines which companding mode has been set on a network port timeslot

<Flag>	Parameters	Use
-b	<serial no> <type> <position> <port>	Invokes sw_get_dsp_stream_info()
-s	<serial no> <mode>	Turns the bus termination off (mode=0) or on (mode=1).

Iterating commands

As a convenience some swcmd commands can be iterated by invoking swcmd with the flag to invoke the required command and the "-n" flag to iterate it a given number of times.

For example the command:

```
swcmd -c 12345 16 1 0 0
```

makes a single connection from stream 0 timeslot 0 to stream 16 timeslot 1, whereas the command:

```
swcmd -c 12345 16 1 0 0 -n 15
```

would make 15 connections as follows:

stream 0 timeslot 0 connected to stream 16 timeslot 1
stream 0 timeslot 1 connected to stream 16 timeslot 2
...
stream 0 timeslot 14 connected to stream 16 timeslot 15

Each iterable command has parameters that remain constant (card specifier <serial no> and stream numbers) and parameters that are incremented mod 256 (timeslots and patterns).

The commands invoked by the following flags are iterable:

```
'c', 'o', 'q', 'd', 'a'
```

Examples of use

- Identifying driver versions and card serial numbers

swcmd may be used to display version numbers of all switch drivers installed in the system, identify the type of cards they are associated with and display their serial numbers.

```
swcmd -v
```

- Changing card clock mode

swcmd may be used to change a Prosody X card's clock reference:

```
swcmd -t <serial no> <clkmode>
```

See Appendix A for valid <clkmode> parameters for the various card types.

- Verifying expansion bus operation

A good way to see if a bus is being clocked correctly and to verify H-bus cable integrity is to output a pattern from one card onto the bus, then sample it on a second card.

For example, the first command below will cause card 12345 to output a pattern 0x42 on H-bus stream 0 timeslot 0. The second command will cause card 67890 to sample the pattern on H-bus stream 0 timeslot 0.

```
swcmd -o 12345 0 0 0x42
swcmd -a 67890 0 0
```

would output:

```
sampled 0x42
```

To check every sample on a given timeslot is the expected value, `swcmd` can be invoked in loop mode with a predicted sample value:

```
swcmd -a 12345 16 1 -w 0x54 -l
```

In the above example `swcmd` will continue to execute (with no output) until a non-`0x54` sample is obtained on stream 16 timeslot 1, or `swcmd` is aborted.

- Looping back a network port timeslot

To loopback timeslot 1 on network port 1 of an E1 card, so data received from the network on that B channel is sent straight back again, use `swcmd` as follows:

```
swcmd –c 12345 32 1 32 1
```

See Appendix A for stream/timeslot numbering used for the various card types.

- Tracking application switch driver API calls

Invoke `swcmd` as follows (under Linux use '&' to run `swcmd` as background task; under Windows run in separate command window):

```
swcmd –k –2
```

Trace will appear on `stdout` as an application makes API calls similar to the following:

```
[18:05:35]00003685: 0 <- sw_clock_control(67890,CLOCK_REF_H100)
[18:05:35]00003695: 0 <- sw_set_output(12345,{ost=32,ots=0,
                                    mode=PATTERN_MODE,pattern=0x54})
```

Alternatively, the Aculab `trace_mode` tool may be used to generate a log of switch API calls.