

Software for Aculab digital network access cards

V6 switch API guide

Proprietary information

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

All trademarks are recognised and acknowledged.

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

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

© Aculab Plc 2009: All rights reserved

Revision record

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 16port 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	01.09.09	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. Fonts changed

CONTENTS

1	Introduction	5
1.1	Terminology	5
1.2	Switching	6
1.3	Clock control	7
1.3.1	H.100 bus	7
1.3.2	H.110 bus	8
1.3.3	MVIP bus	8
1.3.4	SC bus	9
1.3.5	Monitoring two party conversations - DSP/Prosody required	9
2	API types, header files and libraries	10
2.1	Operating systems	10
2.1.1	Windows 2000/XP	10
2.2	Opening cards for use with the switch driver	10
3	API call summary and descriptions	12
3.1	sw_ver_switch() - Get switch driver version	14
3.2	sw_mode_switch() - Get switch driver mode	16
3.3	sw_card_info() - Retrieve card details	17
3.4	sw_set_card_notification_queue()	19
3.5	sw_get_card_notification()	20
3.6	sw_get_notification_wait_object()	21
3.7	sw_set_card_h100_termination() – Set up H.100 termination	22
3.8	sw_config_timeslot_companding() – Set up A-law/mu-law conversion	23
3.9	sw_query_timeslot_companding() - Query A-law/mu-law conversion	25
3.10	sw_config_companding() – Set up A-law/mu-law conversion	26
3.11	sw_query_companding() - Query A-law/mu-law conversion	27
3.12	sw_component_version() – Get switch driver component version	28
3.13	sw_get_dsp_stream_info()	30
3.14	sw_abort_api_calls()	31
3.15	sw_set_output() - Control switch matrix	32
3.15.1	Switching restrictions for E1/T1 cPCI 16-port and Prosody X cards:	32
3.16	sw_query_output() - Query switch matrix	35
3.17	sw_switch_override_mode() - Set override mode	36
3.18	sw_sample_input(), (sw_sample_input0()) - Sample timeslot	37
3.19	sw_tristate_switch() - Tristate switch matrix	38
3.20	sw_reset_switch() - Reset switch matrix	39
3.21	sw_clock_control() - Set clock reference	40
3.22	sw_query_clock_control() - Query clock reference	41
3.23	sw_h00_config_board_clock() - Set up H.100 (H-Bus) clocking	42
3.24	sw_h100_config_netref_clock() - Set up H.100 (H-Bus) fallback clock	45
3.25	sw_h100_query_board_clock() - Query H.100 (H-Bus) clock mode	47
3.26	sw_h100_query_netref_clock() - Query H.100 (H-Bus) fallback clock	49
3.27	sw_track_api_calls() - Track API calls	50
3.28	sw_query_switch_caps() - Get switch capabilities	51
4	Switching issues	53
4.1	General principles	53
4.2	H.100/H.110 (H-Bus) switching	54
4.3	MVIP bus switching	55
4.3.1	Switching between network cards and resource cards	55
4.3.2	Switching between multiple network cards	56
5	System clocking issues	58
5.1	Clocking of digital access cards and expansion buses	58
5.1.1	H.100 bus clocking	58
5.1.2	H.110 bus clocking	63

5.1.3	MVIP bus clocking	63
5.1.4	SCbus clocking	65
5.2	Controlling clocking set up during system initialisation	66
5.2.1	Manually configuring the automatic configuration mechanism	66
5.2.2	H.100 configuration	67
5.2.3	Legacy configuration	68
Appendix A:	digital access card stream numbering and clock settings	71
A.1	Prosody PCI/cPCI card stream usage	71
A.2	Prosody PCI/cPCI card clock settings	73
A.3	Prosody X card stream usage	75
A.4	Prosody X card clock settings	76
A.5	E1/T1 PCI card stream usage	77
A.6	E1/T1 PCI card clock settings	79
A.7	E1/T1 cPCI card stream usage	81
A.7.1	PM4 stream numbering	81
A.7.2	PMX stream numbering	82
A.8	E1/T1 cPCI card clock settings	84
A.9	IP telephony PCI H.323 gateway card stream usage	85
Appendix B:	Sampling bearer channels	87
Appendix C:	API error codes	88
Appendix D:	Using swcmd	90

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 Digital Access cards:

- Prosody PCI card (with or without primary rate module)
- E1/T1 PCI card
- Prosody cPCI card (with or without primary rate module)
- E1/T1 cPCI card
- Prosody X PCI card
- Prosody X cPCI card
- Prosody X PCIe card
- IP telephony card

The same API is supported across diverse operating systems.

This document includes the changes to the Aculab switch API for V6 drivers. It may be necessary to refer to the V5 driver switch API guide for documentation regarding functions that are not explicitly documented or have been made obsolete by this document.

In general, the API has changed in the following ways:

- API calls that perform global initialisation and/or querying (most of which are in the V5 Call API) have been replaced by API calls that work on a per-card or per-port basis.
- Resources are no longer globally opened by every application but must be enumerated and explicitly opened.
- A number of state changes that the V5 driver required you to poll for are now notified as events.
- Most V5 driver API calls remain, the difference being that a port ID is used instead of the V5 port index and a card ID instead of the V5 switch index.

There is a V5 driver backwards compatibility mode that implements the V5 driver API in terms of the V6 driver API. For further details, please refer to the V5 to V6 migration guide.

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 routines in the Aculab Call Control library, `idle_net_ts` and `port_init`. These routines are not formally part of the switch API but are intimately related to it.

Aculab Digital Access cards may have one or more network ports allowing attachment 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 Digital Access cards can be integrated. Aculab cards currently support the following expansion buses:

- H.100 bus (also known as CT Bus or H-Bus) – PCI cards only
- H.110 bus (also known as H-Bus) – compact PCI cards only
- MVIP bus – not supported on Prosody X cards

- SC Bus – not supported on Prosody X cards

Aculab Digital Access cards can be interconnected or connected to resource cards, such as an Aculab Prosody Speech Processing card, via an expansion bus. All cards must however reside in the same chassis.

Each Aculab Digital Access 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 Prosody or 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 Primary Rate Digital Access cards can switch data between timeslots on network ports, on-card resources, and associated expansion buses, for example, H.100 or H.110 (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 Digital Access 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 a Digital Access 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.

Expansion buses

1.3.1 H.100 bus

The H.100, also known 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.

Digital Access 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 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 #0 is outputting data to timeslot 6 of stream D0, then another card #1 may not output onto the same timeslot on the same stream. If this actually occurs (in error) it is called 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 other 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. Refer to the call, switch and speech driver installation guide for details on how this is done. An example is also provided in section 5.2.1.

1.3.2 H.110 bus

The H.110 bus is used with compact PCI cards only. It has the same capacity and stream arrangement as the H.100 bus but has additional clock fallback features and has a different bus termination mechanism. From a switch control API perspective it is treated identically to the H.100 Bus except where specifically noted.

Bus termination on H.110 bus does not require any configuration flags or jumpers to be set up; it is all handled automatically by the driver at run time.

1.3.3 MVIP bus

(Not applicable to Prosody X PCI cards)

The MVIP bus is a time division multiplexed bus consisting of 16 streams, each stream able to carry 32 unidirectional 64Kbps speech paths (termed timeslots).

The 16 streams are named DSo0, DSo1, ... , DSo7, DSi0, DSi1, ..., DSi7 and the timeslots within these streams are identified using numbers 0 to 31.

Digital Access cards attached to the MVIP bus may switch data onto the MVIP bus from network port timeslots and on-card resources, and vice versa.

By convention the MVIP streams DSi0 to DSi7 are used to carry data from the network to resource (e.g. speech processing) cards, and the MVIP streams DSo0 to DSo7 are used to carry data from resource cards to the network. If, for a particular telephone call, a duplex connection were required to a resource card, the same timeslot number would typically be used on both a DSoN and a DSiN stream. For example:

- DSi0 timeslot 0 - Data from Digital Access card network port to resource card
- DSo0 timeslot 0 - Data from resource card to Digital Access card network port

The switch driver API reflects this convention in its numbering of MVIP streams.

When connecting together phone calls occurring on two different Digital access cards, an application must use its own convention as to how DSo and DSi streams are used.

In all cases, it is the responsibility of the application to manage the allocation and use of MVIP streams and timeslots.

No more than one card at a time may output data onto a particular MVIP bus stream timeslot. For example, if a card #0 is outputting data to timeslot 6 of stream DSi0, then another card #1 may not output onto the same timeslot on the same stream (this error situation if it occurs is called bus contention).

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

In order for data to be switched successfully between cards on the MVIP bus, exactly one card must be set up to be the MVIP bus clock master, and all other cards must clock slave off the MVIP bus.

The first and last cards on an MVIP bus ribbon cable should have MVIP bus terminations enabled. Refer to specific card installation guide for details on how this is done for particular card. If terminations are software rather than jumper enabled, see [section 5.2.1](#) for the correct configuration flag setting.

Refer to a specific card installation guide for information on how to set up PCI cards in legacy bus modes such as MVIP.

1.3.4 SC bus

(Not applicable to Prosody X PCI cards)

The SC Bus is a time division multiplexed bus seen through the switch driver API as a single stream able to carry up to 1024 unidirectional 64Kbps speech paths (termed timeslots).

Timeslots on the SC Bus stream are numbered 0...1023.

Unlike the MVIP bus, there is no convention on which SC Bus timeslots should be used to carry data from the network to resource cards and vice versa. No two cards should assert the same SC Bus timeslot simultaneously (this error situation, if it occurs, is called bus contention).

In order for data to be switched successfully between cards on the SC Bus, exactly one card must be set up to be the SC Bus clock master, and all other cards must clock slave off the MVIP bus.

The SC Bus is never terminated so no termination configuration switches or jumpers are required. Refer to specific card installation guide for information on how to set up PCI cards in legacy bus modes such as SC Bus.

1.3.5 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.

Note 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. This would not work if tried using a single card, or would result in expansion bus contention if tried with multiple cards.

Note 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.

All applications built to use Aculab drivers Applications using the switch driver must include the mentioned 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.

Note 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 2000/XP

The `ACU_INT` type defined in “`acu_type.h`” must be a 32-bit value so that size of `ACU_INT` is 4.

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

```
SW_LIB.DLL
```

2.2 Opening cards for use with the switch driver

Before an application can make use of the Switch API for a card, the card must 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
<code>sw_ver_switch()</code>	Determines the version of a given switch driver.
<code>sw_component_version()</code>	Determines the version of a given component.
<code>sw_mode_switch()</code>	Determines which expansion buses a card may switch data onto/off-from.
<code>sw_card_info()</code>	Returns card information for the card whose switching is controlled by the switch driver <code>card_id</code> .
<code>sw_set_card_notification_queue()</code>	Used to register a queue that will receive switch notification events (such as clock fallback events) for the specified card.
<code>sw_get_card_notification()</code>	The switch driver queues events such as clock fallbacks. This function is used to collect those events.
<code>sw_get_card_notification_wait_object()</code>	This function retrieves an operating system specific wait event that can be used to wait for switch driver events.
<code>sw_config_timeslot_companing()</code>	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.
<code>sw_query_timeslot_companing()</code>	Query A-law/mu-law to mu-law/A-law conversion, determines the companing conversion settings of a given card timeslot.
<code>sw_config_companing()</code>	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.
<code>sw_query_companing()</code>	Query A-law/mu-law to mu-law/A-law conversion, determines the companing conversion settings of a given card.
<code>sw_get_dsp_stream_info()</code>	Maps DSP position and port indexes to TDM streams and timeslots.
<code>sw_set_card_h100_termination()</code>	Enables or disables the H-Bus bus termination on a PCI card.
<code>sw_set_output()</code>	Make a connection; break a connection or output pattern on timeslot.

Switch API Call	Description
<code>sw_query_output()</code>	Determine the source of switch connection.
<code>sw_switch_override_mode()</code>	Control driver behaviour when an attempt is made to make a connection that would break an existing connection.
<code>sw_sample_input()</code>	Obtain 8-bit sample from given timeslot.
<code>sw_sample_input0()</code>	Faster version of above call.
<code>sw_reset_switch()</code>	Reset card switch devices to disable all current connections.
<code>sw_tristate_switch()</code>	Tri-state card from expansion bus.
<code>sw_clock_control()</code>	Change clock generation circuit reference source and/or expansion bus clock master/slave mode.
<code>sw_query_clock_control()</code>	Determine last clock mode set for card and any special clock mode to be used during system initialisation.
<code>sw_h100_config_board_clock()</code>	Change clock generation circuit reference source and/or H-Bus bus clock master/slave mode.
<code>sw_h100_config_netref_clock()</code>	Configure fallback reference clock for H-Bus bus.
<code>sw_h100_query_board_clock()</code>	Determine last H-Bus clock mode set for card and status of H-Bus bus clocks.
<code>sw_h100_query_netref_clock()</code>	Determine fallback reference clock for H-Bus bus.
<code>sw_track_api_calls()</code>	For applications running on multi-tasking operating systems, this function may be used to track API calls made to a switch driver.
<code>sw_abort_api_calls()</code>	Abort pending API calls.
<code>sw_query_switch_caps()</code>	Function for compatibility with application code generated to use MVIP 90 API.

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

Call API routine	Description
<code>port_init()</code>	Used to write an idle pattern onto all timeslots on a port. For CAS protocols will also connect each timeslot to the signalling DSP.
<code>idle_net_ts()</code>	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:

Quality	Character
'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 <i>card_id</i>
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
<code>SWMODE_CTBUS_MVIP</code>	MVIP bus
<code>SWMODE_CTBUS_SCBUS</code>	SC Bus
<code>SWMODE_CTBUS_H100</code>	H.100/H.110 Bus (H-Bus)
<code>SWMODE_CTBUS_H100_TERM</code>	H.100/H.110 Bus (H-Bus) with terminated bus (Prosody X PCI only)
<code>SWMODE_CTBUS_NONE</code>	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:

<code>ERR_SW_INVALID_SWITCH</code>	no switch driver corresponding to <code>card_id</code>
<code>ERR_SW_DEVICE_ERROR</code>	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_STRUCTURE`

Return values

card_type

This parameter indicates the type of card:

card type value	Description
SW_PROSODY_PCI_CARD	Prosody PCI card – speech processing plus optional digital access
SW_E1_T1_PCI_TRUNK_CARD SW_E1_T1_PCI_CARD	PCI E1/T1 digital access card – no speech processing
SW_PROSODY_CPCI_CARD	Compact PCI version of Prosody PCI card
SW_E1_T1_CPCI_CARD	Compact PCI version of PCI trunk card (up to 8 ports)
SW_VOIP_PCI_H323_GATEWAY_CARD	Voice over IP PCI card
SW_E1_T1_CPCI_PMX_CARRIER_CARD	Compact PCI 16 port trunk card
SW_PROSODY_X_PCI_CARD	Prosody X PCI card
SW_PROSODY_X_CPCI_CARD	Prosody X cPCI card
SW_PROSODY_X_PCIE_CARD	Prosody X PCIe card

card_present

This parameter is set to 1 for all cards other than compact PCI cards. For compact PCI cards it is set to 1 if the card is inserted and operational.

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 (Invalid for Prosody X, returned as zero)

This parameter indicates the physical address the cards switching resources are mapped to.

io_address_or_pcidev (Invalid for the Prosody X, returned as zero)

This parameter indicates the PCI device number (PCI cards).

physical_irq (Invalid for the Prosody X, returned as zero)

Represents the computer system interrupt number, which is allocated to the card in the PCI slot.

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:

<code>ERR_SW_INVALID_SWITCH</code>	no switch driver corresponding to <i>card_id</i>
<code>ERR_SW_DEVICE_ERROR</code>	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](#)).

Note 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 value 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

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

Event

This 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_NETREF1/2

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_PARAMS* wo_params)

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

The `sw_get_notification_wait_object()` function takes a pointer `wo_params`, to a structure `SW_WAIT_OBJECT_PARAMS`. 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

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_PARAMS event_params;
SW_WAIT_OBJECT_PARAMS wo_params;
ACU_ERR error;

INIT_ACU_STRUCT(&wo_params);
INIT_ACU_STRUCT(&event_params);

error = sw_get_notification_wait_object(card_id, &wo_params);

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

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

Specific functions for PCI and PCIe cards

3.7 `sw_set_card_h100_termination()` – Set up H.100 termination

Enables or disables the H-Bus termination on PCI and PCIe cards.

Synopsis

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

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`

h100termination

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

<i>h100termination</i>	
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.

Specific functions for cards fitted with PMX modules

The following two functions apply to any product using the PMX card, for example, the 16-port E1/T1 cPCI trunk card.

3.8 `sw_config_timeslot_companding()` – Set up A-law/mu-law conversion

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

Note By default the 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 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, `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 3.14.1 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 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:

<code>COMPANDING_DISABLED</code>	0
<code>COMPANDING_A_TO_MU_LAW</code>	2
<code>COMPANDING_MU_TO_A_LAW</code>	3

In addition, `rx_mode` may be set to one of the following values:

<code>COMPANDING_T1_IDLE</code>	6
<code>COMPANDING_E1_IDLE</code>	7
<code>COMPANDING_E1_SILENCE</code>	8
<code>COMPANDING_T1_SILENCE</code>	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()` – Set up A-law/mu-law conversion

This function is used to select A-law to mu-law or mu-law to A-law conversion on network port streams.

Note 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_STRUCTURE`

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:

<code>COMPANDING_DISABLED</code>	0
<code>COMPANDING_A_TO_MU_LAW</code>	2
<code>COMPANDING_MU_TO_A_LAW</code>	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 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:

<code>COMPANDING_DISABLED</code>	0
<code>COMPANDING_A_TO_MU_LAW</code>	2
<code>COMPANDING_MU_TO_A_LAW</code>	3

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

Prosody X specific functions

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 the `t8110.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:

Quality	Character
'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:

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

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
<code>kSW_DSP_TYPE_SIGNALLING</code>	DSP used for CAS or SS7 signalling such as DSP65 or PMX 8101/3
<code>kSW_DSP_TYPE_PROSODY</code>	DSP used for Prosody such as SHARC or 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:

<code>ERR_SW_NO_SUCH_DSP</code>	DSP of the given type is not fitted to the indicated position
<code>ERR_SW_NO_SUCH_DSP_PORT</code>	indicated serial port does not exist on DSP
<code>ERR_SW_INVALID_PARAMETER</code>	invalid <code>dsp_type</code> 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

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, struct 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) for all Aculab PCI and cPCI card types.

In general, the only limitations associated with distributed or local switching are those imposed by the switching capacities of the switch devices used. There are some special switching restrictions that apply only to distributed switching on the E1/T1 cPCI 16-port and the Prosody X cards.

3.15.1 Switching restrictions for E1/T1 cPCI 16-port and Prosody X cards:

Note The switching device on this card does not allow a given H.100/H.110 (H-bus) timeslot to be used as an input in one connection and the output in another connection simultaneously. For example, it is not possible to make a pair of connections such as `0,0<-1,0` and `1,0<-2,0`, as stream 1 timeslot 0 appears as an input in the first connection and as an output in the second connection.

However, to maintain compatibility with existing applications, the switch driver will allow a pair of connections from local stream timeslots to be "looped back" on an H-bus timeslot.

For example, it is possible to make a pair of connections such as `33,1<-0,0` and `0,0<-32,1`. The switch driver maps these API connections to physical connections `33,1<-32,1` and `0,0<-32,1`. This remapping of H-bus loopback connections occurs regardless of the order in which the 2 API connections are made. If timeslot `33,1` is subsequently queried with `sw_query_output()`, the connection will be reported as `33,1<-0,0`.

If the input side of an H-bus loopback connection is changed, the driver will make appropriate changes to the physical connections. Continuing the above example, if the connection `0,0<-32,1` is overwritten with connection `0,0<-48,1`, the switch driver will replace physical connections `33,1<-32,1` and `0,0<-32,1` with connections `33,1<-48,1` and `0,0<-48,1`.

This type of behaviour extends to the case where a loopback connection has 2 or more local outputs. For example API connections `34,1<-0,0 33,1<-0,0 0,0<-32,1` will be mapped to physical connections `34,1<-32,1 33,1<-32,1 0,0<-32,1`. Overwriting `0,0<-32,1` with `0,0<-48,1` will cause the input of all 3 physical connections to be changed from `32,1` to `48,1`.

The switching device fitted on this card does not allow a given H-bus timeslot to output a pattern and to be used as input to a switch connection simultaneously.

However, it is possible to use the `sw_config_timeslot_companing()` API call to output an idle or silence pattern on a PMX receive timeslot, which may then be connected to an H-Bus timeslot that is configured as an input (this uses the remapping mechanism described above).

There is an additional limitation on the Prosody X cPCI card where it is not possible to make direct connections between the two DSP farms (streams 64 to 71 and streams 80 to 87). It is possible to make indirect connections between the two DSP farms via the H.110 bus.

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`

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

<i>ost</i>	output stream number
<i>ots</i>	output timeslot number
<i>ist</i>	input stream number
<i>its</i>	input timeslot number
<i>pattern</i>	the value to be written to the output timeslot

mode

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.

Note **For E1/T1 cPCI 16-port and Prosody X cards:**

`ERR_SW_NO_PATH` indicates a problem was encountered making the requested connection. Below are five examples of `sw_set_output()` sequences that will produce this return value.

1. `32,1<-0,1` then `0,1<-1,1`. Cannot configure `0,1` as an output with an H-bus input while it is operating as an input.
If the input to `0,1` were not an H-bus timeslot, this error would not occur.
2. `32,1<-0,1` then output a pattern on `0,1`. Cannot output a pattern on `0,1` while it is operating as an input.
3. `0,1<-1,1` then `32,1<-0,1`. Cannot configure `0,1` as an input while it is operating as an output with an H-bus input.
If the input to `0,1` were not an H-bus timeslot, this error would not occur.
4. Output a pattern on `0,1` then `32,1<-0,1`. Cannot configure `0,1` as an input while it is operating as an output.
5. `80,0<-64,0`. Prosody X cPCI only. Cannot make a direct connection between DSP farm 0 and DSP farm 1.

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, struct 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:

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

3.17 `sw_switch_override_mode()` - Set override mode

This call is only applicable to ISA cards. The call is used to change the mode of the switch driver to either blocking or override.

Synopsis

```
int sw_switch_override_mode( ACU_CARD_ID card_id, int mode);
```

Description

Some Aculab Digital Access cards have switching restrictions, for example, when making a connection to a timeslot on one stream of the MVIP expansion bus would break a connection made to a timeslot on another stream of the MVIP expansion bus.

The switch driver can operate in two modes:

- Override mode in which the driver simply breaks the previous connection that would potentially block the new connection to be made.
- Blocking mode where the switch driver returns an error, `ERR_SW_PATH_BLOCKED`, whenever an application attempts to make such a connection through a call to `sw_set_output()`.

By default (and for backward compatibility with previous versions) the switch driver operates in override mode.

Input parameters

card_id

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

mode

0 - will cause the driver to operate in blocking mode

1 - will cause the driver to operate in override mode.

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:

<code>ERR_SW_INVALID_SWITCH</code>	no switch driver corresponding to <i>card_id</i>
<code>ERR_SW_INVALID_MODE</code>	invalid mode specified
<code>ERR_SW_DEVICE_ERROR</code>	no switch drivers installed, switch driver initialisation failed or device I/O error occurred

3.18 `sw_sample_input()`, (`sw_sample_input0()`) - Sample timeslot

This function samples an octet from the indicated timeslot. Two variants are provided, `sw_sample_input()`, which delays so that the sample will be valid following any recent switching, and `sw_sample_input0()`, which does not perform this delay.

Note 16 port E1/T1 cPCI card only. With this type of card it is not possible to sample from an H.110 timeslot that is being used as an output. For example, if a connection 0,0<-1,0 exists, it is not possible to sample from stream 0 timeslot 0.

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_STRUCTURE`

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:

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

3.19 `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 (tristate)
```

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:

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

3.20 `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. 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.

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

Clock Control Functions

3.21 `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.100 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 expansion bus specific 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:

<code>ERR_SW_INVALID_SWITCH</code>	no switch driver corresponding to <i>card_id</i>
<code>ERR_SW_INVALID_CLOCK_PARM</code>	clock mode invalid for card or card mode
<code>ERR_SW_OTHER_SCBUS_CLOCK</code>	modifier <code>DRIVE_SCBUS</code> was specified and another card is already driving the SCbus clock
<code>ERR_SW_DEVICE_ERROR</code>	no switch drivers installed, switch driver initialisation failed or device I/O error occurred

3.22 `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:

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

3.23 sw_h100_config_board_clock() - Set up H.100 (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

and for H.110 cards only:

Clock Source	Value
H100_SOURCE_NETREF_1	10
H100_SOURCE_NETREF_2	11

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

and for H.110 cards only:

Auto fallback Qualifier	Value
H100_CHANGEOVER_TO_NETREF_2	128

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.

For H.110 cards, if *auto_fall_back* is qualified with `H100_CHANGEOVER_TO_NETREF_2`, then the alternative `CT_NETREF_2` signal will be used instead.

If *auto_fall_back* is qualified by `H100_AUTO_RETURN`, then when the original reference source once more becomes available, it will be reverted back.

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 (for H.110 cards only `H100_CHANGEOVER_TO_NETREF_2` indicates that the reference source `CT_NETREF_2` is to be used). Specifying both qualifiers indicates the network port is to be used and that fallback to `CT_NETREF/CT_NETREF_2` 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` (and for H.110 cards the `CT_NETREF_2`) fallback clock line is running at, and should be set to one of the following values:

NETREF Clock Speed	Value
<code>H100_NETREF_8KHZ</code> (8 kHz)	0
<code>H100_NETREF_1544MHZ</code> (1.544MHz)	1
<code>H100_NETREF_2048MHZ</code> (2.048MHz)	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:

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

3.24 sw_h100_config_netref_clock() - Set up H.100 (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` or `H100_ENABLE_NETREF_2`, 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` or `CT_NETREF_2` clock signal.

netref_clock_mode

NETREF clock mode	Value
<code>H100_DISABLE_NETREF</code>	0
<code>H100_ENABLE_NETREF</code>	1
<code>H100_ENABLE_NETREF_2</code>	2

For H.110 cards only, the parameter `netref_clock_mode` may be set to `H100_ENABLE_NETREF_2` in order to drive the alternative `CT_NETREF_2` clock signal.

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
<code>H100_NETREF_8KHZ</code> (8 kHz)	0
<code>H100_NETREF_1544MHZ</code> (1.544MHz)	1
<code>H100_NETREF_2048MHZ</code> (2.048MHz)	2

The reference clock source for the generated `CT_NETREF/CT_NETREF_2` will be the network port indicated by the `network` parameter.

The type (E1 or T1) of network port selected as the `CT_NETREF` reference clock source determines the possible speeds that `CT_NETREF` may be driven at. If the network parameter indicates an E1 port, then `netref` may be driven at 8KHz or 2.048MHz, if the network parameter indicates a T1 port, then `netref` may be driven at 8KHz or 1.544MHz,

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:

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

Note For cards fitted with PMX modules, `call_restart_fmw()` must be called before `sw_h100_config.netref_clock()`.

3.25 sw_h100_query_board_clock() - Query H.100 (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()`.

Note 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

For H110 cards only, the output parameter `netref_b_clock_status` will always be set to one of the above values to reflect the state of the `CT_NETREF_2` signal.

Note The availability of clock status information depends on the current card clock configuration:

1. Primary master configured cards are unable to determine H-Bus clock status and will return the value `H100_CLOCK_STATUS_UNKNOWN`
2. Secondary master configured cards can determine status of H-Bus primary clocks but only if H.100 fallback is enabled.
3. Slave configured cards can determine status of primary and secondary clocks but only if H-Bus fallback is enabled.
4. `CT_NETREF` status may currently only be determined by secondary master and slave cards with H.100 fallback enabled. These cards can only determine the `CT_NETREF` status when the `NETREF` clock frequency is 8kHz.
5. Status of Primary master card clock source may be determined by value in `fall_back_occurred`, if this is set non-zero primary master clocks are currently been driven from fallback (`CT-NETREF`) 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:

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

3.26 `sw_h100_query_netref_clock()` - Query H.100 (H-Bus) fallback clock

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

typedef struct 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/CT_NETREF_2` generation, values for these output parameters are as described for the `sw_h100_config_netref_clock()` API call.

Note For Prosody X cards: after a card has been configured to generate the H.100 NETREF signal from one of its ports, the H.100 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 NETREF signal will not be generated. The returned `netref_clock_mode` allows an application to determine if NETREF is currently being generated. If a card has previously been configured to generate NETREF from a network port and 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:

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

Diagnostic and trace functions

3.27 `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.

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_STRUCTURE`

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.

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:

<code>ERR_SW_DEVICE_ERROR</code>	no switch drivers installed, switch driver initialisation failed or device I/O error occurred
----------------------------------	---

MVIP 90 compatibility functions

3.28 `sw_query_switch_caps()` - Get switch capabilities

This function returns information about the MVIP switch matrix. The call may be made at any time and may be made before `reset_switch` if required.

Synopsis

```
int sw_query_switch_caps (ACU_CARD_ID card_id, struct capabilities_parms*
                          capabilitiesp);

typedef struct capabilities_parms
{
    ACU_INT      size;           /* IN */
    ACU_INT      revision;      /* OUT */
    ACU_INT      domain;       /* OUT */
    ACU_INT      routing;      /* OUT */
    ACU_INT      blocking;     /* OUT */
    ACU_INT      networks;     /* OUT */
    ACU_INT      channels[8];  /* OUT */
} CAP_PARMS;
```

The `sw_query_switch_caps()` function takes a pointer `capabilitiesp`, to a structure `CAP_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

revision

Contains the revision level of the device driver multiplied by 100. For example revision 1.23 will be returned as 123.

domain

The return value *domain*, is a bit pattern indicating which of the MVIP streams have full duplex connections to the MVIP switch matrix.

```
Bit 0 indicates if a forward connection DSo0 and DSi0 is supported.
Bit 1 indicates if a forward connection DSo1 and DSi1 is supported.
Bit 8 indicates if a reverse connection DSi0 and DSo0 is supported.
Bit 9 indicates if a reverse connection DSi1 and DSo1 is supported and so
on.
```

The *domain* of a compliant switch (by definition) is 1111,1111,1111,1111.

routing

The return value *routing* is a bit pattern indicating MVIP switch matrix half-duplex routing capabilities.

```
Bit 0 indicates MVIP DSo to MVIP DSi capability
Bit 1 indicates MVIP DSi to MVIP DSo capability
Bit 2 indicates MVIP DSo to MVIP DSo capability
Bit 3 indicates MVIP DSi to MVIP DSi capability
Bit 4 indicates MVIP DSo to NETWORK capability
Bit 5 indicates NETWORK to MVIP DSi capability
Bit 6 indicates MVIP DSi to NETWORK capability
Bit 7 indicates NETWORK to MVIP DSo capability
Bit 8 indicates NETWORK to NETWORK capability
```

blocking

The return value *blocking*, is a bit pattern indicating that a switch is unable to provide all possible connections on a stream. It should be noted that the *blocking* bits are only valid if the corresponding *routing* bits are set. By ANDing the *routing* and *blocking* fields one can determine the blocking limitations of the switch.

```

Bit 0 indicates MVIP DSo to MVIP DSi blocking
Bit 1 indicates MVIP DSi to MVIP DSo blocking
Bit 2 indicates MVIP DSo to MVIP DSo blocking
Bit 3 indicates MVIP DSi to MVIP DSi blocking
Bit 4 indicates MVIP DSo to NETWORK blocking
Bit 5 indicates NETWORK to MVIP DSi blocking
Bit 6 indicates MVIP DSi to NETWORK blocking
Bit 7 indicates NETWORK to MVIP DSo blocking
Bit 8 indicates NETWORK to NETWORK blocking

```

networks

The return value *networks*, contains the number of *NETWORK* connections, i.e. the number of streams connected to the *NETWORK* devices including streams used for signalling.

channels

The return value *channels* array, gives the number of channels of each *NETWORK* stream specified by the *networks* field.

The Aculab MVIP E1/G703 card will return the following capabilities structure.

```

size           operating system dependent
revision       device driver dependent
domain        1111,1111,1111,1111
routing       1,1111,1111
blocking      xxxx,xxxx,xxxx,xxxx
networks      8
channels[]    networks[0]   =   30
              networks[1]   =    1
              networks[2]   =   30
              networks[3]   =    1
              networks[4]   =   30
              networks[5]   =    1
              networks[6]   =   30
              networks[7]   =    1

```

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

```

4 Switching issues

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 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 call `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.

Note In an application that needs to support hot-swap, any nail-up operations should be performed on a per-card basis. When a card is removed from the system, any switch connections on it will be broken. The application will need to recreate the switch connections on any replacement card.

When implementing an on-demand switching strategy, applications must have the some 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/H.110 (H-Bus) switching

Aculab PCI/cPCI 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 Digital access 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);
```

4.3 MVIP bus switching

Note Not applicable to Prosody X or cPCI cards

In order to switch data from digital access cards to/from the MVIP bus, connections must be established through invocations of the switch control API call `sw_set_output()`. The parameter block for this API call should be set up for `CONNECT_MODE` and the `ist`, `its`, `ost` and `ots` parameters set to the appropriate stream and timeslot numbers for the connection input and output.

The numbering convention used for MVIP streams reflects the convention of using DSi streams for data switched from network cards to resource cards, and DSo streams for data going in the opposite direction.

When used to specify an output onto the MVIP bus (`ost` parameter for `sw_set_output()`), streams 0 to 7 correspond to the MVIP bus streams DSi0 to DSi7. When used to specify an input from the MVIP bus (`ist` parameter for `sw_set_output()`), streams 0 to 7 correspond to MVIP bus streams DSo0 to DSo7.

When used to specify an output onto the MVIP bus (`ost` parameter for `sw_set_output()`), streams 8 to 15 correspond to the MVIP bus streams DSo0 to DSo7. When used to specify an input from the MVIP bus (`ist` parameter for `sw_set_output()`), streams 8 to 15 correspond to MVIP bus streams DSi0 to DSi7.

The following table summarises the MVIP stream numbers to be used for the `ist` and `ost` parameters of the switch control API `OUTPUT_PARMS` structure:

	0..7	8..15
<i>ist</i>	DSo0...DSo7	DSi0...DSi7
<i>ost</i>	DSi0...DSi7	DSo0...DSo7

4.3.1 Switching between network cards and resource cards

The following code fragment shows how two connections may be made to make a full duplex bi-directional connection between a network port timeslot and a resource card on the MVIP bus.

Note an application would normally invoke `call_details` from the call control API [1] to determine which network port stream and timeslot a phone call came in on.

```

DETAIL_XPARMS      call_details;
OUTPUT_PARMS      connection;
ACU_CARD_ID       switch_driver;
ACU_CARD_ID       call_port;
/*
 * invoke call details.
 */

...
/* Make connection from network port timeslot to DSi0 timeslot 0 (MVIP
 * bus timeslot chosen by this application to route data to resource
 * card for this call).*/

connection.ist     = call_details.stream;
connection.its     = call_details.ts;
connection.mode    = CONNECT_MODE;
connection.ost     = 0;
connection.ots     = 0;

call_port          = call_handle_2_port(call_details.handle);
switch_driver      = call_port_2_swdrvvr(call_port);

sw_set_output(switch_driver,&connection);

```

```

/* Make connection from DSo0 timeslot 0 (MVIP bus timeslot chosen
 * by this application to route data from resource card for this call)
 * to network port timeslot.*/

connection.ist      = 0;
connection.its      = 0;
connection.mode     = CONNECT_MODE;
connection.ost      = call_details.stream;
connection.ots      = call_details.ts;

sw_set_output(switch_driver,&connection);

/* Make any calls to API for resource card to tell it to input data from
 * DSi0 timeslot 0 and to output data to DSo0 timeslot 0.*/

...

```

Note the same stream number (0) is used to denote MVIP stream DSo0 when specified for the *ist* parameter and MVIP stream DSi0 when specified for the *ost* parameter.

4.3.2 Switching between multiple network cards

The following code fragment shows how four connections may be made to make a full duplex bi-directional connection over the MVIP bus between two network port timeslots on two different Digital access cards (card 0 and card 1).

Note an application would normally invoke `call_details` from the call control API to determine which network port stream and timeslot a phone call came in on.

```

DETAIL_XPARMS      call_details0;
DETAIL_XPARMS      call_details1;
OUTPUT_PARMS       connection;
ACU_CARD_ID        switch_driver0;
ACU_CARD_ID        call_port0;
ACU_CARD_ID        switch_driver1;
ACU_CARD_ID        call_port1;

/* invoke call details for card 0 (call_details0)
 * and card 1 (call_details1).*/

call_port0         = call_handle_2_port(call_details0.handle);
switch_driver0     = call_port_2_card_id(call_port0);

call_port1         = call_handle_2_port(call_details1.handle);
switch_driver1     = call_port_2_card_id(call_port1);

if (switch_driver0 != switch_driver1)
{
/* Calls terminated on different cards, we will have to use MVIP
 * bus to connect them */
/* Make connection from card 0 net port timeslot to DSi0 timeslot 0
 * (MVIP bus timeslot chosen by this application to route data from
 * card 0 to card 1 for this call)*/

connection.ist      = call_details0.stream;
connection.its      = call_details0.ts;
connection.mode     = CONNECT_MODE;
connection.ost      = 0;
connection.ots      = 0;

sw_set_output(switch_driver0,&connection);

```

```
/* Make connection from DSo0 timeslot 0 (MVIP bus timeslot chosen by
 * this application to route data from card 1 to card 0 for this
 * call) to network port timeslot */

connection.ist          = 0;
connection.its          = 0;
connection.mode         = CONNECT_MODE;
connection.ost          = call_details0.stream;
connection.ots          = call_details0.ts;

sw_set_output(switch_driver0,&connection);

/* Make connection from card 1 net port timeslot to DSo0 timeslot 0
 * (MVIP bus timeslot chosen by this application to route data from
 * card 1 to card 0 for this call)*/

connection.ist          = call_details1.stream;
connection.its          = call_details1.ts;
connection.mode         = CONNECT_MODE;
connection.ost          = 8+0;
connection.ots          = 0;

sw_set_output(switch_driver1,&connection);

/* Make connection from DSi0 timeslot 0 (MVIP bus timeslot chosen by
 * this application to route data from card 0 to card 1 for this
 * call) to network port timeslot */

connection.ist          = 8+0;
connection.its          = 0;
connection.mode         = CONNECT_MODE;
connection.ost          = call_details1.stream;
connection.ots          = call_details1.ts;

sw_set_output(switch_driver1,&connection);
}
```

Note The use of stream 8 for the card #1 for the *ost* and *ist* parameters in order to specify a connection with a DSo stream output and a DSi stream input. This is sometimes referred to as using the MVIP bus streams in the reverse direction.

5 System clocking issues

5.1 Clocking of digital access 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 via configuration of expansion bus clocking.

Current Aculab Digital Access cards may be integrated into one or more of the following expansion bus types:

- H.100 – PCI cards only
- H.110 – compact PCI cards only
- MVIP – PCI card with legacy adapter only *
- SC Bus – PCI card with legacy adapter only *

* not applicable to Prosody X PCI cards

Each of these bus types 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 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
- Another expansion bus or a secondary clock on a expansion bus

Each type of expansion bus has its own set of clocking features and its own terminology for the expansion bus clocks. A brief overview of each expansion bus clocking mechanism follows:

5.1.1 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 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 bus '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);
```

An alternative method to set up such a simple system clocking scenario would be to use the MVIP-90 compatible switch driver API clock control calls as follows:

```
/* In H.100 mode sw_clock_control always implicitly sets up 'A' clocks */

sw_clock_control(card_id_a, CLOCK_REF_NET1);
sw_clock_control(card_id_b, CLOCK_REF_H100);
```

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 2.048 MHz. The rate it is driven at is immaterial but all the cards on the H.100 bus must be configured to use it at a single given rate. If `CT_NETREF` is driven from a E1 network port it can be driven at 8KHz or 2.048 MHz, if `CT_NETREF` is driven from a T1 network port it can be driven at 8KHz or 1.544 MHz.

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

Note 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_NETFREF.
     */
}

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, and 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 recovered from it, this is done by ORing an auto-return bit to enable this feature when initially setting up the primary clock master:

```

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);

```

If auto-return is not enabled, 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 previously mentioned, 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 for use through 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 the secondary clocks will be timing locked to the primary master clocks, if however 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 said to be promoted to being 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 with any new secondary clock master having to 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 being 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);

```

Note It will take up to 1 second for secondary B clocks to become phase aligned with A clocks following a call to `sw_h100_config_board_clock()` for card 1.

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 becoming 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 changeover to becoming 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);

```

Now if it is required that a third H.100 clock slave card should automatically switch over to 'B' clocks if the primary clock master card fails, then the slave card clocking should 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);

```

Note The `auto_fall_back` parameter for a card configured to operate in `H100_SLAVE` mode is used to enable clock slave swap over from 'A' clocks to 'B' clocks (and 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 can have its reference source pre-configured for use when it becomes promoted to primary clock master. If this reference source is a trunk attached to a network port, and if when promoted, the card is required to fallback to `CT_NETREF` on trunk failure, then the secondary clock master should be configured as follows:

```

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 and cause bus contention
- discontinuities of clocks on cards with DSP devices do not occur causing DSPs to lose synchronization with card

Such a changeover can be achieved by use of primary and secondary clocks, and appropriate secondary master clock fallback programming, for example as follows:

- set up card that will take over as clock master (needs a net port) as secondary clock master (say B clocks), slaving from primary clocks (say 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, secondary master will at this point automatically fallback to change its clock source to being network port (because A clocks will have disappeared) and become new primary master of H.100 bus (using B clocks)

Note that if instead of using secondary master fallback to change new master card clock reference, the clock reference was changed using a `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 matter if it did not have any connections made to from bus, but if it did, then its desynchronized state could cause H.100 bus contention to occur until such time its clock reference was changed to B clocks slave.

5.1.2 H.110 bus clocking

H.110 Clocking for compact PCI cards is for the most part identical to that for H.100 clocking of PCI cards. The main difference is that the H.110 card has an additional fallback clock signal `CT_NETREF_2`.

One application of this additional signal could be to allow a fall back clock presence to be maintained while the card that was previously generating the fall back clock on the bus is being replaced. Thus if card was H.110 bus A clocks master (with fallback to `CT_NETREF`) and card X generating `CT_NETREF` needed replacing, card Y could be configured to generate `CT_NETREF_2`, card A re-configured to fallback to `CT_NETREF_2`, and then card X could be removed from the system.

Another application for `CT_NETREF_2` might be to provide redundancy for the case that a hardware fault on a card prevented the normal `CT_NETREF` signal being reliably driven.

5.1.3 MVIP bus clocking

Note Not applicable to Prosody X PCI cards

The MVIP bus has two sets of clock signals used for synchronising data transfer across the bus and distributing network timing among multiple network cards, these are the main

MVIP clocks, and the SEC8K secondary trunk clocks.

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 MVIP bus during system initialisation and later by the application API calls.

Initial MVIP bus clocking will be set up during switch driver initialisation, (see 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 MVIP clock master and drive MVIP bus clocks, and all other cards would be designated MVIP bus clock slaves and take their timing from the MVIP bus clocks.

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

bus clock slave synchronized to MVIP bus clocks:

```
sw_clock_control(card_id_a, CLOCK_REF_NET1);
sw_clock_control(card_id_b, CLOCK_REF_MVIP);
```

MVIP bus clock reference changeover

Sometimes (for example in the case of trunk failure) it is necessary to change the clock reference for a system from one network port to another, or from the local oscillator on one card to one on another.

As previously noted in the previous section, some resource cards, especially DSP carrier cards can be disrupted by any discontinuity occurring in the expansion bus clock signal if they are clock slaving off this expansion bus.

If the clock reference is simply changed from one source to another source on the expansion bus clock mastering card, for example:

```
sw_clock_control(card_id_a, CLOCK_REF_NET1); /* Initial clock reference */

/* application determines clock changeover required */

sw_clock_control(card_id_a, CLOCK_REF_NET2); /* New clock reference */
```

Then no discontinuity will occur because of the Phase Locked Loop (PLL) device with which the card is equipped.

If, however the clock reference must change to a source on a card that is not the expansion bus clock master, then the method of setting the current clock mastering card to be a clock slave, and then setting the second card to be expansion bus clock master, would cause a discontinuity in the expansion bus clock signal during the clock change over period. For example:

```
sw_clock_control(card_id_a, CLOCK_REF_NET1); /* Initial clock reference */
sw_clock_control(card_id_b, CLOCK_REF_MVIP); /* Initial clock reference */

/* application determines clock changeover required */

sw_clock_control(card_id_a, CLOCK_REF_MVIP); /* Make sure never 2 MVIP clock
                                             masters */

/* No clock on MVIP bus at this point as it could cause DSP carrier cards to
   fail */

sw_clock_control(card_id_b, CLOCK_REF_NET1); /* New clock reference on second
                                             card */
```

The MVIP bus provides a clocking feature to allow clock changeover between cards without

any discontinuity occurring through use of its secondary clock line (SEC8K). A second card may supply timing information from one of its clock reference sources to the MVIP SEC8K line while remaining synchronized to the MVIP bus clock generated by the card acting as MVIP bus clock master. If clock change over to the second card's source becomes necessary, then the card acting as clock master can then switch its clock generation circuit reference source to be the MVIP SEC8K line. This card's PLL ensures that no discontinuity occurs in the MVIP bus clock. For example:

```
sw_clock_control(card_id_a,CLOCK_REF_NET1); /* Initial clock reference */
sw_clock_control(card_id_b,CLOCK_REF_MVIP); /* Initial clock reference */

/* application determines clock changeover required */

/* Net 1 on this card is required new clock source */

sw_clock_control(card_id_b, SEC8K_DRIVEN_BY_NET1);

/* Card 1 still slaving off MVIP bus */

sw_clock_control(card_id_a, CLOCK_REF_SEC8K);

/* MVIP clock source now taken from SEC8K which is driven by other card's
network port */
```

Note when a call to `sw_clock_control()` is made with a parameter value of `SEC8K_DRIVEN_BY_NET1`, the affected card still continues to clock slave of the MVIP bus clock (i.e. there is an implicit `CLOCK_REF_MVIP` inherent in `SEC8K_DRIVEN_BY_NET1`).

Note calls to `sw_clock_control()` are never cumulative in effect; clock generation circuit behavior is determined completely from the parameter specified in the call made.

5.1.4 SCbus clocking

Note Not applicable to Prosody X PCI cards

The SC bus has only one set of clock signals used for synchronising data transfer across the bus and distributing network timing among multiple network cards, referred to here as the SC bus clocks.

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

Initial SC bus clocking will be set up during switch driver initialisation, (see section - Controlling clocking set up during system initialisation).

Single clock master/multiple clock slaves configuration

One card would be designated as SC bus clock master and drive SC bus clocks, and all other cards would be designated SCbus clock slaves and take their timing from the SC bus clocks.

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

```
/* PCI cards with legacy adaptor running in SCbus mode */
sw_clock_control(card_id_a, CLOCK_REF_NET1);
sw_clock_control(card_id_b, CLOCK_REF_SCBUS);
```

The above code fragment is applicable to a system consisting of PCI cards fitted with CT BUS legacy adapters interconnected by an SC bus. Here the clock reference for the first card does not need to be modified, as the SC bus is the only operational expansion bus in this configuration.

SC bus clock reference changeover

Sometimes (for example in the case of trunk failure) it is necessary to change the clock reference for a system from one network port to another, or from the local oscillator on one card to one on another.

As previously noted in the previous section, some resource cards, especially DSP carrier cards can be disrupted by any discontinuity occurring in the expansion bus clock signal if they are clock slaving off this expansion bus.

If the clock reference is simply changed from one source to another source on the expansion bus clock mastering card, for example:

```
/* Initial clock reference */
sw_clock_control(card_id_a, CLOCK_REF_NET1+DRIVE_SCBUS);

/* application determines clock changeover required */
/* New clock reference */
sw_clock_control(card_id_a, CLOCK_REF_NET2+DRIVE_SCBUS);
```

Then no discontinuity will occur because of the Phase Locked Loop (PLL) device with which the card is equipped.

The SC bus does not have any secondary clock signals that would allow glitch free clock master changer from a reference source on one card to another card.

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 at:

- system boot up time through automatic configuration mechanism
- use of the Aculab Configuration Tool
- application run time through explicit calls to `sw_clock_control()` or H-Bus clock set up API calls

Once a card has been configured as a bus clock master or bus clock slave it will remain configured in the same way until any call is made to change clock configuration such as `sw_clock_control()`.

5.2.1 Manually configuring the automatic configuration mechanism

The clocking settings for cards in the system, is 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 file is named according to the serial number of the card it applies to. 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 a line that contains the word "[Switch]". It ends with a line that contains the word "[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 field name followed by an equals and then the value assigned to that field.

Note The fields in this section are case sensitive

5.2.2 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 primary or secondary master or a 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* or *H100_SOURCE_NETWORK*).
H100_MASTER_B - card will drive the B-clocks on the H.100 bus (*Source* must be one of *H100_SOURCE_INTERNAL* or *H100_SOURCE_NETWORK*).

AutoFallBack - This field determines the card's behavior when an H.100 clock failure occurs. This field can contain one of the following values:

H100_FALLBACK_DISABLED
H100_FALLBACK_ENABLED
H100_AUTO_RETURN
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 card should terminate the H.100 bus. Possible values:

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

Note Changing this setting requires a restart of the drivers for the card. This will only occur the first time this setting is applied to a card - upon subsequent reboots the card will start in the correct mode

5.2.3 Legacy configuration

It is possible to put the following Aculab cards into legacy modes that support SC-BUS or MVIP:

Aculab Prosody PCI
Aculab E1/T1 Trunk PCI

To connect to cards that have SC-BUS or MVIP connectors it is necessary to use an Aculab Legacy Adapter.

The configuration settings for legacy bus modes are controlled by the following entries in the `config` file for a card:

ctBus: This setting controls the bus that the card will use. The possible values are:

```
SWMODE_CTBUS_H100 - H.100 mode (the default)
SWMODE_CTBUS_MVIP - MVIP mode
SWMODE_CTBUS_SCBUS - SC-BUS mode
```

Mode: This setting controls the clocking behaviour of the card. It can take one of the following values:

```
CLOCK_REF_NET1 - drive bus, take clock from network port 0
CLOCK_REF_NET2 - drive bus, take clock from network port 1
CLOCK_REF_NET3 - drive bus, take clock from network port 2
CLOCK_REF_NET4 - drive bus, take clock from network port 3
CLOCK_REF_MVIP - take clock from MVIP bus
CLOCK_REF_LOCAL - drive bus, generate clock locally
CLOCK_REF_SCBUS - take clock from SC-BUS
```

Note Changing bus mode will trigger a restart of the driver for this card. This will only happen the first time the card is put into legacy mode - upon subsequent reboots it will start in the correct mode.

ctLimitOverride: When in MVIP or SCBUS mode, Aculab PCI cards restrict their switching capacity in order to better operate with substandard competitor's cards. There is a method to enable the card's full capacity for when you know that the cards it is interfacing with can cope with it.

The possible values for this entry can be:

```
TRUE
FALSE
```

Note Changing this setting requires a restart of the drivers for the card. This will only occur the first time this setting is applied to a card - upon subsequent reboots the card will start in the correct mode.

Note This setting only applies to SCBUS and MVIP modes.

ctBusTermination: This card should terminate the bus. Possible values:

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

Note Only the MVIP bus can be terminated in this way.

Note Changing this setting requires a restart of the drivers for the card. This will only occur the first time this setting is applied to a card - upon subsequent reboots the card will start in the correct mode."

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)
- that MVIP stream numbers specified for `sw_set_output()` take into account your application's use of DSi and DSo streams and how MVIP stream numbering is different for the `ist` and `ost` `OUTPUT_PARMS` parameters
- 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 invoke `sw_set_output()` with mode set to `DISABLE_OUTPUT` for that expansion bus timeslot
- your application has not tried to switch simultaneously to the same timeslot on a particular MVIP DSo/DSi stream (see above section: MVIP Switching Limitations of Network Cards)
- 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 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 digital Access 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 digital Access 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_OUTPUT` for that expansion bus timeslot
- the signal has the appropriate encoding for the network or speech processing resource (A-law or μ -law)
- check use of legacy bus mode on PCI card is appropriate - see card installation guide for more details
- check expansion bus has been terminated appropriately
- if expansion bus is being used at or near its loading limit, try making the card that is expansion bus clock master physically the card in the middle of the 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 (in `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. 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`. Only a limited number of connections can be made up to and down from the expansion bus for a particular digital access card, see the appropriate card installation guide for more details.

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: digital access card stream numbering and clock settings

A.1 Prosody PCI/cPCI card stream usage

The interpretation of the first 32 logical stream numbers used in switch driver API calls for the Prosody PCI/cPCI card depend on the card mode.

When the card expansion bus is operating in H-Bus mode these streams are used as follows:

Stream Name	Available channels	Stream number	Details
H.100 stream D0-D31	0-127	0..31	H.100 streams D0..D31

When the card expansion bus is operating in MVIP bus mode (not applicable to cPCI card) these streams are used as follows:

Prosody PCI card Stream Numbering			
Stream Name	Available channels	Stream number	Details
MVIP Stream 0	0-31	0	normal direction DSi0 if output, DSo0 if input
MVIP Stream 1	0-31	1	normal direction DSi1 if output, DSo1 if input
MVIP Stream 2	0-31	2	normal direction DSi2 if output, DSo2 if input
MVIP Stream 3	0-31	3	normal direction DSi3 if output, DSo3 if input
MVIP Stream 4	0-31	4	normal direction DSi4 if output, DSo4 if input
MVIP Stream 5	0-31	5	normal direction DSi5 if output, DSo5 if input
MVIP Stream 6	0-31	6	normal direction DSi6 if output, DSo6 if input
MVIP Stream 7	0-31	7	normal direction DSi7 if output, DSo7 if input
MVIP Stream 0	0-31	8	reverse direction, DSo0 if output, DSi0 if input
MVIP Stream 1	0-31	9	reverse direction, DSo1 if output, DSi1 if input
MVIP Stream 2	0-31	10	reverse direction, DSo2 if output, DSi2 if input
MVIP Stream 3	0-31	11	reverse direction, DSo3 if output, DSi3 if input
MVIP Stream 4	0-31	12	reverse direction, DSo4 if output, DSi4 if input
MVIP Stream 5	0-31	13	reverse direction, DSo5 if output, DSi5 if input
MVIP Stream 6	0-31	14	reverse direction, DSo6 if output, DSi6 if input
MVIP Stream 7	0-31	15	reverse direction, DSo7 if output, DSi7 if input

When the card expansion bus is operating in SCbus mode (not applicable to cPCI card) these streams are used as follows:

Prosody PCI card Stream Numbering			
Stream Name	Available Channels	Stream Number	Details
SC bus stream	0-1023	24	SC bus stream

The remaining streams are numbered as follows:

Prosody PCI/cPCI card Stream Numbering			
Stream Name	Available Channels	Stream Number	Details
Network Port 1	E1: 1-15,17-31 T1: 0-22,23	32	Port 1 bearer channels T1 use of timeslot 23 depends on signalling
Network Port 2	E1: 1-15,17-31 T1: 0-22,23	33	Port 2 bearer channels T1 use of timeslot 23 depends on signalling
Network Port 3	E1: 1-15,17-31 T1: 0-22,23	34	Port 3 bearer channels T1 use of timeslot 23 depends on signalling
Network Port 4	E1: 1-15,17-31 T1: 0-22,23	35	Port 4 bearer channels T1 use of timeslot 23 depends on signalling
DSP32/65 0	0-31	40	
DSP32/65 1	0-31	41	
DSP32/65 2	0-31	42	
DSP32/65 3	0-31	43	
Prosody Module 0 Stream A	0-31	48	
Prosody Module 0 Stream B	0-31	49	
Prosody Module 1 Stream A	0-31	50	
Prosody Module 1 Stream B	0-31	51	
Prosody Module 2 Stream A	0-31	52	
Prosody Module 2 Stream B	0-31	53	

Prosody PCI/cPCI card Stream Numbering			
Stream Name	Available Channels	Stream Number	Details
Prosody Module 3 Stream A	0-31	54	
Prosody Module 3 Stream B	0-31	55	

A.2 Prosody PCI/cPCI card clock settings

The possible clock settings depend on the mode the card is operating in.

In H-Bus mode, clock control is best achieved via H-Bus specific clock control calls. If however an application makes calls to `sw_clock_control()`, the following modes are applicable (master modes drive H-Bus 'A' clocks):

Reference source	<code>sw_clock_control()</code> parameter
Network port 1..4	<code>CLOCK_REF_NET1, CLOCK_REF_NET2, CLOCK_REF_NET3, CLOCK_REF_NET4</code>
H.100 bus clock	<code>CLOCK_REF_MVIP</code> or <code>CLOCK_REF_H100</code>
Local oscillator	<code>CLOCK_REF_LOCAL</code>

The card will act as H-Bus bus 'A' clocks master for all the above clock modes except:

- `CLOCK_REF_H100` – card is H-Bus clock slave
- `CLOCK_REF_MVIP` – as above, for compatibility

In MVIP bus mode (not applicable to cPCI card) the following modes are applicable:

Reference source	<code>sw_clock_control()</code> parameter
Network port 1..4	<code>CLOCK_REF_NET1, CLOCK_REF_NET2, CLOCK_REF_NET3, CLOCK_REF_NET4</code>
MVIP bus clock	<code>CLOCK_REF_MVIP</code> or <code>SEC8K_DRIVEN_BY_NET1</code> or <code>SEC8K_DRIVEN_BY_NET2</code> or <code>SEC8K_DRIVEN_BY_NET3</code> or <code>SEC8K_DRIVEN_BY_NET4</code> or <code>SEC8K_DRIVEN_BY_LOCAL</code>

The card will act as MVIP bus clock master for all the above clock modes except

- `CLOCK_REF_MVIP` – card is MVIP bus clock slave
- `SEC8K_DRIVEN_BY_NET1` – card is MVIP bus clock slave
- `SEC8K_DRIVEN_BY_NET2` – card is MVIP bus clock slave
- `SEC8K_DRIVEN_BY_NET3` – card is MVIP bus clock slave
- `SEC8K_DRIVEN_BY_NET4` – card is MVIP bus clock slave
- `SEC8K_DRIVEN_BY_LOCAL` – card is MVIP bus clock slave
- `CLOCK_REF_PRIVATE` – card not synchronized to MVIP bus

In SC bus mode (not applicable to cPCI card) the following modes are applicable:

Reference source	sw_clock_control() parameter
Network port 1...4	CLOCK_REF_NET1, CLOCK_REF_NET2, CLOCK_REF_NET3, CLOCK_REF_NET4
SCbus bus clock	CLOCK_REF_SCBUS
Local oscillator	CLOCK_REF_LOCAL

The card will act as SCbus clock master for all the above clock modes except:

- CLOCK_REF_SCBUS – card is SC bus clock slave

A.3 Prosody X card stream usage

Stream names	Available channels	Stream numbers	Details	Notes
H.100/110-bus stream D0-D31	0-127	0-31	H.100/110-bus streams D0...D31	1
Network Ports 1-16	E1: 1-15,17-31 T1: 0-22,23	32-47	Port 1-16 bearer channels	2,3
Network Ports 1-16	E1: 1-15,17-31 T1: 0-22,23	32-47	Port 1-16 bearer channels	2,3
Signalling DSP0 A1-A4	0-31	48-51		4
Signalling DSP0 B1-B4	0-31	52-55		4
Signalling DSP0 B1-B4	0-31	52-55		4
Signalling DSP1 A1-A4	0-31	56-59		4
Signalling DSP1 A1-A4	0-31	56-59		4
Signalling DSP1 B1-B4	0-31	60-63		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
TiNG2 0-1	0-127	66-67		5
TiNG3 0-1	0-127	68-69		5
TiNG3 0-1	0-127	68-69		5
TiNG4 0-1	0-127	70-71		5
TiNG4 0-1	0-127	70-71		5
TiNG5 0-1	0-127	80-81		5
TiNG6 0-1	0-127	82-83		5
TiNG7 0-1	0-127	84-85		5
TiNG8 0-1	0-127	86-87		5

Note 1: The Prosody X PCI and Prosody X PCIe card can only be configured to operate the CTBus in H.100 mode. The Prosody X cPCI card can only be configured to operate the CTBus in H.110 mode.

Note 2: T1 use of timeslot 23 depends on signalling.

Note 3: Maximum number of network ports permitted on each card type is as follows:

Card type	Maximum ports	Stream numbers
Prosody X PCIe	4	32-35
Prosody X PCI	8	32-39
Prosody X cPCI	16	32-47

For Prosody X PCI and Prosody X cPCI cards, the number of ports available depends on the type of PMX module fitted. For Prosody X PCIe cards, the number of ports available depends on the PCIe card variant.

Note 4: For Prosody X PCI and Prosody X cPCI cards, the number of signalling DSPs available depends on the type of PMX module fitted. For Prosody X PCIe, the number of signalling DSPs available depends on the PCIe card variant.

Note 5: Maximum number of TiNG DSPs available on each card type is as follows:

Card type	Maximum TiNG DSPs	Stream numbers
Prosody X PCIe	2	64-67
Prosody X PCI	4	64-71
Prosody X cPCI	8	64-71, 80-87

For Prosody X PCI cards and Prosody X PCIe cards, the number of TiNG DSPs available depends on the card variant. For Prosody X cPCI cards, the number of TiNG DSPs available depends on the type of TiNG DSP module fitted.

A.4 Prosody X card clock settings

The Prosody X PCI and Prosody X PCIe card can only be configured to operate the CTBus in H.100 mode. The Prosody X cPCI card can only be configured to operate the CTBus in H.110 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 modes are applicable:

Reference source	sw_clock_control() parameter	Card type
Network ports 1-3	CLOCK_REF_NET1 CLOCK_REF_NET2 CLOCK_REF_NET3 CLOCK_REF_NET4	All Prosody X cards
Network ports 4-7	CLOCK_REF_NET5 CLOCK_REF_NET6 CLOCK_REF_NET7 CLOCK_REF_NET8	Prosody X PCI and Prosody X cPCI only
Network ports 8-16	CLOCK_REF_NET9 CLOCK_REF_NET10 CLOCK_REF_NET11 CLOCK_REF_NET12 CLOCK_REF_NET13 CLOCK_REF_NET14 CLOCK_REF_NET15 CLOCK_REF_NET16	Prosody X cPCI only
H.100/110 bus clock	CLOCK_REF_H100	All Prosody X cards
Local oscillator	CLOCK_REF_LOCAL	All Prosody X cards

The card will act as H.100/110 bus 'A' clocks master in all of these clock modes except for CLOCK_REF_H110 where the card is configured as an H.100/110 bus clock slave.

A.5 E1/T1 PCI card stream usage

The interpretation of the first 32 logical stream numbers used in switch driver API calls for the E1/T1 PCI Card depend on the card mode.

When the card expansion bus is operating in H.100 mode these streams are used as follows:

E1/T1 PCI card Stream Numbering			
Stream Name	Available Channels	Stream Number	Details
H.100 stream D0-D31	0-127	0..31	H.100 streams D0..D31

When the card expansion bus is operating in MVIP bus mode these streams are used as follows:

E1/T1 PCI card Stream Numbering			
Stream Name	Available Channels	Stream Number	Details
MVIP Stream 0	0-31	0	normal direction DSi0 if output, DSo0 if input
MVIP Stream 1	0-31	1	normal direction DSi1 if output, DSo1 if input
MVIP Stream 2	0-31	2	normal direction DSi2 if output, DSo2 if input
MVIP Stream 3	0-31	3	normal direction DSi3 if output, DSo3 if input
MVIP Stream 4	0-31	4	normal direction DSi4 if output, DSo4 if input
MVIP Stream 5	0-31	5	normal direction DSi5 if output, DSo5 if input

E1/T1 PCI card Stream Numbering			
Stream Name	Available Channels	Stream Number	Details
MVIP Stream 6	0-31	6	normal direction DSi6 if output, DSo6 if input
MVIP Stream 7	0-31	7	normal direction DSi7 if output, DSo7 if input
MVIP Stream 0	0-31	8	reverse direction, DSo0 if output, DSi0 if input
MVIP Stream 1	0-31	9	reverse direction, DSo1 if output, DSi1 if input
MVIP Stream 2	0-31	10	reverse direction, DSo2 if output, DSi2 if input
MVIP Stream 3	0-31	11	reverse direction, DSo3 if output, DSi3 if input
MVIP Stream 4	0-31	12	reverse direction, DSo4 if output, DSi4 if input
MVIP Stream 5	0-31	13	reverse direction, DSo5 if output, DSi5 if input
MVIP Stream 6	0-31	14	reverse direction, DSo6 if output, DSi6 if input
MVIP Stream 7	0-31	15	reverse direction, DSo7 if output, DSi7 if input

When the card expansion bus is operating in SCbus mode these streams are used as follows:

E1/T1 PCI card Stream Numbering			
Stream Name	Available Channels	Stream Number	Details
SC bus stream	0-1023	24	SC bus stream

The remaining streams are numbered as follows:

E1/T1 PCI card Stream Numbering			
Stream Name	Available Channels	Stream Number	Details
Network Port 1	E1: 1-15,17-31 T1: 0-22,23	32	Port 1 bearer channels T1 use of timeslot 23 depends on signalling
Network Port 2	E1: 1-15,17-31 T1: 0-22,23	33	Port 2 bearer channels T1 use of timeslot 23 depends on signalling
Network Port 3	E1: 1-15,17-31 T1: 0-22,23	34	Port 3 bearer channels T1 use of timeslot 23 depends on signalling
Network Port 4	E1: 1-15,17-31 T1: 0-22,23	35	Port 4 bearer channels T1 use of timeslot 23 depends on signalling
DSP32/65 0	0-31	40	
DSP32/65 1	0-31	41	

E1/T1 PCI card Stream Numbering			
Stream Name	Available Channels	Stream Number	Details
DSP32/65 2	0-31	42	
DSP32/65 3	0-31	43	
DSP32/65 4	0-31	44	
DSP32/65 5	0-31	45	
DSP32/65 6	0-31	46	
DSP32/65 7	0-31	47	

A.6 E1/T1 PCI card clock settings

The possible clock settings depend on the mode the card is operating in.

In H.100 bus mode, clock control is best achieved via H.100 specific clock control calls. If however an application makes calls to `sw_clock_control()`, the following modes are applicable (master modes drive H.100 'A' clocks):

Reference source	<code>sw_clock_control()</code> parameter
Network port 1..4	<code>CLOCK_REF_NET1</code> , <code>CLOCK_REF_NET2</code> , <code>CLOCK_REF_NET3</code> , <code>CLOCK_REF_NET4</code>
H.100 bus clock	<code>CLOCK_REF_MVIP</code> or <code>CLOCK_REF_H100</code>
Local oscillator	<code>CLOCK_REF_LOCAL</code>

The card will act as H.100 bus 'A' clocks master for all the above clock modes except:

- `CLOCK_REF_H100` – card is H.100 bus clock slave
- `CLOCK_REF_MVIP` – as above, for compatibility

In MVIP bus mode the following modes are applicable:

Reference source	<code>sw_clock_control()</code> parameter
Network port 1..4	<code>CLOCK_REF_NET1</code> , <code>CLOCK_REF_NET2</code> , <code>CLOCK_REF_NET3</code> , <code>CLOCK_REF_NET4</code>
MVIP bus clock	<code>CLOCK_REF_MVIP</code> or <code>SEC8K_DRIVEN_BY_NET1</code> or <code>SEC8K_DRIVEN_BY_NET2</code> or <code>SEC8K_DRIVEN_BY_NET3</code> or <code>SEC8K_DRIVEN_BY_NET4</code> or <code>SEC8K_DRIVEN_BY_LOCAL</code>
Secondary MVIP bus clock	<code>CLOCK_REF_SEC8K</code>
Local oscillator	<code>CLOCK_REF_LOCAL</code> or <code>CLOCK_REF_PRIVATE</code>

The card will act as MVIP bus clock master for all the above clock modes except:

CLOCK_REF_MVIP	– card is MVIP bus clock slave
SEC8K_DRIVEN_BY_NET1	– card is MVIP bus clock slave
SEC8K_DRIVEN_BY_NET2	– card is MVIP bus clock slave
SEC8K_DRIVEN_BY_NET3	– card is MVIP bus clock slave
SEC8K_DRIVEN_BY_NET4	– card is MVIP bus clock slave
SEC8K_DRIVEN_BY_LOCAL	– card is MVIP bus clock slave
CLOCK_REF_PRIVATE	– card not synchronized to MVIP bus

In SC bus mode the following modes are applicable:

Reference source	sw_clock_control() parameter
Network port 1...4	CLOCK_REF_NET1, CLOCK_REF_NET2, CLOCK_REF_NET3, CLOCK_REF_NET4
SCbus bus clock	CLOCK_REF_SCBUS
Local oscillator	CLOCK_REF_LOCAL

The card will act as SCbus clock master for all the above clock modes except:

CLOCK_REF_SCBUS	– card is SC bus clock slave
-----------------	------------------------------

A.7 E1/T1 cPCI card stream usage

The first 32 logical stream numbers (#) used in switch driver API calls for the E1/T1 cPCI Card are used as follows:

H.110 CTBus stream numbering

Stream Name	Available Channels	Stream Number	Details
H.110 stream D0-D31	0-127	0...31	H.110 streams D0...D31

The remaining streams are detailed below:

Note The available channels for each Network port are; E1:1-15, 17-31 and T1: 0-22,23. Use of timeslot 23 for T1 is subject to signalling.

Note The available channels for each DSP65 is 0-31

A.7.1 PM4 stream numbering

E1/T1 cPCI card Stream Numbering (using PM4s)

Stream Name	Available Channels	Stream number	Details
Network Port 1	E1: 1-15,17-31 T1: 0-22,23	32	Port 1 bearer channels
Network Port 2	E1: 1-15,17-31 T1: 0-22,23	33	Port 2 bearer channels
Network Port 3	E1: 1-15,17-31 T1: 0-22,23	34	Port 3 bearer channels
Network Port 4	E1: 1-15,17-31 T1: 0-22,23	35	Port 4 bearer channels
Network Port 5	E1: 1-15,17-31 T1: 0-22,23	36	Port 5 bearer channels
Network Port 6	E1: 1-15,17-31 T1: 0-22,23	37	Port 6 bearer channels
Network Port 7	E1: 1-15,17-31 T1: 0-22,23	38	Port 7 bearer channels
Network Port 8	E1: 1-15,17-31 T1: 0-22,23	39	Port 8 bearer channels

E1/T1 cPCI card Stream Numbering (using PM4s)

Stream Name	Available Channels	Stream number	Details
DSP65 1 st port – DSPA 1	0-31	40	
DSP65 1 st port – DSPA 2	0-31	41	
DSP65 1 st port – DSPA 3	0-31	42	
DSP65 1 st port – DSPA 4	0-31	43	
DSP65 1 st port – DSPA 5	0-31	44	
DSP65 1 st port – DSPA 6	0-31	45	
DSP65 1 st port – DSPA 7	0-31	46	
DSP65 1 st port – DSPA 8	0-31	47	
DSP65 2 nd port – DSPB 1	0-31	56	
DSP65 2 nd port – DSPB 2	0-31	57	
DSP65 2 nd port – DSPB 3	0-31	58	
DSP65 2 nd port – DSPB 4	0-31	59	
DSP65 2 nd port – DSPB 5	0-31	60	
DSP65 2 nd port – DSPB 6	0-31	61	
DSP65 2 nd port – DSPB 7	0-31	62	
DSP65 2 nd port – DSPB 8	0-31	63	

A.7.2 PMX stream numbering

E1/T1 cPCI card Stream Numbering (using PMX)

Stream Name	Available Channels	Stream number	Details
Network Port 1	E1: 1-15,17-31 T1: 0-22,23	32	Port 1 bearer channels
Network Port 2	E1: 1-15,17-31 T1: 0-22,23	33	Port 2 bearer channels
Network Port 3	E1: 1-15,17-31 T1: 0-22,23	34	Port 3 bearer channels
Network Port 4	E1: 1-15,17-31 T1: 0-22,23	35	Port 4 bearer channels
Network Port 5	E1: 1-15,17-31 T1: 0-22,23	36	Port 5 bearer channels
Network Port 6	E1: 1-15,17-31 T1: 0-22,23	37	Port 6 bearer channels
Network Port 7	E1: 1-15,17-31 T1: 0-22,23	38	Port 7 bearer channels
Network Port 8	E1: 1-15,17-31 T1: 0-22,23	39	Port 8 bearer channels

Stream Name	Available Channels	Stream number	Details
Network Port 9	E1: 1-15,17-31 T1: 0-22,23	40	Port 9 bearer channels
Network Port 10	E1: 1-15,17-31 T1: 0-22,23	41	Port 10 bearer channels
Network Port 11	E1: 1-15,17-31 T1: 0-22,23	42	Port 11 bearer channels
Network Port 12	E1: 1-15,17-31 T1: 0-22,23	43	Port 12 bearer channels
Network Port 13	E1: 1-15,17-31 T1: 0-22,23	44	Port 13 bearer channels
Network Port 14	E1: 1-15,17-31 T1: 0-22,23	45	Port 14 bearer channels
Network Port 15	E1: 1-15,17-31 T1: 0-22,23	46	Port 15 bearer channels
Network Port 16	E1: 1-15,17-31 T1: 0-22,23	47	Port 16 bearer channels

E1/T1 cPCI card Stream Numbering (using PMX)

Stream Name	Available Channels	Stream Number	Details
DSP0 A1	0 - 31	48	PMX module DSPs 32 timeslots each
DSP0 A2	0 - 31	49	
DSP0 A3	0 - 31	50	
DSP0 A4	0 - 31	51	
DSP0 B1	0 - 31	52	
DSP0 B2	0 - 31	53	
DSP0 B3	0 - 31	54	
DSP0 B4	0 - 31	55	
DSP1 A1	0 - 31	56	
DSP1 A2	0 - 31	57	
DSP1 A3	0 - 31	58	
DSP1 A4	0 - 31	59	
DSP1 B1	0 - 31	60	
DSP1 B2	0 - 31	61	
DSP1 B3	0 - 31	62	
DSP1 B4	0 - 31	63	

A.8 E1/T1 cPCI card clock settings

The E1/T1 cPCI card can only be configured to operate the CT Bus in H.110 mode.

In this CT Bus mode, clock control is best achieved via H.100 specific clock control calls. If however an application makes calls to `sw_clock_control()`, the following modes are applicable (master modes drive H.100 'A' clocks):

Reference source	<code>sw_clock_control()</code> parameter
Network port 1... 16	CLOCK_REF_NET1 CLOCK_REF_NET2 CLOCK_REF_NET3 CLOCK_REF_NET4 CLOCK_REF_NET5 CLOCK_REF_NET6 CLOCK_REF_NET7 CLOCK_REF_NET8 CLOCK_REF_NET9 CLOCK_REF_NET10 CLOCK_REF_NET11 CLOCK_REF_NET12 CLOCK_REF_NET13 CLOCK_REF_NET14 CLOCK_REF_NET15 CLOCK_REF_NET16
H.100 bus clock	CLOCK_REF_MVIP or CLOCK_REF_H100
Local oscillator	CLOCK_REF_LOCAL

The card will act as H.100 bus 'A' clocks master for all the above clock modes except:

CLOCK_REF_H100 – card is H.100 bus clock slave

A.9 IP telephony PCI H.323 gateway card stream usage

The interpretation of the first 32 logical stream numbers used in switch driver API calls for the E1/T1 PCI VoIP Card depend on the card mode.

When the card expansion bus is operating in H.100 mode these streams are used as follows:

VoIP PCI H.323 Gateway Card Stream Numbering			
Stream Name	Available Channels	Stream Number	Details
H.100 stream D0-D31	0-127	0...31	H.100 streams D0...D31

The remaining streams are numbered as follows:

VoIP PCI H.323 Gateway Card Stream Numbering			
Stream Name	Available Channels	Stream Number	Details
Network Port 1	E1: 1-15,17-31 T1: 0-22,23	32	Port 1 bearer channels T1 use of timeslot 23 depends on signalling
Network Port 2	E1: 1-15,17-31 T1: 0-22,23	33	Port 2 bearer channels T1 use of timeslot 23 depends on signalling
DSP32 0	0-31	40	
DSP32 1	0-31	41	
DSP32 2	0-31	42	
DSP32 3	0-31	43	
VoIP module 0	0-15	48	
VoIP module 1	0-15	49	
VoIP module 2	0-15	50	
VoIP module 3	0-15	51	
VoIP module 4	0-15	52	
VoIP module 5	0-15	53	reserved for future use
VoIP module 6	0-15	54	reserved for future use
VoIP module 7	0-15	55	reserved for future use

Note The stream and timeslot numbering on an IP port of an IP Telephony card (SIP or H.323) is different to that for ISDN ports.

With the ISDN ports there is a one to one relationship between the port and the stream so that each port has a unique stream associated with it.

This is not the case with the IP Telephony ports. Here the stream and timeslot correspond to a channel on a VoIP DSP, where the media packets are processed, rather than a

physical timeslot on a TDM link. As there are multiple DSPs on an IP Telephony card, there are therefore multiple streams per IP Telephony port.

VoIP PCI H.323 gateway card clock settings

The possible clock settings depend on the mode the card is operating in.

In H.100 bus mode, clock control is best achieved via H.100 specific clock control calls. If however an application makes calls to `sw_clock_control()`, the following modes are applicable (master modes drive H.100 'A' clocks):

Reference source	<code>sw_clock_control()</code> parameter
Network port 1, 2	<code>CLOCK_REF_NET1</code> , <code>CLOCK_REF_NET2</code>
H.100 bus clock	<code>CLOCK_REF_MVIP</code> or <code>CLOCK_REF_H100</code>
Local oscillator	<code>CLOCK_REF_LOCAL</code>

The card will act as H.100 bus 'A' clocks master for all the above clock modes except:

- `CLOCK_REF_H100` – card is H.100 bus clock slave
- `CLOCK_REF_MVIP` – as above, for compatibility

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. It is not however possible using `sw_sample_input()` for an application to record or process all the samples on a bearer channel. 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).

The Prosody PCI card, the E1/T1 PCI card, and the cPCI Prosody card have restrictions on sampling external bus timeslots. If a connection exists on a card with an input from a given external bus timeslot, then it is not in general possible to invoke `sw_sample_input()` using the same card specifying the same external bus timeslot. Note this limitation does not apply to sampling of local resource timeslots (such as network port timeslots or DSP32 modules), and note also that sampling from local resource timeslots is a substantially faster operation than sampling from external bus timeslots.

Appendix C: API error codes

As defined in `sw_lib.h`

Error	Value	Meaning
SUCCESS	0	Command executed OK
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	Connection not possible. For Prosody X cards see Note in <code>sw_set_output()</code> .
ERR_SW_NO_SCBUS_CLOCK	-224	No card driving SC Bus clock
ERR_SW_OTHER_SCBUS_CLOCK	-225	Non Aculab card
ERR_SW_PATH_BLOCKED	-226	Switch connection cannot be made
ERR_SW_OS_INTERRUPTED	-227	Event wait interrupted.
ERR_SW_OS_INCONSISTENT_STATE	-228	Switch device and driver inconsistent
ERR_SW_PORT_NOT_LOADED	-229	Firmware is not running on PMX 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 <code>PX_Sw_Driver</code> package may be required
ERR_SW_T8110_UNKNOWN_IOCTL	-237	New <code>Prosody_IP_Firmware</code> may need to be flashed
ERR_SW_PXSCS_COMMS_FAILED	-238	Prosody X only: problem communicating with pxscs on Prosody X card
ERR_SW_API_CALL_ABORTED	-239	Prosody X only: 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 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 PCI Digital Access 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 under which the Aculab switch driver is supported.

Under multi-tasking operating systems, such as Windows and Unix, `swcmd` may be run concurrently with an application to verify which switch connections are made. Determine current card clocking modes and even output a log of API calls made to the switch driver 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 e.g.

```
swcmd -t 12345 2 -t 12346 1 -v
```

Which equates to:

```
swcmd <flag> <serial#> <clkmode> <flag> <serial#> <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#>	- card serial number i.e. 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	Invoke <code>swcmd</code> tool in verbose output mode, and print out driver version and card information, invokes <code>sw_ver_switch()</code> , and <code>sw_card_info()</code> .
-t	<serial#> <clkmode>	Set card clock circuit reference and/or expansion bus clock mastering/slaving mode using <clkmode> for card <serial#>, invokes <code>sw_clock_control()</code> .
-ti	None	Set clock mode interactively
-e	None	Display system clocking for all cards.
-c	<serial#> <ost> <ots> <ist> <its>	Make switch connection on card <serial#> switching data from <ist,its> to <ost,ots>
-o	<serial#> <ost> <ots> <pat>	Output constant pattern <pat> on card <serial#> on <ost,ots>
-d	<serial#> <ost> <ots>	Disable switch output <ost,ots> for card <serial#>

<Flag>	Parameters	Use
-q	<serial#> <ost> <ots>	Determine source of data being switched to <ost,ots> by card <serial#>, invokes <code>sw_query_output()</code> .
-y	<serial#>	Report no. of outputs been driven on expansion bus by card.
-p	<serial#> <mode>	Invokes <code>sw_switch_override_mode()</code> for driver policing of MUX switching on card <serial#>. If <mode> is zero, switch driver will operate in blocking mode, if <mode> is non-zero it will operate in override mode.
-a	<serial#> <ist> <its>	Obtain 8 bit sample of data currently being received by card <serial#> on <ist,its>. Uses <code>sw_sample_input()</code> API call.
-w	<predicted sample>	When used with <code>-a</code> and <code>-l</code> options, allows <code>swcmd</code> to loop sampling data until value not equal to <predicted sample> is obtained.
-r	<serial#>	Invokes <code>sw_reset_switch()</code> for driver <serial#>
-m	none	Determine if SCbus clock is being driven by any card, invokes <code>sw_clock_scbus_master</code> .
-k	none	Monitor switch driver API calls being made by an application to standard output, repeatedly invokes <code>sw_track_api_calls()</code> .
-2	none	Turn switch driver API call tracking on.
-0	none	Turn switch driver API call tracking off.
-?	none	Output <code>swcmd</code> command line syntax.
-h	<serial#> <src> <net> <mode> <afb> <ncs>	Configure H.100 clocks, invokes <code>sw_h100_config_board_clock()</code> for card <serial#>
-hi	none	Configure H.100 clocks interactively
-i	<serial#>	Query H.100 clocks, invokes <code>sw_h100_query_board_clock()</code> for card <serial#>
-j	<serial#> <net> <mode> <ncs>	Configure <code>CT_NETREF</code> , invokes <code>sw_h100_config_netref_clock()</code> for card <serial#>
-ji	none	Configure <code>CT_NETREF</code> interactively
-u	<serial#>	Query H.100 <code>CT_NETREF</code> , invokes <code>sw_h100_query_netref_clock()</code> for card <serial#>
-n	<repeat-count>	Iterate <code>swcmd</code> command - see below.
-l	none	Loop forever invoking the switch driver calls specified by other <code>swcmd</code> flags.
-f	<serial#> <stream> <rx_mode> <tx_mode>	Sets the companding mode (valid only for cards fitted with a PMX module).

<Flag>	Parameters	Use
-ft	<serial#> <stream> <timeslot> <rx_mode> <tx_mode>	Sets the companding mode on a timeslot (valid only for cards fitted with a PMX module).
-g	<serial#> <stream>	Determines which companding mode has been set (valid only for cards fitted with a PMX module)
-gt	<serial#> <stream> <timeslot>	Determines which companding mode has been set on a timeslot (valid only for cards fitted with a PMX module)
-b	<serial#> <type> <position> <port>	Invokes <code>sw_get_dsp_stream_info()</code> , (Prosody X cards only).
-s	<serial#> <mode>	Turns the bus termination on or off (mode = 1 turns the termination on, mode = 0 turns the termination off). (PCI cards only).

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 them 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 which remain constant (card specifier <drv>, and stream numbers) and parameters, which 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 display version numbers of all switch drivers installed in the system, identify the type of cards they are associated with, and for PCI/cPCI cards only display their machine readable serial numbers. used to change a card circuit clock reference:

```
swcmd -v
```

Changing card clock mode

`swcmd` may be used to change a card circuit clock reference:

```
swcmd -t <drv> <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 that the cable is OK, is to output a pattern from one card onto the bus, and sample it on a second card.

Note here we specify output stream DSo0 using 0 for first command where it is an output stream, and 8 on second command where it is an input stream, see section 4.3 for more details:

```
swcmd -o 12345 0 0 0x42
swcmd -a 67890 8 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` aborted by the operator.

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 16 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 UNIX 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})
```