Dialogic
Making Innovation Thrive™

Dialogic® NaturalAccess™ Switching Interface API Developer's Manual

October 2009 64-0504-01

www.dialogic.com

Copyright and legal notices

All contents of this document are furnished for informational use only and are subject to change without notice and do not represent a commitment on the part of Dialogic Corporation or its subsidiaries ("Dialogic"). Reasonable effort is made to ensure the accuracy of the information contained in the document. However, Dialogic does not warrant the accuracy of this information and cannot accept responsibility for errors, inaccuracies or omissions that may be contained in this document.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH DIALOGIC® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN A SIGNED AGREEMENT BETWEEN YOU AND DIALOGIC, DIALOGIC ASSUMES NO LIABILITY WHATSOEVER, AND DIALOGIC DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF DIALOGIC PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY INTELLECTUAL PROPERTY RIGHT OF A THIRD PARTY.

Dialogic products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

Due to differing national regulations and approval requirements, certain Dialogic products may be suitable for use only in specific countries, and thus may not function properly in other countries. You are responsible for ensuring that your use of such products occurs only in the countries where such use is suitable. For information on specific products, contact Dialogic Corporation at the address indicated below or on the web at www.dialogic.com.

It is possible that the use or implementation of any one of the concepts, applications, or ideas described in this document, in marketing collateral produced by or on web pages maintained by Dialogic may infringe one or more patents or other intellectual property rights owned by third parties. Dialogic does not provide any intellectual property licenses with the sale of Dialogic products other than a license to use such product in accordance with intellectual property owned or validly licensed by Dialogic and no such licenses are provided except pursuant to a signed agreement with Dialogic. More detailed information about such intellectual property is available from Dialogic's legal department at 9800 Cavendish Blvd., 5th Floor, Montreal, Quebec, Canada H4M 2V9. Dialogic encourages all users of its products to procure all necessary intellectual property licenses required to implement any concepts or applications and does not condone or encourage any intellectual property infringement and disclaims any responsibility related thereto. These intellectual property licenses may differ from country to country and it is the responsibility of those who develop the concepts or applications to be aware of and comply with different national license requirements.

Any use case(s) shown and/or described herein represent one or more examples of the various ways, scenarios or environments in which Dialogic® products can be used. Such use case(s) are non-limiting and do not represent recommendations of Dialogic as to whether or how to use Dialogic products.

Dialogic, Dialogic Pro, Brooktrout, Diva, Cantata, SnowShore, Eicon, Eicon Networks, NMS Communications, NMS (stylized), Eiconcard, SIPcontrol, Diva ISDN, TruFax, Exnet, EXS, SwitchKit, N20, Making Innovation Thrive, Connecting to Growth, Video is the New Voice, Fusion, Vision, PacketMedia, NaturalAccess, NaturalCallControl, NaturalConference, NaturalFax and Shiva, among others as well as related logos, are either registered trademarks or trademarks of Dialogic Corporation or its subsidiaries. Dialogic's trademarks may be used publicly only with permission from Dialogic. Such permission may only be granted by Dialogic's legal department at 9800 Cavendish Blvd., 5th Floor, Montreal, Quebec, Canada H4M 2V9. Any authorized use of Dialogic's trademarks will be subject to full respect of the trademark guidelines published by Dialogic from time to time and any use of Dialogic's trademarks requires proper acknowledgement.

Windows is a registered trademark of Microsoft Corporation in the United States and/or other countries. The names of actual companies and product mentioned herein are the trademarks of their respective owners.

This document discusses one or more open source products, systems and/or releases. Dialogic is not responsible for your decision to use open source in connection with Dialogic products (including without limitation those referred to herein), nor is Dialogic responsible for any present or future effects such usage might have, including without limitation effects on your products, your business, or your intellectual property rights.

Revision history

Revision	Release date	Notes
9000-60006-10	July 2000	LBG, Platform support for Fusion 4.0
9000-60006-11	September 2000	LBG, CT Access 4.0
9000-60006-12	March 2001	GJG, NACD 2000-2
9000-60006-13	June 2001	NBS
9000-60006-14	August 2001	NBS, NACD 2001-1
9000-60006-15	November 2001	NBS, NACD 2002-1 Beta
9000-60006-16	May 2002	SRG, NACD 2002-1
9000-60006-17	April 2003	MCM, NACD 2003-1
9000-60006-18	December 2003	MVH, NACD 2004-1
9000-60006-19	November 2004	SRR, NACD 2005-1
9000-60006-20	December 2006	LBZ, 2005-1, SP3
9000-60006-21	June 2008	SRG/LBZ, Natural Access R8
64-0504-01	October 2009	LBG, NaturalAccess R9.0
Last modified: September 13, 2009		

Refer to www.dialogic.com for product updates and for information about support policies, warranty information, and service offerings.

Table Of Contents

1 Introduction

The *Dialogic® NaturalAccess™ Switching Interface API Developer's Manual* provides:

- An overview of the Switching service
- A reference of functions and errors

This manual defines telephony terms where applicable, but assumes that you are familiar with telephony concepts and switching. It also assumes that you are familiar with the C programming language.

Note: References to MVIP-90 in this manual are for legacy systems only.

Read the *Dialogic® NaturalAccess™ Software Developer's Manual* before using this manual. The *Dialogic® NaturalAccess™ Software Developer's Manual* contains detailed information on NaturalAccess concepts, architecture, and application development. This information must be fully understood before you develop a switching application using NaturalAccess.

2 Terminology

Note: The product to which this document pertains is part of the NMS Communications Platforms business that was sold by NMS Communications Corporation ("NMS") to Dialogic Corporation ("Dialogic") on December 8, 2008. Accordingly, certain terminology relating to the product has been changed. Below is a table indicating both terminology that was formerly associated with the product, as well as the new terminology by which the product is now known. This document is being published during a transition period; therefore, it may be that some of the former terminology will appear within the document, in which case the former terminology should be equated to the new terminology, and vice versa.

Former terminology	Dialogic terminology
CG 6060 Board	Dialogic® CG 6060 PCI Media Board
CG 6060C Board	Dialogic® CG 6060C CompactPCI Media Board
CG 6565 Board	Dialogic® CG 6565 PCI Media Board
CG 6565C Board	Dialogic® CG 6565C CompactPCI Media Board
CG 6565e Board	Dialogic® CG 6565E PCI Express Media Board
CX 2000 Board	Dialogic® CX 2000 PCI Station Interface Board
CX 2000C Board	Dialogic® CX 2000C CompactPCI Station Interface Board
AG 2000 Board	Dialogic® AG 2000 PCI Media Board
AG 2000C Board	Dialogic® AG 2000C CompactPCI Media Board
AG 2000-BRI Board	Dialogic® AG 2000-BRI Media Board
NMS OAM Service	Dialogic® NaturalAccess™ OAM API
NMS OAM System	Dialogic® NaturalAccess™ OAM System
NMS SNMP	Dialogic® NaturalAccess™ SNMP API
Natural Access	Dialogic® NaturalAccess™ Software
Natural Access Service	Dialogic® NaturalAccess™ Service
Fusion	Dialogic® NaturalAccess™ Fusion™ VoIP API
ADI Service	Dialogic® NaturalAccess™ Alliance Device Interface API
CDI Service	Dialogic® NaturalAccess™ CX Device Interface API
Digital Trunk Monitor Service	Dialogic® NaturalAccess™ Digital Trunk Monitoring API
MSPP Service	Dialogic® NaturalAccess™ Media Stream Protocol Processing API
Natural Call Control Service	Dialogic® NaturalAccess™ NaturalCallControl™ API
NMS GR303 and V5 Libraries	Dialogic® NaturalAccess™ GR303 and V5 Libraries

Former terminology	Dialogic terminology
Point-to-Point Switching Service	Dialogic® NaturalAccess™ Point-to-Point Switching API
Switching Service	Dialogic® NaturalAccess™ Switching Interface API
Voice Message Service	Dialogic® NaturalAccess™ Voice Control Element API
NMS CAS for Natural Call Control	Dialogic® NaturalAccess™ CAS API
NMS ISDN	Dialogic® NaturalAccess™ ISDN API
NMS ISDN for Natural Call Control	Dialogic® NaturalAccess™ ISDN API
NMS ISDN Messaging API	Dialogic® NaturalAccess™ ISDN Messaging API
NMS ISDN Supplementary Services	Dialogic® NaturalAccess™ ISDN API Supplementary Services
NMS ISDN Management API	Dialogic® NaturalAccess™ ISDN Management API
NaturalConference Service	Dialogic® NaturalAccess™ NaturalConference™ API
NaturalFax	Dialogic® NaturalAccess™ NaturalFax™ API
SAI Service	Dialogic® NaturalAccess™ Universal Speech Access API
NMS SIP for Natural Call Control	Dialogic® NaturalAccess™ SIP API
NMS RJ-45 interface	Dialogic® MD1 RJ-45 interface
NMS RJ-21 interface	Dialogic® MD1 RJ-21 interface
NMS Mini RJ-21 interface	Dialogic® MD1 Mini RJ-21 interface
NMS Mini RJ-21 to NMS RJ-21 cable	Dialogic® MD1 Mini RJ-21 to MD1 RJ-21 cable
NMS RJ-45 to two 75 ohm BNC splitter cable	Dialogic® MD1 RJ-45 to two 75 ohm BNC splitter cable
NMS signal entry panel	Dialogic® Signal Entry Panel

3 Overview of the Switching service

Switching service definition

The Natural Access Switching service:

- Provides a set of functions for controlling multi-vendor integration protocol (MVIP) switch blocks on MVIP and H.100/H.110 compliant switching devices. The switch block is controlled by the Switching service functions to make or break connections, send patterns, sample data, query, reset, configure, and run diagnostics on different parts of the MVIP switching device.
- Is based on the MVIP-95 and CT bus device driver standards. It can be used to access MVIP-95 and MVIP-90 device drivers.

NMS Communications also offers the Point-to-Point Switching service that provides an application program interface (API) for making switch connections between boards connected by a telephony bus without having to specify the intervening timeslots. The Point-to-Point Switching service maintains an internal database that records the topology of the switching configuration for the system and the state of all the timeslots. For more information, refer to the *Point-to-Point Switching Service Developer's Reference Manual*.

You need to understand the following characteristics of the Natural Access Switching service as you prepare to create an application:

- Switch handles
- Terminus
- MVIP-95 switch block model

Switch handles

Many functions take or return a switch handle. A switch handle identifies an open MVIP switching device.

To access an MVIP switching device, call **swiOpenSwitch** to retrieve a switch handle. For more information, refer to *Opening a switch handle* on page 17.

swiCloseSwitch releases a switch handle. For more information, refer to *Closing a switch handle* on page 18.

Terminus

A terminus is a single access point to a switch block input or switch block output. Many of the Switching service functions take one or more terminus elements an argument.

A terminus contains a bus, a stream, and a timeslot:

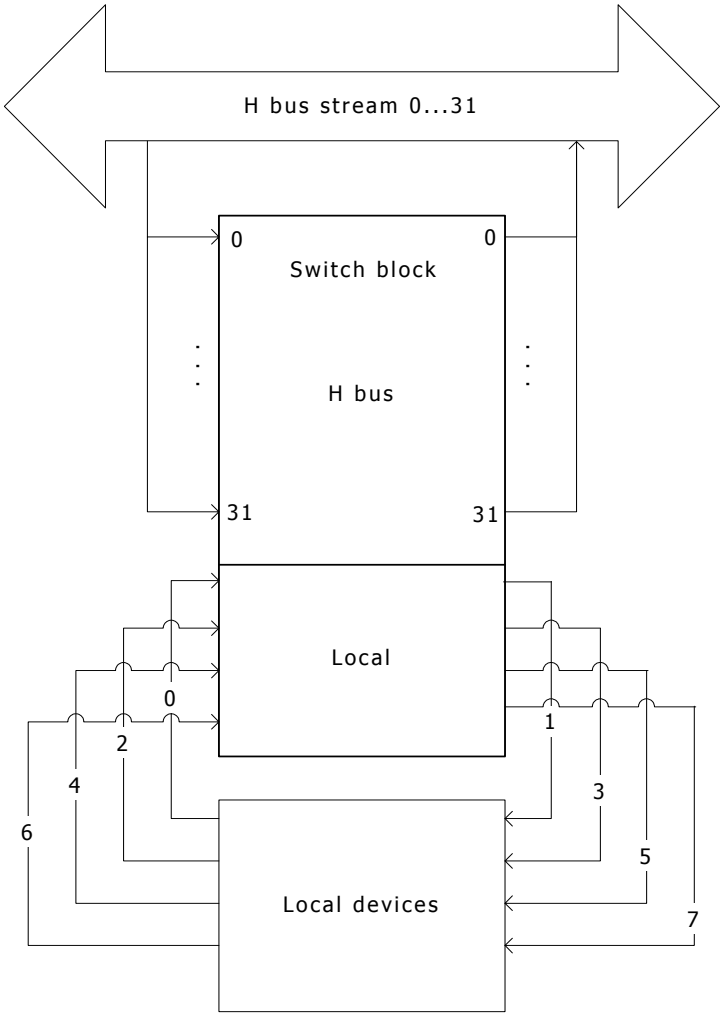
Component	Description
Bus	Specifies the interface point of the switch block. Devices can reside directly on the telephony bus. Devices can also reside on a board's local bus and may require a switch block to access the telephony bus.
Stream	Specifies a grouping of timeslots that usually corresponds to a particular bit-stream of TDM (time-division multiplexed) serial data on an individual track or wire of a bus.
Timeslot	Specifies a particular 64 kbit/second subdivision of a TDM bus stream. Timeslots number from zero (0) to ***n***, where ***n*** is stream-dependent.

MVIP-95 switch block model

The H.100/H.110 bus has 32 data streams. The MVIP-95 switch model was created to accommodate the complete set of streams (0 to 31).

The MVIP-95 switch model is used with H.100/H.110 busses. The MVIP-95 model is based on the premise that a given stream number corresponds to the same physical wire on both sides of the switch block.

As shown in the following illustration, MVIP-95 uses one number for each bus signal, regardless of the side of the switch block. In MVIP-95, bus signals are numbered sequentially starting at 0, allowing for future expansion of the switch capacity without renumbering.



In MVIP-95, local devices are connected to a logical bus called a local bus. The streams they are connected to (either the H bus or the local) are numbered sequentially starting from 0 (zero). Therefore, you must explicitly specify the bus (the H bus or local) when referring to a switch block input or output. MVIP-95 switching commands use a new data structure called a terminus that contains a bus specifier, as well as a stream number and a timeslot number, to refer to a switch block input and output.

Setting up the Natural Access environment

Before calling functions from the Switching library, set up the Natural Access environment by performing the following steps:

Step	Action
1	Initialize the Natural Access application.
2	Create event queues and contexts.
3	Open services on each context.

To set up a second Natural Access application that shares a context with the first application:

Step	Action
1	Initialize the Natural Access application.
2	Create event queues.
3	Attach to the existing context that is to be shared by the applications.

Initializing Natural Access

Register services in the call to **ctaInitialize** by specifying the service (SWI) and service manager (SWIMGR). Only the services initialized in the call to **ctaInitialize** can be opened by the application. Service managers are dynamic link libraries (DLL) in Windows and shared libraries in UNIX that are linked to the application.

Creating event queues and contexts

After initializing Natural Access, create the event queues and the contexts. Create one or more event queues by calling **ctaCreateQueue** and specifying the service manager to attach to each queue. When you attach or bind a service manager to an event queue, you make that service manager available to the event queue.

To create a context, call **ctaCreateContext** and provide the queue handle (***ctaqueuehd***) returned from **ctaCreateQueue**. All events for services on the context are received in the specified event queue.

ctaCreateContext returns a context handle (***ctahd***). The context handle is supplied by the application when invoking **swiOpenSwitch**.

Refer to the *Natural Access Developer's Reference Manual* for details on the programming models created by the use of contexts and queues.

Opening services

To open services on a context, call **ctaOpenServices** and pass a context handle and a list of service descriptors. The service descriptor specifies the name of the service, service manager, and service-specific arguments.

Linking with the Switching service

The Natural Access Switching service contains two components, the Switching service interface (*swiapi*) and the Switching service implementation (*swimgr*). When building a new Natural Access application that uses the Switching service, link to *swiapi.lib* (under UNIX, *libswiapi.so*). The *swimgr.dll* (under UNIX, *libswimgr.so*) is dynamically loaded at runtime.

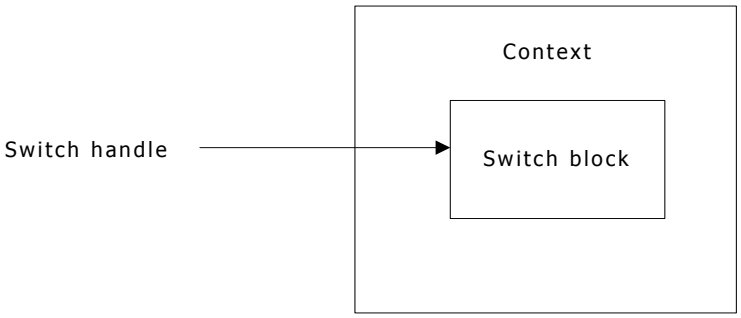
Refer to the *Natural Access Service Writer's Manual* for information about service implementation.

4 Developing applications

Opening a switch handle

A switch handle identifies a switch block on an open MVIP switching device. **swiOpenSwitch** opens a switching device and creates a context for subsequent switching calls. The handle is used to refer to the switch block in the Switching service functions.

A switch handle is always associated with a context. When you specify a switch handle, the context is implied, as shown in the following illustration:



To open a switch handle:

- Specify MVIP-90 mode or MVIP-95 mode
- Enable terminus output state restoration

Natural Access services can share contexts and service object handles. Therefore, you can share open switch handles by using **ctaAttachObject** on a shared context. Refer to the *Natural Access Developer's Reference Manual* for more information.

Specifying MVIP-95 or MVIP-90 mode

Switch handles can be opened in the MVIP-95 mode or in the MVIP-90 mode.

In MVIP-95 mode, the terminus arguments to the Switching service functions are interpreted as MVIP-95 bus, stream, and timeslot values. MVIP-95 is the default mode.

If the underlying device driver is an MVIP-95 device driver, the switch can be opened only in the default mode (MVIP-95).

Caution:	Since MVIP-95 device drivers can be opened only in MVIP-95 mode, NMS recommends writing all applications that may use MVIP-95 device drivers in the future to use MVIP-95 mode even if the application is not currently using MVIP-95 device drivers.

If the underlying device driver is an MVIP-95 device driver or an MVIP-90 device driver and the SWI_MVIP90 constant is used as a parameter to **swiOpenSwitch**, the switch is opened as an MVIP-90 switch. The terminus streams and timeslots are interpreted as MVIP-90 streams and timeslots. The bus field of the terminus is ignored.

If the underlying device driver is an MVIP-90 device driver, and the switch was opened in MVIP-95 mode, the Switching service translates the MVIP-95 terminus parameters into the appropriate MVIP-90 streams and timeslots before passing the commands down to the MVIP-90 device driver.

Enabling terminus output state restoration

When the SWI_ENABLE_RESTORE constant is used as a parameter to **swiOpenSwitch**, the states of the switch block outputs are saved so that they can be restored when the switch handle is closed using **swiCloseSwitch**. If you want to use the SWI_ENABLE_RESTORE mode, a single terminus should not be controlled by more than one switch handle. Otherwise, a terminus affected by your application might be restored to a condition set by another application or to a condition set through a different switch handle in your own application.

swiResetSwitch cannot be called if the switch handle is opened with the SWI_ENABLE_RESTORE constant enabled.

Closing a switch handle

swiCloseSwitch closes a switch handle, releases the associated switching context, and closes the MVIP switching device. Switching calls that are passed to a closed switch handle or an uninitialized switch handle return CTAERR_INVALID_HANDLE.

If the switch handle was opened with the SWI_ENABLE_RESTORE ***flag*** set in **swiOpenSwitch**, switch block outputs affected by switching calls (using this handle) are restored to the state they were in when **swiOpenSwitch** was called.

If switch objects are shared, applications must coordinate switch closing. Switching service functions are synchronous; therefore, no events are generated. When a switch object is shared between Natural Access clients, the clients must take responsibility for notifying peer clients of actions taken on the switch object that affect the peer's continued use of the switch. For example, when a switch object is destroyed within the service manager through a call to **swiCloseSwitch**, peer clients with open handles (SWIHD) to the destroyed switch need to be prepared to take appropriate action.

Each client sharing the switch object has local resources associated with the open switch handle that must be de-allocated when the switch object is closed. When a peer notifies a client that it has closed the switch, the client invokes **ctaDetachObject** to de-allocate these local resources. Refer to the *Natural Access Developer's Reference Manual* for suggested methods of notification between peer clients.

If peer clients choose not to notify one another when they close a switch, there are other ways for the switch handle de-allocation to occur. When a client makes a call with the invalid switch handle, CTAERR_INVALID_HANDLE is returned from the Switching service manager. Upon receiving this error, the client invokes **ctaDetachObject** to de-allocate the local resources.

When **swiCloseSwitch** is called with an invalid switch handle, it automatically destroys the local handle upon being returned CTAERR_INVALID_HANDLE by the service manager. The caller does not need to make an explicit call to **ctaDetachObject** in this situation.

Configuring the clocks

The boards in a CT bus (MVIP-95, H.100, H.110) system are synchronized using clocks. In a system, one board drives the bus clock signals. All other boards reference their clocks from the bus.

On the MVIP-95 bus, H.100 bus, and H.110 bus, the board that drives the bus clocks is the MVIP clock master. In a digital system, the MVIP clock master derives the bus clock signals from the digital trunk's high-quality timing references. In an analog system with no digital telephone network interfaces, a board is configured as the MVIP bus clock master using the on-board oscillator to drive the clock signals.

The H.100 bus has a secondary clock signal that can be configured as a backup clock reference. On the H.100 bus, this clock signal is CT_NETREF. The H.110 bus has two secondary clock signals that can be configured as a backup clock reference: CT_NETREF_1 and CT_NETREF_2.

Note: CT_NETREF and CT_NETREF_1 are the same physical signal.

The H.100 bus and the H.110 bus can be configured to have a backup secondary clock master to which a clock fallback can occur. For more information, refer to *swiConfigBoardClock* on page 36. For further information about configuring clocks on the H.100 bus and H.110 bus, refer to the *ECTF H.100 Hardware Compatibility Specification: CT Bus R1.0* and the *ECTF H.110 Hardware Compatibility Specification: CT Bus R1.0*.

In a multi-chassis MVIP system, all the boards in the PC chassis must be driven by the same clock source. Configure one H bus (H.100 or H.110) in the system as the H bus clock master, driving the H bus clock signals. Then configure the H bus in each PC chassis to drive the MVIP bus in their chassis.

The Switching service provides functions to control the clocks on the underlying MVIP switching device. It provides functions for configuring the:

- Board's clock source (H bus or network).
- Secondary clock signal on the bus.

The following table lists when to use the clock configuration functions available in the Switching service:

To...	Use...
Establish the clock source for an MVIP board	**swiConfigBoardClock**
Define the source of the NETREF clocks on the H.100/H.110 bus	**swiConfigNetrefClock**
Retrieve information regarding the configuration of the board clocking and current status of the clocks	**swiGetBoardClock**
Retrieve the status of a potential TDM bus clock timing reference	**swiGetTimingReference**

Refer to the *H.110 clock configuration example* on page 117 for a sample clock configuration using **swiConfigBoardClock** and **swiConfigNetrefClock**.

Establishing MVIP switching

The Switching service provides functions to control the underlying MVIP switching device. It provides functions for:

- Making a connection between an input terminus and an output terminus.
- Sending a pattern from an output terminus.
- Breaking a connection between an input terminus and an output terminus and stopping the send pattern.
- Sampling data on an input terminus.
- Querying the board about its switching capabilities.

Use the Switching service functions as described in the following table:

To...	Use...
Reset specified switch block outputs to their idle state	**swiDisableOutput**
Retrieve the state of specified switch block outputs	**swiGetOutputState**
Query the device driver and return information about the capabilities of the device driver and the switch controlled by it	**swiGetSwitchCaps**
Connect inputs to outputs	**swiMakeConnection**
Connect inputs to outputs with identical throughput delay for all connections	**swiMakeFramedConnection**
Reset the entire switch block to the idle state	**swiResetSwitch**
Retrieve the current datum values present on specified switch block inputs	**swiSampleInput**
Assert fixed patterns on specified switch block outputs	**swiSendPattern**

Refer to the *Switching application example* on page 105 for information on using Switching to develop a call center application.

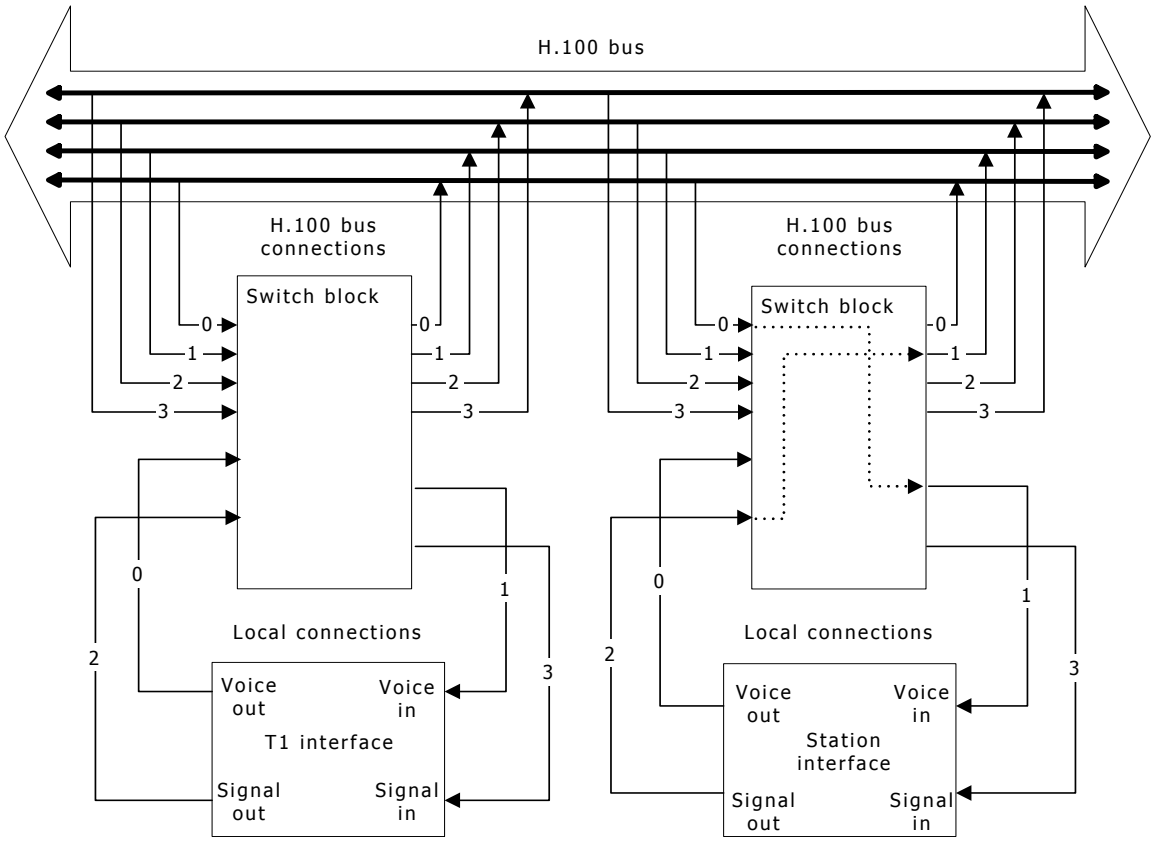
Making connections

The Switching service makes a connection between an input terminus and an output terminus with **swiMakeConnection** or **swiMakeFramedConnection**.

swiMakeConnection connects inputs to the switch block to outputs from the switch block. For example, it can be used to connect the voice paths of an incoming digital line and an operator station so that the caller and the operator can carry on a conversation.

swiMakeFramedConnection is identical to **swiMakeConnection** except that all connections made on the same switch device with this function have the same constant throughput delay. Use **swiMakeFramedConnection** to make a single high bandwidth connection using multiple timeslots where the data is synchronized across the timeslots.

In the following illustration, the operator station voice signals are on local bus streams 0 and 1. They are connected to the MVIP bus on MVIP streams 0 and 1. To connect the voice streams on the incoming digital line to the operator station's voice streams, connect the incoming digital line's voice streams to MVIP bus streams 1 and 0.



The following example code connects the incoming digital line's voice streams to the operator station's voice streams over the MVIP bus:

```
/* Connect voice paths of the station interface and incoming call on the T1 interface */
void myConnectVoice(SWIHD t1hd, DWORD timeslot)
{
    SWI_TERMINUS output, input;

    /* Connect T1 local:0:timeslot to mvip:0:timeslot */
    output.bus = MVIP95_MVIP_BUS;
    output.stream = 0;
    output.timeslot = timeslot;

    input.bus = MVIP95_LOCAL_BUS;
    input.stream = 0;
    input.timeslot = timeslot;

    swiMakeConnection(t1hd, &input, &output, 1);

    /*
     * Make a DUPLEX connection by also connecting
     * mvip:1:timeslot to local:1:timeslot
     */
    output.bus = MVIP95_LOCAL_BUS;
    output.stream = 1;
    output.timeslot = timeslot;

    input.bus = MVIP95_MVIP_BUS;
    input.stream = 1;
    input.timeslot = timeslot;

    swiMakeConnection(t1hd, &input, &output, 1);
}
```

Sending a pattern

Use **swiSendPattern** to send a fixed eight-bit pattern to an output terminus of the switch block for testing and debugging. **swiSendPattern** also allows access to low-level signaling on boards that do not support protocols.

The following code issues a known pattern to MVIP:3:0..31:

```
/* Send a test pattern to 32 timeslots on the CT Bus */
void MySendPattern (SWIHD cxhd)
{
    SWI_TERMINUS outputs[32];
    BYTE patterns[32];
    unsigned count;

    /* SendPattern out to operator interfaces via local:3:timeslot */
    for (count = 0; count < 32; count++)
    {
        outputs[count].bus = MVIP95_MVIP_BUS;
        outputs[count].stream = 3;
        outputs[count].timeslot = (DWORD)count;
        patterns[count] = SWI_A_BIT_ON;
    }
    swiSendPattern(cxhd, patterns, outputs, count);
}
```

Disabling output

If an input terminus is connected to an output terminus or if a pattern is being sent out of the output terminus, **swiDisableOutput** breaks the connection or stops sending the pattern.

For example, to break the connection from the incoming digital line's voice streams to the operator station's voice streams through the MVIP bus, disable the outputs from the incoming digital line's switch block.

Caution:	Disable an output when the connection or pattern is no longer required. Leftover connections or patterns can cause unpredictable behavior in the application.

The following code shows how to break the connection:

```
void myReconfigureT1Line(SWIHD t1hd, DWORD timeslot)
{
    SWI_TERMINUS output;

    /* Disable outputs of switch to mvip:0:timeslot */
    output.bus = MVIP95_MVIP_BUS;
    output.stream = 0;
    output.timeslot = timeslot;

    swiDisableOutput(t1hd, &output, 1);

    /* Disable outputs of switch to local:1:timeslot */
    output.bus = MVIP95_LOCAL_BUS;
    output.stream = 1;
    output.timeslot = timeslot;

    swiDisableOutput(t1hd, &output, 1);
}
```

Sampling data

To determine the data present on a switch block input terminus, use **swiSampleInput**.

For example, to read the bits on an incoming analog line, sample the data on the incoming line's timeslot.

The following example code reads the bits on a specified timeslot:

```
void myPrintInput(SWIHD hd, DWORD bus, DWORD stream, DWORD timeslot)
{
    BYTE data;
    SWI_TERMINUS input;

    input.bus = bus;
    input.stream = stream;
    input.timeslot = timeslot;

    swiSampleInput(hd, &input, &data, 1);

    switch (input.bus)
    {
    case MVIP95_MVIP_BUS:
        printf(" %s", " mvip");
        break;
    case MVIP95_LOCAL_BUS:
        printf(" %s", "local");
        break;

    printf(":%2d:%02d=%02X\n", input.stream, input.timeslot, data);
}
```

Querying switch capabilities

To query the device driver and return information about the capabilities of the device driver and the switch block, use **swiGetSwitchCaps**. For example:

```
void myPrintSwitchCaps(SWIHD hd)
{
    SWI_SWITCHCAPS_ARGS cp;
    SWI_LOCALDEV_DESC *localdevs;
    swiGetSwitchCaps(hd, &cp, NULL, 0);
    localdevs = (SWI_LOCALDEV_DESC *)malloc(
        sizeof(SWI_LOCALDEV_DESC)*cp.numlocalstreams);

    swiGetSwitchCaps(hd, &cp, localdevs, cp.numlocalstreams);

    printf("Driver Software Std. %s Rev. %2.f\n",
        ((cp.swstandard == MVIP95_STANDARD_MVIP95)? "MVIP-95" :
         "MVIP-90"),
        (float)cp.swstdrevision/100.0);

    printf("Hardware Std. %s Rev. %2.f.\n",
           ((cp.hwstandard == MVIP95_STANDARD_HMVIP)? "HMVIP" :
            "MVIP-90"),
           (float)cp.hwstdrevision/100.0);
    printf("Driver Rev. %.2f\n", (float)cp.dvrrevision/100.0);
    printf(" Domain %04X, Routing %04X, Blocking %04X.\n",
           cp.domain, cp.routing, cp.blocking );

    if( cp.numlocalstreams > 0 )
    {
        DWORD i;

        printf("Supports %d local streams:\n\t",
               cp.numlocalstreams );
        for( i=0; i<cp.numlocalstreams; i++ )
            printf( "%2d ", i+16 );
        printf("with\n\t");
        for( i=0; i<cp.numlocalstreams; i++ )
            printf( "%2d ", localdevs[i].timeslots );
        printf("timeslots respectively.\n");
    }
    free(localdevs);
}
```

Configuring stream speed

The following table lists when to use the Switching service stream speed configuration functions:

To...	Use...
Identify all H.100 streams operating at one specified speed	**swiGetStreamsBySpeed**
Configure the speed of one or more streams of the H.100 bus	**swiConfigStreamSpeed**

These functions are used to query and configure the speed of an H.100 stream on an H.100 bus. The speed is specified in millions of bits per second. The functions can also be used to configure parts of the H.100 bus to be compatible with the MVIP bus.

Configuring boards and drivers

Use the following Switching service functions to configure and query boards and drivers:

To...	Use...
Configure stream-specific characteristics of a local device	**swiConfigLocalStream**
Configure stream-specific and timeslot-specific characteristics of a local device	**swiConfigLocalTimeslot**
Obtain vendor-specific information about the board	**swiGetBoardInfo**
Retrieve general and vendor-specific information about the device driver	**swiGetDriverInfo**
Retrieve stream-specific characteristics of a local device	**swiGetLocalStreamInfo**
Retrieve stream-specific and timeslot-specific characteristics of a local device	**swiGetLocalTimeslotInfo**

Note: swiGetLocalStreamInfo, **swiGetLocalTimeslotInfo**, **swiConfigLocalStream**, and **swiConfigLocalTimeslot** take board-specific arguments. Refer to the board-specific documentation for the arguments to pass to these functions.

The following code shows how to use **swiConfigLocalTimeslot** to access the carrier function:

```
#include "swidef.h"  /*  Natural Access Switching service                        */
#include "mvip95.h"  /*  MVIP-95 definitions                                     */
#include "nmshw.h"   /*  NMS hardware-specific definitions                       */
*/
DWORD mySetReceiveGain ( SWIHD swihd, SWI_TERMINUS terminus, INT32 gain_dB )
{
     SWI_LOCALTIMESLOT_ARGS  args;
     NMS_LINE_GAIN_PARMS     device ;

     args.localstream      = terminus.stream ;
     args.localtimeslot    = terminus.timeslot ;
     args.deviceid         = MVIP95_ANALOG_LINE_DEVICE ;
     args.parameterid      = MVIP95_INPUT_GAIN ;

     device.gain  =  gain_dB * 1000  ;

     return swiConfigLocalTimeslot (
               /* Natural Access switch handle             */ swihd,
               /* target device and config item            */ & args,
               /* buffer (defined by parameterid)          */ (void*) & device,
               /* buffer size in bytes                     */ sizeof(device));

}
```

5 Function summary

Switching device functions

Use the following functions to open and close switching devices and obtain information about the switch block:

Function	Description
swiCloseSwitch	Closes the switching device and invalidates the specified switch handle.
swiGetSwitchCaps	Queries the device driver and returns information about the capabilities of the device driver and the switch controlled by it.
swiOpenSwitch	Opens the switching device and returns a switch handle.

All functions in the Switching service are synchronous.

Refer to *Opening a switch handle* on page 17 and *Closing a switch handle* on page 18 for more information.

Connection functions

Use the following functions to make connections:

Function	Description
swiDisableOutput	Resets specified switch block outputs to their idle state.
swiGetOutputState	Retrieves the state of specified switch block outputs.
swiMakeConnection	Connects inputs to outputs.
swiMakeFramedConnection	Connects inputs to outputs with identical throughput delay for all connections.
swiResetSwitch	Resets the entire switch block to the idle state.
swiSampleInput	Retrieves the current datum values present on specified switch block inputs.
swiSendPattern	Asserts fixed patterns on specified switch block outputs.

All functions in the Switching service are synchronous.

Refer to *Disabling output* on page 23 and *Making connections* on page 20 for more information.

Clock functions

The Switching service allows you to control the clocks on the underlying MVIP switching device. It provides functions for configuring the

- Board's clock source (H bus or network).
- Secondary clock signal on the bus.
- Netref clock source on the H.100/H.110 bus.

It also provides functions to retrieve clock configuration, timing reference, and status information.

The following table describes the clock configuration functions:

Function	Description
swiConfigBoardClock	Establishes the clock source for an MVIP board.
swiConfigNetrefClock	Defines the source of the NETREF clocks on the H.100/H.110 bus.
swiGetBoardClock	Retrieves information regarding the configuration of the board clocking and current status of the clocks.
swiGetTimingReference	Retrieves the status of a potential TDM bus clock timing reference.

All functions in the Switching service are synchronous.

For more information, refer to *Configuring the clocks* on page 19.

Stream speed functions

Use the following functions to obtain and control the speed of an H.100 stream on an H.100 bus:

Function	Description
swiConfigStreamSpeed	Configures the speed of one or more streams of the H.100 bus.
swiGetStreamsBySpeed	Identifies all H.100 streams operating at one specified speed.

Note: The H.110 bus is specified for 8 MHz streams only.

All functions in the Switching service are synchronous.

Refer to *Configuring stream speed* on page 24 for more information.

Board and driver functions

Use the following functions to configure devices on local streams and local timeslots, obtain information about the board and the driver, and obtain information about devices on a local stream and a local timeslot:

Function	Description
swiConfigLocalStream	Configures stream-specific characteristics of a local device.
swiConfigLocalTimeslot	Configures stream-specific and timeslot-specific characteristics of a local device.
swiGetBoardInfo	Retrieves information about the board controlled by the MVIP device driver.
swiGetDriverInfo	Retrieves general and vendor-specific information about the device driver.
swiGetLocalStreamInfo	Retrieves stream-specific characteristics of a local device.
swiGetLocalTimeslotInfo	Retrieves stream-specific and timeslot-specific characteristics of a local device.

All functions in the Switching service are synchronous.

Refer to *Configuring boards and drivers* on page 25 for more information.

MVIP device driver functions

Use the following functions to directly access the MVIP device driver:

Function	Description
swiCallDriver	Makes a direct call to the MVIP device driver.
swiGetLastError	Gets the last MVIP device error on the switch handle.

All functions in the Switching service are synchronous.

6 Function reference

Using the function reference

This section provides an alphabetical reference to the Switching service functions. A typical function includes:

Prototype	The prototype is followed by a list of the function arguments. Data types include: • WORD (16-bit unsigned) • DWORD (32-bit unsigned) • INT16 (16-bit signed) • INT32 (32-bit signed) • BYTE (8-bit unsigned) If a function argument is a data structure, the complete data structure is defined.
Return values	The return value for a function is either SUCCESS or an error code. Refer to the *Alphabetical error summary* on page 101 for a list of all errors returned by the Switching service functions.
Example	Example functions that start with *Demo* are excerpts taken from the demonstration code that is shipped with the product. Example functions that start with *my* are excerpts taken from sample application programs. The notation /* ... */ indicates additional code, which is not shown.
Details	Additional information about the function.
See also	A list of related functions.
Example	Example functions that start with *my* are excerpts taken from sample application programs. The notation /* ... */ indicates additional code, which is not shown.

swiCallDriver

Makes a direct call to the MVIP device driver.

Prototype

DWORD **swiCallDriver** (SWIHD ***swihd***, DWORD ***command***, DWORD ****args***, DWORD ***size***, DWORD ****errorcode***)

Argument	Description
swihd	Switch handle.
command	MVIP driver command to call.
args	Pointer to the arguments to the driver call.
size	Size, in bytes, of the arguments to the driver call.
errorcode	Pointer to the device driver return code.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
CTAERR_SVR_COMM	Communication error in the server environment.
SWIERR_CONNECTION_NOT_SUPPORTED	Switch block does not support the requested connection. A permanent limitation in the switch block prevents the connection.
SWIERR_DEVICE_ERROR	MVIP device driver encountered an error while using the services of another device driver.
SWIERR_DLL_INVALID_DEVICE	Dynamic link library (DLL) could not find the requested device.
SWIERR_INTERNAL_CONFLICT	Switch component of a switch matrix conflicts with another switch component. (The state of the switch matrix is ambiguous.)
SWIERR_INVALID_CLOCK_PARM	Value of a clock configuration parameter is invalid.
SWIERR_INVALID_COMMAND	Device driver does not support the requested operation.
SWIERR_INVALID_MINOR_SWITCH	Value of the switch parameter is invalid.
SWIERR_INVALID_MODE	Value of the mode parameter is incorrect or the device driver does not support the mode.
SWIERR_INVALID_PARAMETER	Value of a parameter that does not have its own error code is invalid.
SWIERR_INVALID_SPEED	Value of the speed parameter is out of range.
SWIERR_INVALID_STREAM	Value of the stream parameter is out of range.
SWIERR_INVALID_TIMESLOT	Value of the timeslot parameter is out of range.
SWIERR_MISSING_PARAMETER	Insufficient number of parameters were provided for the driver to complete the command.

Return value	Description
SWIERR_MVIP_BUS_NOT_ENABLED	Switching command was called on a switch block on which the MVIP bus was not enabled.
SWIERR_NO_PATH	Device driver is unable to complete the connection because there is blocking or some other temporary switch limitation encountered.
SWIERR_NOT_CONFIGURABLE	Device does not support configuration of the parameters and/or values requested.
SWIERR_SWITCH_VERIFY_ERROR	Verification of the switch operation failed.
SWIERR_UNSUPPORTED_MODE	Mode is not supported by either the device driver or the hardware below the driver.

Details

swiCallDriver is an escape mechanism that allows the application to make a direct call to the underlying MVIP device driver. The error codes returned by the device driver are returned to the application in the ***errorcode*** argument.

This function does no error checking on the parameters or on the MVIP command code. Elements affected by this function are not restored even if the SWI_ENABLE_RESTORE flag is set in **swiOpenSwitch**.

Note: All commands to the drivers are synchronous; that is, they do not return until the driver completes the command. Some functions take a long time to return. When this happens, all other activity on the context is blocked until the function returns. It is advisable to create a new thread, a new context, and a new switch handle to use a blocking function on a switch handle.

For more information about the MVIP driver command, refer to the *MVIP-95 Device Driver Standard Manual*.

Example

```
/* Supervision Monitoring using MVIP-95 Driver Extensions */
int MonitorSupervision(SWIHD hd)
{
    struct start_supv_parms supvparms;
    struct wait_supv_parms waitparms;
    LONG SupvId;
    DWORD errorcode;

    /* Enable supervision on signalling stream 17, timeslot 0 (MVIP-95) */
    supvparms.stream = 17;
    supvparms.slot = 0;

    /* Wait for hangup */
    supvparms.target_pattern = SWI_A_BIT_OFF;

    /* Assert hangup from our side */
    supvparms.output_pattern = SWI_A_BIT_OFF + SWI_B_BIT_OFF;
    supvparms.qualify_time = 100;

    /* Use MVIP-95 supervision monitoring command */
    swiCallDriver(hd, START_SUPV, (void *)&supvparms,
                  sizeof(supvparms), &errorcode);
    if (errorcode != SUCCESS)
    {
        fprintf(stderr, "** MVIP driver error code: %d\n", errorcode);
        return -1;
    }

    SupvId = supvparms.handle_ret;

    /* Wait until target condition is detected, no time limit on wait */
    waitparms.handle = SupvId;
    waitparms.timeout = -1L;

    /* Use MVIP-95 supervision monitoring command */
    swiCallDriver(hd, WAIT_SUPV, (void *)&waitparms,
                  sizeof(waitparms), &errorcode);
    if (errorcode != SUCCESS)
    {
        fprintf(stderr, "** MVIP driver error code: %d\n", errorcode);
        return -1;
    }
    return 0;
}
```

swiCloseSwitch

Closes a switching device and invalidates the specified switch handle.

Prototype

DWORD **swiCloseSwitch** (SWIHD ***swihd***)

Argument	Description
swihd	Switch handle.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
CTAERR_SVR_COMM	Communication error in the server environment.

Details

swiCloseSwitch frees memory associated with the open switch handle and invalidates the handle. If the switch handle was opened with the SWI_ENABLE_RESTORE ***flag*** set in **swiOpenSwitch**, switch block outputs affected by switching calls (using this handle) are restored to the state they were in when **swiOpenSwitch** was called.

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

For more information, refer to *Closing a switch handle* on page 18.

Example

```
void myApplicationShutdown(SWIHD hd[], unsigned count)
{
    unsigned i;

    for (i = 0; i < count; i++)
    {
        swiCloseSwitch(hd[i]);
    }
}
```

swiConfigBoardClock

Establishes the clock source for an MVIP board.

Prototype

DWORD **swiConfigBoardClock** (SWIHD ***swihd***, SWI_CLOCK_ARGS ****args***)

Argument	Description
swihd	Switch handle.

args	Pointer to the SWI_CLOCK_ARGS clock parameter structure for configuring the MVIP board clock:
	`typedef struct`
	`{`
	`  DWORD size;`
	`  DWORD clocktype;`
	`  DWORD clocksource;`
	`  DWORD network;`
	`  union`
	`  {`
	`  struct`
	`  {`
	`  DWORD clockmode;`
	`  DWORD autofallback;`
	`  DWORD netrefclockspeed;`
	`  DWORD fallbackclocksource;`
	`  DWORD fallbacknetwork;`
	`  }h100;`
	`  } ext; /* specific extensions per board based on clocktype */`
	`} SWI_CLOCK_ARGS;`
	Refer to the Details section for a description of these fields.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_ERROR	Underlying driver retrieved an unrecognized error. Call **swiGetLastError** to retrieve the MVIP device error code.
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
CTAERR_SVR_COMM	Communication error in the server environment.
SWIERR_INVALID_CLOCK_PARM	Value of a clock configuration parameter is invalid.
SWIERR_INVALID_PARAMETER	Board-specific clock parameter value is invalid.

Details

swiConfigBoardClock establishes the clock source for a CT bus board. This function supports clock configuration for all standard MVIP boards, H.100/H.110, and future generations of CT bus boards.

Note: Call **swiConfigBoardClock** before calling **swiConfigSec8KClock** for the first time on a particular switch handle, to prevent the clock source from changing to OSC.

The size, clocktype, clocksource, and network fields of the SWI_CLOCK_ARGS structure are identical across all board types:

Field	Description
size	Number of bytes contained in the structure. If size is less than the size of SWI_CLOCK_ARGS, the command uses only the number of bytes specified by size. If size is greater than the size of SWI_CLOCK_ARGS, size is set to the size of SWI_CLOCK_ARGS, and the command uses this number of bytes.
clocktype	MVIP standard to which clocking on the board applies. Acceptable clocktype values are: MVIP95_STD_CLOCKING MVIP95_H100_CLOCKING
clocksource	Where the clock reference originates. Acceptable values for clocksource are: MVIP95_SOURCE_INTERNAL MVIP95_SOURCE_SEC8K MVIP95_SOURCE_MVIP Additional values for clocksource for H.100/H.110 boards are: MVIP95_SOURCE_NETWORK MVIP95_SOURCE_H100_A MVIP95_SOURCE_H100_B MVIP95_SOURCE_H100_NETREF MVIP95_SOURCE_H100_NETREF_1 MVIP95_SOURCE_H100_NETREF_2
network	Device source for the MVIP clock signals. Acceptable values for network are 1 to ***n*** where ***n*** is the number of devices on the specified board capable of being a clock source. A value for network is valid only when clocksource is equal to MVIP95_SOURCE_NETWORK.

The extension section of the SWI_CLOCK_ARGS structure is a union that currently has a structure for H.100/H.110 boards. Further evolution of MVIP and CT bus standardization work will modify this part of the SWI_CLOCK_ARGS structure.

The following table describes the fields for H.100 and H.110 boards:

Field	Descriptions
clockmode	Board's control of the H.100/H.110 clocks. Acceptable values for clockmode are: MVIP95_H100_SLAVE MVIP95_H100_MASTER_A MVIP95_H100_MASTER_B MVIP95_H100_STAND_ALONE
autofallback	Whether the board is to automatically switch to fallback mode and become a slave to the alternate H.100/H.110 clock. Acceptable values for autofallback are: MVIP95_H100_DISABLE_AUTO_FB MVIP95_H100_ENABLE_AUTO_FB
netrefclockspeed	Speed of the NETREF clock signal. Acceptable values are: MVIP95_H100_NETREF_8KHZ MVIP95_H100_NETREF_1544MHZ MVIP95_H100_NETREF_2048MHZ
fallbackclocksource	Clock reference to be used when an automatic clock fallback occurs. This field is ignored unless autofallback is set to MVIP95_H100_ENABLE_AUTO_FB. Acceptable values are the same as for clocksource.
fallbacknetwork	On-board source of an external network timing for the fallback reference when there are multiple external network connections to the board and fallbackclocksource is set to MVIP95_SOURCE_NETWORK. Acceptable values are: 1..***n*** where ***n*** is the number of devices on the specified board capable of being a clock source.

Refer to *Configuring the clocks* on page 19 for more information.

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

See also

swiGetBoardClock, **swiGetTimingReference**, **swiOpenSwitch**

Example

```
void myAGT1ClockFallback(SWIHD swihd)
{
    SWI_CLOCK_ARGS boardclock;

    /* Make AG-T1 sync off SEC8K */
    boardclock.size = sizeof(SWI_CLOCK_ARGS);
    boardclock.clocktype = MVIP95_STD_CLOCKING;
    boardclock.clocksource = MVIP95_SOURCE_SEC8K;
    swiConfigBoardClock(t1hd, &boardclock);
}
```

swiConfigLocalStream

Configures the stream-specific characteristics of a local device.

Prototype

DWORD **swiConfigLocalStream** (SWIHD ***swihd***, SWI_LOCALSTREAM_ARGS ***args***, void ****buffer***, unsigned ***size***)

Argument	Description
swihd	Switch handle.

args	Pointer to a SWI_LOCALSTREAM_ARGS structure for configuring a device associated with a local stream:
	`typedef struct`
	`{`
	`  DWORD localstream;`
	`  DWORD deviceid;`
	`  DWORD parameterid;`
	`} SWI_LOCALSTREAM_ARGS;`
	Refer to the Details section for a description of these fields.

buffer	Pointer to stream-specific information interpreted by the device driver.
size	Size of ***buffer***, in bytes.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_ERROR	Underlying driver retrieved an unrecognized error. Call **swiGetLastError** to retrieve the MVIP device error code.
CTAERR_FUNCTION_NOT_AVAIL	Underlying driver does not support the configuration of stream-specific characteristics of a local device.
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
SWIERR_INVALID_PARAMETER	Either the parameters passed in ***buffer*** are invalid for the deviceid or the parameterid, or the deviceid/parameterid combination is not supported, or configuration of the specific local stream is not supported.
CTAERR_SVR_COMM	Communication error in the server environment.

Details

swiConfigLocalStream configures stream-specific characteristics of a local device. The content of the buffer portion of the call contains the configuration information and is vendor dependent and device dependent.

CG boards do not support **swiConfigLocalStream**.

The SWI_LOCALSTREAM_ARGS structure contains the following fields:

Field	Description
localstream	Stream to be configured on the local bus.
deviceid	Device type on the local stream. The deviceid is hardware dependent. Acceptable values for deviceid are: MVIP95_T1_TRUNK_DEVICE MVIP95_E1_TRUNK_DEVICE MVIP95_ANALOG_LINE_DEVICE MVIP95_CONFERENCE_DEVICE In addition to these values, the device vendor can define device identifiers specific to their products. Refer to the device-specific documentation for these values.
parameterid	Data item for which configuration is to be performed. This value is vendor specific and device driver specific. The combination of the deviceid and the parameterid specify the part of the device to configure.

For more information, refer to *Configuring boards and drivers* on page 25. Refer to the installation and developer's manual for the board you are using for board-specific information.

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

See also

swiConfigLocalTimeslot, **swiGetBoardInfo**, **swiGetDriverInfo**, **swiGetLocalStreamInfo**, **swiGetLocalTimeslotInfo**, **swiOpenSwitch**

swiConfigLocalTimeslot

Configures the stream-specific and timeslot-specific characteristics of a local device.

Prototype

DWORD **swiConfigLocalTimeslot** (SWIHD ***swihd***, SWI_LOCALTIMESLOT_ARGS ***args***, void ****buffer***, unsigned ***size***)

Argument	Description
swihd	Switch handle.

args	Pointer to a SWI_LOCALTIMESLOT_ARGS structure for configuring a device associated with a local stream and timeslot:
	`typedef struct`
	`{`
	`  DWORD localstream;`
	`  DWORD localtimeslot;`
	`  DWORD deviceid;`
	`  DWORD parameterid;`
	`} SWI_LOCALTIMESLOT_ARGS;`
	Refer to the Details section for a description of these fields.

buffer	Pointer to timeslot specific information interpreted by the device driver.
size	Size of ***buffer***, in bytes.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_ERROR	Underlying driver retrieved an unrecognized error. Call **swiGetLastError** to retrieve the MVIP device error code.
CTAERR_FUNCTION_NOT_AVAIL	Underlying driver does not support the configuration of timeslot-specific characteristics of a local device.
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
SWIERR_INVALID_PARAMETER	Either the parameters passed in ***buffer*** are invalid for the deviceid or the parameterid, or the deviceid/parameterid combination is not supported, or configuration of the specific local timeslot is not supported.
CTAERR_SVR_COMM	Communication error in the server environment.

Details

swiConfigLocalTimeslot configures stream-specific and timeslot-specific characteristics of a local device. The content of the buffer portion of the call contains the configuration information and is vendor dependent and device dependent.

The SWI_LOCALTIMESLOT_ARGS structure contains the following fields:

Field	Description
localstream	Stream to be configured on the local bus.
localtimeslot	Timeslot to be configured on the local bus.
deviceid	Device type on the local bus. The deviceid is hardware dependent. Acceptable values for deviceid are: MVIP95_T1_TRUNK_DEVICE MVIP95_E1_TRUNK_DEVICE MVIP95_ANALOG_LINE_DEVICE MVIP95_CONFERENCE_DEVICE In addition to these values, the device vendor can define device identifiers specific to their products. Refer to the device-specific documentation for these values.
parameterid	Data item for which configuration is to be performed. This value is vendor specific and device driver specific. The combination of the deviceid and the parameterid specify the part of the device to configure.

Refer to *Configuring boards and drivers* on page 25 for more information.

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

Refer to the documentation for the board you are using for board-specific information.

See also

swiConfigLocalStream, **swiGetBoardInfo**, **swiGetDriverInfo**, **swiGetLocalStreamInfo**, **swiGetLocalTimeslotInfo**, **swiOpenSwitch**

Example

```
void myConfigChannels(SWIHD age1hd)
{
    SWI_LOCALTIMESLOT_ARGS args;
    struct channel_parms cp;
    DWORD i;

    args.localstream = 0;
    args.deviceid = MVIP95_T1_TRUNK_DEVICE;
    args.parameterid = CONFIG_CHANNEL;

    cp.size = sizeof(struct channel_parms);
    cp.invert = 0;
    cp.loopback = 0;
    cp.robbedbit = 1;

    for (i = 0; i < 24; i++)
    {
        args.localtimeslot = i;
        cp.trunk = i;
        swiConfigLocalTimeslot(age1hd, &args, &cp, cp.size);
    }
}
```

swiConfigNetrefClock

Defines the source of the NETREF clocks on the H.100/H.110 bus.

Prototype

DWORD **swiConfigNetrefClock** (SWIHD ***swihd***, SWI_NETREF_CLOCK_ARGS ****args***)

Argument	Description
swihd	Switch handle.

args	NETREF clock configuration parameters:
	`typedef struct`
	`{`
	`  DWORD size;`
	`  DWORD network;`
	`  DWORD netrefclockmode;`
	`  DWORD netrefclockspeed;`
	`} SWI_NETREF_CLOCK_ARGS;`
	Refer to the Details section for a description of the structure and acceptable values.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_ERROR	Underlying driver retrieved an unrecognized error. Call **swiGetLastError** to retrieve the MVIP device error code.
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
CTAERR_SVR_COMM	Communication error in the server environment.
SWIERR_INVALID_CLOCK_PARM	Value of a clock configuration parameter is invalid.
SWIERR_INVALID_PARAMETER	Board-specific clock parameter value is invalid.

Details

swiConfigNetrefClock establishes the source of the NETREF clocks on the H.100/H.110 bus. The SWI_NETREF_CLOCK_ARGS structure contains the following fields:

Field	Description
size	Number of bytes contained in the structure. If size is less than the size of SWI_NETREF_CLOCK_ARGS, the command uses only the number of bytes specified by size. If size is greater than the size of SWI_NETREF_CLOCK_ARGS, size is set to the size of SWI_NETREF_CLOCK_ARGS, and the command uses this number of bytes.
network	Acceptable values are 1 to ***n*** where ***n*** is the number of devices on the board. Some devices are not capable of being a source for NETREF. Refer to the device-specific hardware documentation for the devices that are capable of being a source for NETREF.
netrefclockmode	Acceptable values are: MVIP95_H100_NETREF MVIP95_H100_NETREF_1 MVIP95_H100_NETREF_2
netrefclockspeed	Acceptable values are: MVIP95_H100_NETREF_8KHZ MVIP95_H100_NETREF_1544MHZ MVIP95_H100_NETREF_2048MHZ MVIP95_H100_NETREF_DISABLED **Note:** If MVIP95_H100_NETREF_DISABLED is used, the board stops driving the NETREF specified in the netrefclockmode field.

Refer to *Configuring the clocks* on page 19 for more information.

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

See also

swiConfigBoardClock, **swiGetBoardClock**, **swiGetTimingReference**, **swiOpenSwitch**

Example

```
void myNetrefClockInit(SWIHD t1hd)
{
    SWI_NETREF_CLOCK_ARGS netrefclock;

    netrefclock.size = sizeof(SWI_NETREF_CLOCK_ARGS);
    netrefclock.network = 1;
    netrefclock.netrefclockmode = MVIP95_H100_NETREF;
    netrefclock.netrefclockspeed = MVIP95_H100_NETREF_8KHZ;
    swiConfigNetrefClock(t1hd, &netrefclock);
}
```

swiConfigSec8KClock

Defines the source of the secondary 8 kHz clock on the bus.

Prototype

DWORD **swiConfigSec8KClock** (SWIHD ***swihd***, DWORD ***source***, DWORD ***network***)

Argument	Description
swihd	Switch handle.
source	Clock source for the secondary 8 kHz signal on a board. Refer to the Details section for acceptable values.
network	Device source of the secondary 8 kHz signal when ***source*** is MVIP95_SOURCE_NETWORK. Refer to the Details section for acceptable values.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_ERROR	Underlying driver retrieved an unrecognized error. Call **swiGetLastError** to retrieve the MVIP device error code.
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
CTAERR_SVR_COMM	Communication error in the server environment.
SWIERR_INVALID_CLOCK_PARM	Value of a clock configuration parameter is invalid.
SWIERR_INVALID_PARAMETER	Board-specific clock parameter value is invalid.

Details

swiConfigSec8KClock establishes the source of the secondary 8 kHz clock on the bus.

Note: Call **swiConfigBoardClock** before calling **swiConfigSec8KClock** for the first time on a particular switch handle, to prevent the clock source from changing to OSC.

Acceptable values for ***source*** are:

- MVIP95_SOURCE_DISABLE
- MVIP95_SOURCE_INTERNAL
- MVIP95_SOURCE_NETWORK

Acceptable values for ***network*** are 1 to ***n*** where ***n*** is the number of devices on the board. Some devices are not capable of being a source for SEC8K. Refer to the device-specific hardware documentation for the devices that are capable of being a source for SEC8K.

Refer to *Configuring the clocks* on page 19 for more information.

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

See also

swiConfigNetrefClock, **swiGetBoardClock**, **swiGetTimingReference**, **swiOpenSwitch**

Example

```
void myT1ClockInit(SWIHD swihd)
{
    SWI_CLOCK_ARGS boardclock;

    /* Make board sync off MVIP bus clock */
    boardclock.size = sizeof(SWI_CLOCK_ARGS);
    boardclock.clocktype = MVIP95_STD_CLOCKING;
    boardclock.clocksource = MVIP95_SOURCE_MVIP;
    swiConfigBoardClock(swihd, &boardclock);

    /* Make board's network 1 provide the source of the Sec8K signal */
    swiConfigSec8KClock(swihd, MVIP95_SOURCE_NETWORK, 1);
}
```

swiConfigStreamSpeed

Configures the speed of one or more streams of the H.100 bus.

Prototype

DWORD **swiConfigStreamSpeed** (SWIHD ***swihd***, DWORD ***speed***, DWORD ***streams[]***, unsigned ***count***)

Argument	Description
swihd	Switch handle.
speed	Specifies in millions of bits per second the capacity of one or more streams. Refer to the Details section for acceptable values.
streams	Array of one or more streams that specify the H.100 streams to be configured to the specified speed.
count	Number of ***streams*** to be configured.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_ERROR	Underlying driver retrieved an unrecognized error. Call **swiGetLastError** to retrieve the MVIP device error code.
CTAERR_FUNCTION_NOT_AVAIL	Underlying driver does not support the configuration of stream-specific characteristics of a local device. This is usually the case if the driver is MVIP-90.
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
CTAERR_SVR_COMM	Communication error in the server environment.
SWIERR_INVALID_SPEED	Specified stream speed is not supported.
SWIERR_INVALID_STREAM	Stream speed configuration is not supported on one or more of the specified streams.

Details

swiConfigStreamSpeed configures the speed of one or more streams of the H.100 bus. This command is specific to MVIP-95. Calling this function on an MVIP-90 driver returns CTAERR_FUNCTION_NOT_AVAIL.

Acceptable values for ***speed*** are:

- MVIP95_2MBPS_STREAM_SPEED
- MVIP95_4MBPS_STREAM_SPEED
- MVIP95_8MBPS_STREAM_SPEED

Refer to *Configuring stream speed* on page 24 for more information.

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

See also

swiGetStreamsBySpeed

Example

```
void myConfigHMVIPtoMVIP90(SWIHD hd, DWORD streams[], unsigned count)
{
    /* Configure H.100 streams to be compatible with MVIP-90 streams */
    swiConfigStreamSpeed(hd, MVIP95_2MBPS_STREAM_SPEED, streams, count);
}
```

swiDisableOutput

Resets the specified switch block outputs to their idle state.

Prototype

DWORD **swiDisableOutput** (SWIHD ***swihd***, SWI_TERMINUS ***output[]***, unsigned ***count***)

Argument	Description
swihd	Switch handle.

output	Array of terminus structures containing the outputs to disable:
	`typedef struct`
	`{`
	`  DWORD bus;`
	`  DWORD stream;`
	`  DWORD timeslot;`
	`} SWI_TERMINUS;`
	Refer to *swiMakeConnection* on page 76 for a description of these fields.

count	Number of elements in the ***output*** array.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_ERROR	Underlying driver retrieved an unrecognized error. Call **swiGetLastError** to retrieve the MVIP device error code.
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
CTAERR_SVR_COMM	Communication error in the server environment.
SWIERR_INVALID_STREAM	One or more of the specified streams is invalid.
SWIERR_INVALID_TIMESLOT	One or more of the specified timeslots is invalid.

Details

swiDisableOutput resets specified switch block outputs to their idle state.

If the switch block output is on the MVIP bus, it is set to the high impedance state. If the switch block output is on the local bus, the timeslots on the local bus are set to a known state. (The driver vendor must publish these states in its customer documentation.)

Note: Under Solaris, the upper limit for the number of terminuses that can be batched in one call is 32.

Refer to *Disabling output* on page 23 for more information.

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

See also

swiCloseSwitch, **swiGetOutputState**, **swiGetSwitchCaps**, **swiMakeFramedConnection**, **swiOpenSwitch**, **swiResetSwitch**, **swiSampleInput**, **swiSendPattern**

Example

```
void myDisconnectTerminus(SWIHD swihd, SWI_TERMINUS output)
{
    unsigned count;
    SWI_TERMINUS outputs[6];

    /* Disconnect 6 consecutive timeslots */
    for (count = 0; count < 5; count++)
    {
        outputs[count].bus = output.bus;
        outputs[count].stream = output.stream;
        outputs[count].timeslot = output.timeslot + count;
    }
    swiDisableOutput(swihd, outputs, count);
}
```

swiGetBoardClock

Retrieves the board clocking configuration and the current status of the clocks.

Prototype

DWORD **swiGetBoardClock** (SWIHD ***swihd***, DWORD ***clocktype***, SWI_QUERY_CLOCK_ARGS ****args***, unsigned ***size***)

Argument	Description
swihd	Switch handle.
clocktype	Specifies the MVIP standard to which clocking on the board applies. Refer to the Details section for acceptable values.

args	Pointer to a clock parameter structure for querying the CT bus board clock:
	`typedef struct`
	`{`
	`  DWORD size;`
	`  DWORD clocktype;`
	`  DWORD clocksource;`
	`  DWORD network;`
	`  union`
	`  {`
	`  struct`
	`  {`
	`  DWORD mc1clockmode;`
	`  DWORD autofallback;`
	`  DWORD fallbackoccurred;`
	`  } mc1;  /* only for MC1*/`
	`  struct`
	`  {`
	`  DWORD clockmode;`
	`  DWORD autofallback;`
	`  DWORD fallbackoccurred;`
	`  DWORD clockstatus_a;`
	`  DWORD clockstatus_b;`
	`  DWORD clockstatus_netref1;`
	`  DWORD clockstatus_netref2;`
	`  }h100; /* only for h100 */`
	`  } ext; /* extension, specific parts for each board, based on`
	`  clocktype */`
	`} SWI_QUERY_CLOCK_ARGS;`
	Refer to the Details section for a description of these fields.

size	Specifies the number of bytes contained in the structure.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_ERROR	Underlying driver retrieved an unrecognized error. Call **swiGetLastError** to retrieve the MVIP device error code.
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
CTAERR_SVR_COMM	Communication error in the server environment.
SWIERR_INVALID_CLOCK_PARM	Value of a clock configuration parameter is invalid.
SWIERR_INVALID_PARAMETER	Board-specific clock parameter value is invalid.

Details

swiGetBoardClock retrieves information about the configuration of the board clocking and the current status of the clocks for a CT bus board.

H.100/H.110 are supported.

Acceptable values for ***clocktype*** are:

- MVIP95_STD_CLOCKING
- MVIP95_H100_CLOCKING

The clocksource and network fields of the SWI_QUERY_CLOCK_ARGS structure are identical across all board types.

Field	Description
clocksource	Origination of the clock reference. Acceptable values are: MVIP95_SOURCE_INTERNAL MVIP95_SOURCE_NETWORK Additional values for clocksource for H.100/H.110 boards are: MVIP95_SOURCE_H100_A MVIP95_SOURCE_H100_B MVIP95_SOURCE_H100_NETREF MVIP95_SOURCE_H110_NETREF_1 MVIP95_SOURCE_H110_NETREF_2
network	Device source for the MVIP clock signals. Acceptable values for network are 1 to ***n*** where ***n*** is the number of devices on the specified board capable of being a clock source. A value for network is valid only when clocksource is equal to MVIP95_SOURCE_NETWORK.

The fields specific to the H.100/H.110 boards are described in the following table:

Field	Description
clockmode	Board's control of the H.100/H.110 clocks. Acceptable values are: MVIP95_H100_SLAVE MVIP95_H100_MASTER_A MVIP95_H100_MASTER_B MVIP95_H100_STAND_ALONE
autofallback	Whether the board is set to automatically switch to fallback mode and become a slave to the alternate H.100/H.110 clock. Acceptable values are: MVIP95_H100_DISABLE_AUTO_FB MVIP95_H100_ENABLE_AUTO_FB
fallbackoccurred	Wand has fallen back to the secondary reference. Acceptable values are: MVIP95_H100_NO_FALLBACK_OCCURRED MVIP95_H100_FALLBACK_OCCURRED
clockstatus_a	Quality of the A clock master signal. Acceptable values are: MVIP95_CLOCK_STATUS_GOOD MVIP95_CLOCK_STATUS_BAD MVIP95_CLOCK_STATUS_UNKNOWN
clockstatus_b	Quality of the B clock master signal. Acceptable values are: MVIP95_CLOCK_STATUS_GOOD MVIP95_CLOCK_STATUS_BAD MVIP95_CLOCK_STATUS_UNKNOWN
clockstatus_netref1	Quality of the NETREF (H.100) or NETREF_1 (H.110) clock signal. Acceptable values are: MVIP95_CLOCK_STATUS_GOOD MVIP95_CLOCK_STATUS_BAD MVIP95_CLOCK_STATUS_UNKNOWN
clockstatus_netref2	Quality of the NETREF_2 (H.110 only) clock master signal. Acceptable values are: MVIP95_CLOCK_STATUS_GOOD MVIP95_CLOCK_STATUS_BAD MVIP95_CLOCK_STATUS_UNKNOWN

Refer to *Configuring the clocks* on page 19 for more information about querying clocks.

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

See also

swiConfigBoardClock, **swiOpenSwitch**

Example

```
void myGetClock (SWIHD swihd)
{
     SWI_QUERY_CLOCK_ARGS    queryclock;
     DWORD                   clocktype;
     unsigned                size;

     size = sizeof (SWI_QUERY_CLOCK_ARGS);
     swiGetBoardClock  (swihd,  clocktype, &queryclock, size);
}
```

swiGetBoardInfo

Retrieves information about the board controlled by the MVIP device driver.

Prototype

DWORD **swiGetBoardInfo** (SWIHD ***swihd***, SWI_BOARDINFO_ARGS ****args***, unsigned ***size***)

Argument	Description
swihd	Switch handle.

args	Pointer to a SWI_BOARDINFO_ARGS structure for the information about the board controlled by the MVIP driver:
	`typedef struct`
	`{`
	`  BYTE  description [80];`
	`  BYTE  revision [16];`
	`  BYTE  date [12];`
	`  BYTE  vendor [80];`
	`  BYTE  serialnumber [80];`
	`  DWORD  boardid;`
	`  DWORD  base_port_address;`
	`} SWI_BOARDINFO_ARGS;`
	Refer to the Details section for a description of these fields.

size	Size, in bytes, of the ***args*** buffer.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_ERROR	Underlying driver retrieved an unrecognized error. Call **swiGetLastError** to retrieve the MVIP device error code.
CTAERR_FUNCTION_NOT_AVAIL	Underlying driver does not support the configuration of board-specific characteristics of a local device.
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
CTAERR_SVR_COMM	Communication error in the server environment.

Details

swiGetBoardInfo is specific to MVIP-95. **swiGetBoardInfo** retrieves information about the board controlled by the MVIP device driver. The device driver is associated with the switch block handle opened by the user with **swiOpenSwitch**. All BYTE fields are NULL-terminated ASCII strings. With the exception of the date field, there are no restrictions on how ASCII information is represented.

The SWI_BOARDINFO_ARGS structure contains the following fields:

Field	Description
description	Text that provides information about the board.
revision	Version number of the board in a vendor-specific format.
date	Release date of the board. The date format is ***yyyy/mm/dd***.
vendor	Company or organization that created the board.
serialnumber	Vendor-specific ASCII representation of the board's serial number.
boardid	Vendor-specific value used to identify the board. With boardid, an application can determine the actual board type when a single driver supports more than one board type. The board identifier represented by boardid is not unique across all vendors.
base_port_address	Identifies the physical base input/output of the PC bus used by the board. Applications use this information to make the association between the physical hardware and the logical device used by the driver. Not used for PCI boards.

Refer to *Configuring boards and drivers* on page 25 for more information.

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

See also

swiConfigLocalStream, **swiConfigLocalTimeslot**, **swiGetDriverInfo**, **swiGetLocalStreamInfo**, **swiGetLocalTimeslotInfo**

Example

```
void myPrintBoardInfo(SWIHD hd)
{
    SWI_BOARDINFO_ARGS args;
    unsigned size;

    size = sizeof(SWI_BOARDINFO_ARGS);
    swiGetBoardInfo(hd, &args, size);
    printf("%s\n", args.description);
    printf("Revision %s Date %s\n", args.revision, args.date);
    printf("%s\n", args.vendor);
    printf("Board type: %d, Serial No. %s\n", args.boardid, args.serialnumber);
}
```

swiGetDriverInfo

Retrieves general and vendor-specific information about the device driver.

Prototype

DWORD **swiGetDriverInfo** (SWIHD ***swihd***, SWI_DRIVERINFO_ARGS ****args***, unsigned ***size***)

Argument	Description
swihd	Switch handle.

args	Pointer to a SWI_DRIVERINFO_ARGS structure for the information about the device driver:
	`typedef struct`
	`{`
	`  BYTE  description [80];`
	`  BYTE  revision [16];`
	`  BYTE  date [12];`
	`  BYTE  vendor [80];`
	`} SWI_DRIVERINFO_ARGS;`
	Refer to the Details section for a description of these fields.

size	Size, in bytes, of the ***args*** buffer.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_ERROR	Underlying driver retrieved an unrecognized error. Call **swiGetLastError** to retrieve the MVIP device error code.
CTAERR_FUNCTION_NOT_AVAIL	Underlying driver does not support the configuration of driver-specific characteristics of a local device.
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
CTAERR_SVR_COMM	Communication error in the server environment.

Details

swiGetDriverInfo is specific to MVIP-95. **swiGetDriverInfo** retrieves general and vendor-specific information about the device driver. The device driver is associated with the switch block handle opened by the user through **swiOpenSwitch**. All BYTE fields are NULL-terminated ASCII strings. With the exception of the date field, there are no restrictions on how ASCII information is represented.

The SWI_DRIVERINFO_ARGS structure contains the following fields:

Field	Description
description	Information about the device driver.
revision	Version number of the device driver in a vendor-specific format.
date	Release date of the device driver. The date format is ***yyyy***/***mm***/***dd***.
vendor	Company or organization that created the device driver.

Refer to *Configuring boards and drivers* on page 25 for more information.

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

See also

swiConfigLocalStream, **swiConfigLocalTimeslot**, **swiGetBoardInfo**, **swiGetLocalStreamInfo**, **swiGetLocalTimeslotInfo**

Example

```
void myPrintDriverInfo(SWIHD hd)
{
    SWI_DRIVERINFO_ARGS args;
    unsigned size;

    size = sizeof(SWI_DRIVERINFO_ARGS);
    swiGetDriverInfo(hd, &args, size);
    printf("%s\n", args.description);
    printf("Revision %s Date %s\n", args.revision, args.date);
    printf("%s\n", args.vendor);
}
```

swiGetLastError

Retrieves the last MVIP device error on the switch handle.

Prototype

DWORD **swiGetLastError** (SWIHD ***swihd***, DWORD ****errorcode***)

Argument	Description
swihd	Switch handle.
errorcode	Pointer to the code of the device error.

Return values

Return value	Description
SUCCESS	
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
CTAERR_SVR_COMM	Communication error in the server environment.

Details

Call **swiGetLastError** if a Switching service function returned CTAERR_DRIVER_ERROR. **swiGetLastError** retrieves the last error returned by the switching device driver on the specified switch handle.

For a description of the error codes, refer to the *MVIP-95 Device Driver Standard Manual*.

Example

```
void myErrorHandler(SWIHD hd, char *text, DWORD status)
{
    DWORD errorcode, ret;

    fprintf(stderr, "Error (%d): %s", status, text);
    ret = swiGetLastError(hd, &errorcode);
    if (ret == SUCCESS)
        fprintf(stderr, "** MVIP driver error code: %d\n", errorcode);
 }
```

swiGetLocalStreamInfo

Retrieves the stream-specific characteristics of a local device.

Prototype

DWORD **swiGetLocalStreamInfo** (SWIHD ***swihd***, SWI_LOCALSTREAM_ARGS ****args***, void ****buffer***, unsigned ***size***)

Argument	Description
swihd	Switch handle.

args	Pointer to a SWI_LOCALSTREAM_ARGS structure for the configuration information about a device on a specific stream on the local bus:
	`typedef struct`
	`{`
	`  DWORD localstream;`
	`  DWORD deviceid;`
	`  DWORD parameterid;`
	`} SWI_LOCALSTREAM_ARGS;`
	Refer to the Details section for a description of these fields.

buffer	Pointer to a buffer to receive stream-specific information maintained by the device driver.
size	Size of the buffer in bytes.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_ERROR	Underlying driver retrieved an unrecognized error. Call **swiGetLastError** to retrieve the MVIP device error code.
CTAERR_FUNCTION_NOT_AVAIL	Underlying driver does not support the configuration of stream-specific characteristics of a local device.
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
CTAERR_SVR_COMM	Communication error in the server environment.

Details

swiGetLocalStreamInfo retrieves stream-specific characteristics of a local device. The configuration information returned is vendor-specific and device-specific.

CG boards do not support **swiGetLocalStreamInfo**.

The SWI_LOCALSTREAM_ARGS structure contains the following fields:

Field	Description
localstream	Stream to be queried on the local bus.
deviceid	Device type on the local stream. The deviceid is hardware dependent. Acceptable values are: MVIP95_T1_TRUNK_DEVICE MVIP95_E1_TRUNK_DEVICE MVIP95_ANALOG_LINE_DEVICE MVIP95_CONFERENCE_DEVICE In addition to these values, the device vendor can define device identifiers specific to their products. Refer to the device-specific documentation for these values.
parameterid	Data item for which configuration is to be obtained. This value is vendor specific and device driver specific. The combination of the deviceid and the parameterid specify the part of the device to configure.

Refer to *Configuring boards and drivers* on page 25 for more information.

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

See also

swiConfigLocalStream, **swiConfigLocalTimeslot**, **swiGetBoardInfo**, **swiGetDriverInfo**, **swiGetLocalTimeslotInfo**, **swiGetSwitchCaps**

Example

```
void myPrintCarrierStatus(SWIHD swihd)
{
    SWI_LOCALSTREAM_ARGS args;
    struct carrier_status cs;

    args.localstream = 0;
    args.deviceid = MVIP95_T1_TRUNK_DEVICE;
    args.parameterid = CARRIER_STATUS;

    cs.trunk = 0;
    swiGetLocalStreamInfo(swihd, &args, &cs, sizeof(cs));
    printf("Event count: %d\n", cs.event_count);
    printf("Device: %d\n", cs.device);
    if (cs.red_alarm)
        printf("** RED ALARM **\n");
    if (cs.yellow_alarm)
        printf("** YELLOW ALARM **\n");
    if (cs.blue_alarm)
        printf("** BLUE ALARM **\n");
    switch (cs.sync)
    {
    case 0:
        printf("Synchronized\n");
        break;
    case 1:
        printf("Blue Alarm\n");
        break;
    case 2:
        printf("Not Synchronized\n");
        break;
    case 3:
        printf("Super frame not synchronized\n");
        break;
    }
}
```

swiGetLocalTimeslotInfo

Retrieves the stream-specific and timeslot-specific characteristics of a local device.

Prototype

DWORD **swiGetLocalTimeslotInfo** (SWIHD ***swihd***, SWI_LOCALTIMESLOT_ARGS ***args***, void ***buffer***, unsigned ***size***)

Argument	Description
swihd	Switch handle.

args	Pointer to a SWI_LOCALTIMESLOT_ARGS structure for the configuration information about a device on a specific stream and timeslot of the local bus:
	`typedef struct`
	`{`
	`  DWORD localstream;`
	`  DWORD localtimeslot;`
	`  DWORD deviceid;`
	`  DWORD parameterid;`
	`} SWI_LOCALTIMESLOT_ARGS;`
	Refer to the Details section for a description of these fields.

buffer	Pointer to timeslot-specific information maintained by the device driver.
size	Size of ***buffer***, in bytes.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_ERROR	Underlying driver retrieved an unrecognized error. Call **swiGetLastError** to retrieve the MVIP device error code.
CTAERR_FUNCTION_NOT_AVAIL	Underlying driver does not support the configuration of stream-specific characteristics of a local device.
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
CTAERR_SVR_COMM	Communication error in the server environment.

Details

swiGetLocalTimeslotInfo retrieves the stream-specific and timeslot-specific characteristics of a local device. The configuration information returned is vendor specific and device specific.

The SWI_LOCALTIMESLOT_ARGS structure contains the following fields:

Field	Description
localstream	Stream associated with the timeslot to be queried on the local bus.
localtimeslot	Timeslot to be queried on the local bus.
deviceid	Device type on the local stream and timeslot. The deviceid is hardware dependent. Acceptable values are: MVIP95_T1_TRUNK_DEVICE MVIP95_E1_TRUNK_DEVICE MVIP95_ANALOG_LINE_DEVICE MVIP95_CONFERENCE_DEVICE In addition to these values, the device vendor can define device identifiers specific to their products. Refer to the device-specific documentation for these values.
parameterid	Data item for which configuration is to be obtained. This value is vendor and device driver specific. The combination of the deviceid and the parameterid specify the part of the device to configure.

Refer to *Configuring boards and drivers* on page 25 for more information.

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

See also

swiConfigLocalStream, **swiConfigLocalTimeslot**, **swiGetBoardInfo**, **swiGetDriverInfo**, **swiGetLocalStreamInfo**

Example

```
void myGetHybridIds(SWIHD agcxhd, DWORD ids[], unsigned count)
{
    SWI_LOCALTIMESLOT_ARGS args;
    unsigned i;

    args.localstream = 0;
    args.deviceid = MVIP95_ANALOG_LINE_DEVICE;
    args.parameterid = HYBRID_ID;

    for (i = 0; i < count; i++)
    {
        args.localtimeslot = i;
        swiGetLocalTimeslotInfo(agcxhd, &args, &ids[i], sizeof(ids[i]));
    }
}
```

swiGetOutputState

Retrieves the state of the specified switch block outputs.

Prototype

DWORD **swiGetOutputState** (SWIHD ***swihd***, SWI_TERMINUS ***output[]***, unsigned ***mode[]***, BYTE ***pattern[]***, SWI_TERMINUS ***input[]***, unsigned ***count***)

Argument	Description
swihd	Switch handle.

output	Array of terminus structures for which to retrieve the state:
	`typedef struct`
	`{`
	`  DWORD bus;`
	`  DWORD stream;`
	`  DWORD timeslot;`
	`} SWI_TERMINUS;`
	Refer to *swiMakeConnection* on page 76 for a description of these fields.

mode	Array that receives the state of the switch block outputs. Possible ***mode*** values are: MVIP95_DISABLE_MODE MVIP95_PATTERN_MODE MVIP95_CONNECT_MODE MVIP95_FRAMED_CONNECT_MODE Refer to the Details section for more information about ***mode***.
pattern	Array that receives the data values asserted at the switch block outputs if the corresponding ***mode*** is MVIP95_PATTERN_MODE.
input	Array that receives the switch block inputs connected to the switch block outputs if ***mode*** is MVIP95_CONNECT_MODE or MVIP95_FRAMED_CONNECT_MODE.
count	Number of elements in the ***output***, ***mode***, ***pattern***, and ***input*** arrays.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_ERROR	Underlying driver retrieved an unrecognized error. Call **swiGetLastError** to retrieve the MVIP device error code.
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
CTAERR_SVR_COMM	Communication error in the server environment.
SWIERR_INVALID_MODE	***mode*** value returned by the driver is unrecognized.
SWIERR_INVALID_STREAM	One or more of the specified streams is invalid.
SWIERR_INVALID_TIMESLOT	One or more of the specified timeslots is invalid.

Details

swiGetOutputState retrieves the state of the specified switch block outputs. The following table describes the ***mode*** values:

If *mode* is...	Then...
MVIP95_DISABLE_MODE	The switch block outputs on the MVIP bus are in their high impedance state. If a switch block output is on a local bus, the timeslots on the local bus are put into a known state. (The driver vendor must publish these states in customer documentation.)
MVIP95_PATTERN_MODE	A fixed pattern is being asserted on the switch block output.
MVIP95_CONNECT_MODE or MVIP95_FRAMED_CONNECT_MODE	There is a connection from the input terminus to the output terminus.

Note: Under Solaris, the upper limit for the number of terminuses that can be batched in one call is 32.

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

See also

swiDisableOutput, **swiMakeFramedConnection**, **swiResetSwitch**, **swiSendPattern**

Example

```
void myPrintOutputState(SWIHD hd, SWI_TERMINUS output)
{
    unsigned mode;
    BYTE pattern;
    SWI_TERMINUS input;

    swiGetOutputState(hd, &output, &mode, &pattern, &input, 1);

    switch (output.bus)
    {
    case MVIP95_MVIP_BUS:
        printf("STo(%s", "mvip");
        break;
    case MVIP95_LOCAL_BUS:
        ("STo(%s", "local");
        break;
    }
    printf(":%2d:%02d) ", output.stream, output.timeslot);

    switch( mode )
    {
    case MVIP95_CONNECT_MODE:
    case MVIP95_FRAMED_CONNECT_MODE:
        switch (input.bus)
        {
        case MVIP95_MVIP_BUS:
            printf(" %s", " mvip");
            break;
        case MVIP95_LOCAL_BUS:
            printf(" %s", "local");
            break;
        }
        printf(":%2d:%02d", input.stream, input.timeslot);
        break;

    case MVIP95_PATTERN_MODE:
        printf("        m_%02X", pattern );
        break;

    case MVIP95_DISABLE_MODE:
        printf("        t_%02X", 0 );
        break;

    default:
        printf("        %02X_%02X", 0, 0);
        break;
    }
    printf("\n");
}
```

swiGetStreamsBySpeed

Returns information that identifies all H.100 streams operating at one specified speed.

Prototype

DWORD **swiGetStreamsBySpeed** (SWIHD ***swihd***, DWORD ***speed***, DWORD ***streams[]***, unsigned ***maxcount***, unsigned ****count***)

Argument	Description
swihd	Switch handle.
speed	Specifies in millions of bits per second the capacity of one or more streams. Refer to the Details section for acceptable values.
streams	Array that receives the list of the H.100 streams configured at the specified speed.
maxcount	Maximum number of streams allowed.
count	Pointer to the returned number of ***streams***.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_ERROR	Underlying driver retrieved an unrecognized error. Call **swiGetLastError** to retrieve the MVIP device error code.
CTAERR_FUNCTION_NOT_AVAIL	Underlying driver does not support the configuration of stream-specific characteristics of a local device.
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
CTAERR_SVR_COMM	Communication error in the server environment.
SWIERR_INVALID_SPEED	Specified ***speed*** is unrecognized. Use a speed value specified in the Details section.

Details

swiGetStreamsBySpeed is specific to MVIP-95. **swiGetStreamsBySpeed** retrieves a list of the H.100 streams that are operating at one specified speed.

Pass 0 for ***maxcount*** to get the number of streams only.

Acceptable values for ***speed*** are:

- MVIP95_2MBPS_STREAM_SPEED
- MVIP95_4MBPS_STREAM_SPEED
- MVIP95_8MBPS_STREAM_SPEED

Refer to *Configuring stream speed* on page 24 for more information.

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

See also

swiConfigStreamSpeed

Example

```
void myPrintMVIP95Streams(SWIHD hd)
{
    DWORD *streams;
    unsigned count, i;

    /* First get number of streams, by specifying 0 for the maxcount */
    swiGetStreamsBySpeed(hd, MVIP95_2MBPS_STREAM_SPEED, streams, 0, &count);
    streams = (DWORD *)malloc(sizeof(DWORD)*count);

    /* Now get actual stream numbers */
    swiGetStreamsBySpeed(hd, MVIP95_2MBPS_STREAM_SPEED, streams, count,
    &count);

    printf("MVIP-95 compatible streams:\n");
    for (i = 0; i < count; i++)
    {
        printf("%d ", streams[i]);
    }
    printf("\n");
    free(streams);
}
```

swiGetSwitchCaps

Returns information about the capabilities of the device driver and the switch controlled by it.

Prototype

DWORD **swiGetSwitchCaps** (SWIHD ***swihd***, SWI_SWITCHCAPS_ARGS ****args***, SWI_LOCALDEVICE_DESC ***localdevs[]***, unsigned ***maxcount***)

Argument	Description
swihd	Switch handle.

args	Pointer to a switch capabilities parameter structure for the device driver information:
	`typedef struct`
	`{`
	`  DWORD  dvrrevision;`
	`  DWORD  domain;`
	`  DWORD  routing;`
	`  DWORD  blocking;`
	`  DWORD  swstandard;`
	`  DWORD  swstdrevision;`
	`  DWORD  hwstandard;`
	`  DWORD  hwstandardrevision;`
	`  DWORD  numlocalstreams;`
	`} SWI_SWITCHCAPS_ARGS;`
	Refer to the Details section for a description of these fields.

localdevs	Array that receives the number of timeslots and the device type of each local stream. The SWI_LOCALDEVICE_DESC structure is:
	`typedef struct`
	`{`
	`  DWORD  timeslots;`
	`  DWORD  deviceid;`
	`} SWI_LOCALDEVICE_DESC;`
	Refer to the Details section for a description of these fields.

maxcount	Maximum number of elements in the ***localdevs*** array.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_ERROR	Underlying driver retrieved an unrecognized error. Call **swiGetLastError** to retrieve the MVIP device error code.
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
CTAERR_SVR_COMM	Communication error in the server environment.

Details

swiGetSwitchCaps queries the device driver and returns the capabilities of the device driver and the switch block. Use this function any time after opening a switch block.

The SWI_SWITCHCAPS_ARGS structure contains the following fields:

Field	Description
dvrrevision	Revision level of the device driver multiplied by 100 (decimal).
domain	Domain of a switch block. A switch block's domain indicates which MVIP streams have full-duplex connections to the switch block. The representation of these streams is a bit field. If a full-duplex connection to the switch block is supported, the corresponding bit in the domain parameter is set to one. Bit 0 indicates whether the switch block supports a full-duplex connection to HDS0 (MVIP-95). In the context of MVIP-95 hardware implementations, bits 0 through 23 indicate the switch block's support for full-duplex connections with HDS0 through HDS23. A full-duplex connection means that an MVIP HDS stream can provide both input to the switch block and output from the switch block. The domain of all MVIP-95 standard-compliant switches is 1111 1111 1111 1111 (0xFFFF).
routing	Switch block's half-duplex routing capabilities. The representation of these capabilities is a bit field. When the switch block has a routing capability, the bit is set to 1. Refer to the next table for more information on bits and routing capabilities.
blocking	Switch block's possible blocking. Blocking means that a switch block is unable to provide all possible connections. A bit field represents where blocking is possible; a value of 1 signifies this possible restriction. The bit field for the blocking parameter has the same meaning as the bit field for the routing parameter.
swstandard	MVIP software standard to which the device driver conforms. The acceptable value is MVIP95_STANDARD_MVIP95.
swstdrevision	Revision of the software standard to which the device driver conforms multiplied by 100 (decimal).
hwstandard	NMS boards return MVIP95_STANDARD_HMVIP.
hwstandardrevision	Revision of the hardware standard controlled by the device driver multiplied by 100 (decimal).
numlocalstreams	Number of streams on the local bus side of the switch block. This number includes both the data and signaling streams for network devices such as T1 framers.

The following table presents the correspondence of bits in the bit field to the switch block's half-duplex routing capabilities:

Bit	Half-duplex routing capability
0	Even-numbered CT_D stream to odd-numbered CT_D stream.
1	Odd-numbered CT_D stream to even-numbered CT_D stream.
2	Even-numbered CT_D stream to even-numbered CT_D stream.
3	Odd-numbered CT_D stream to odd-numbered CT_D stream.
4	Even-numbered CT_D stream to local.
5	Local to odd-numbered CT_D stream.
6	Odd-numbered CT_D stream to local.
7	Local to even-numbered CT_D stream.
8	Local to local.

The SWI_LOCALDEVICE_DESC structure contains the following fields:

Field	Description
timeslots	Number of timeslots on the local stream.
deviceid	Device identifier of the local stream.

Refer to *Querying switch capabilities* on page 24 for more information.

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

See also

swiDisableOutput, **swiGetLocalStreamInfo**, **swiGetLocalTimeslotInfo**, **swiGetOutputState**, **swiMakeConnection**, **swiMakeFramedConnection**, **swiResetSwitch**, **swiSampleInput**, **swiSendPattern**

Example

```
void myPrintSwitchCaps(SWIHD hd)
{
    SWI_SWITCHCAPS_ARGS cp;
    SWI_LOCALDEV_DESC *localdevs;

    swiGetSwitchCaps(hd, &cp, NULL, 0);

    localdevs = (SWI_LOCALDEV_DESC *)malloc(
        sizeof(SWI_LOCALDEV_DESC)*cp.numlocalstreams);

    swiGetSwitchCaps(hd, &cp, localdevs, cp.numlocalstreams);

    printf("Driver Software Std. %s Rev. %2.f\n",
        ((cp.swstandard == MVIP95_STANDARD_MVIP95)? "MVIP-95" :
         "other"),
        (float)cp.swstdrevision/100.0);

    printf("Hardware Std. %s Rev. %2.f.\n",
           ((cp.hwstandard == MVIP95_STANDARD_HMVIP)? "HMVIP" :
            "other"),
           (float)cp.hwstdrevision/100.0);
    printf("Driver Rev. %.2f\n", (float)cp.dvrrevision/100.0);
    printf(" Domain %04X, Routing %04X, Blocking %04X.\n",
           cp.domain, cp.routing, cp.blocking );

    if( cp.numlocalstreams > 0 )
    {
        DWORD i;

        printf("Supports %d local streams:\n\t",
               cp.numlocalstreams );
        for( i=0; i<cp.numlocalstreams; i++ )
            printf( "%2d ", i+16 );
        printf("with\n\t");
        for( i=0; i<cp.numlocalstreams; i++ )
            printf( "%2d ", localdevs[i].timeslots );
        printf("timeslots respectively.\n");
    }
    free(localdevs);
}
```

Sample Run

The output would be:

```
Driver Software Std. MVIP-95 Rev. 0.00, Hardware Std. H.110 Rev.  0.
Driver Rev. 17.00, Domain FFFF, Routing 01FF, Blocking 00FF.
Supports 4 local streams:
             16 17 18 19 with
             24 24 32 32 timeslots respectively.
```

swiGetTimingReference

Retrieves the status of a potential TDM bus clock timing reference. The timing reference can be an external digital trunk or an internal oscillator.

Prototype

DWORD **swiGetTimingReference** (SWIHD ***swihd***, DWORD ***referencesource***, DWORD ***network***, SWI_QUERY_TIMING_REFERENCE_ARGS ****args***, unsigned ***size***)

Argument	Description
swihd	Switch handle.
referencesource	Specifies which clock reference is to be queried. Refer to the Details section for acceptable values.
network	Specifies the device source for the CT bus clock signals. Refer to the Details section for acceptable values.

args	Pointer to a SWI_QUERY_TIMING_REFERENCE_ARGS clock parameter structure for querying the potential timing reference:
	`typedef struct`
	`{`
	`  DWORD status;`
	`  DWORD statusinfo;`
	`} SWI_QUERY_TIMING_REFERENCE_ARGS;`
	Refer to the Details section for a description of these fields.

size	Specifies the number of bytes contained in the structure.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_ERROR	Underlying driver retrieved an unrecognized error. Call **swiGetLastError** to retrieve the MVIP device error code.
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
CTAERR_SVR_COMM	Communication error in the server environment.
SWIERR_INVALID_CLOCK_PARM	Value of a clock configuration parameter is invalid.
SWIERR_INVALID_PARAMETER	Board-specific clock parameter value is invalid.

Details

swiGetTimingReference retrieves status information about a potential timing reference.

Acceptable values for ***referencesource*** are:

- MVIP95_SOURCE_INTERNAL
- MVIP95_SOURCE_NETWORK

Acceptable values for ***network*** are 1 to ***n***, where ***n*** is the number of devices on the specified board capable of being a clock source. A value for network is valid only when ***referencesource*** is equal to MVIP95_SOURCE_NETWORK.

The following table describes the fields in the SWI_QUERY_TIMING_REFERENCE_ARGS structure:

Field	Description
status	Status of the timing reference. Acceptable values are: MVIP95_TIMING_REF_STATUS_GOOD MVIP95_TIMING_REF_STATUS_BAD MVIP95_TIMING_REF_STATUS_UNKNOWN
statusinfo	Additional optional information about the timing reference status. Acceptable values are: MVIP95_TIMING_REF_UNKNOWN MVIP95_TIMING_REF_NO_ALARM MVIP95_TIMING_REF_RED_ALARM

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

See also

swiConfigNetrefClock, **swiGetBoardClock**, **swiOpenSwitch**

Example

```
void myGetTimingReference (SWIHD swihd)
{
    SWI_QUERY_TIMING_REFERENCE_ARGS   querytimingref;
    DWORD                             referencesource;
    DWORD                             network;
    unsigned                          size;

    size = sizeof (SWI_QUERY_TIMING_REFERENCE_ARGS);
    swiGetTimingReference(swihd, referencesource, network, &querytimingref,
                          size);
}
```

swiMakeConnection

Connects inputs to outputs.

Prototype

DWORD **swiMakeConnection** (SWIHD ***swihd***, SWI_TERMINUS ***input[]***, SWI_TERMINUS ***output[]***, unsigned ***count***)

Argument	Description
swihd	Switch handle.

input	Array of terminus structures for the input side of the connection:
	`typedef struct`
	`{`
	`  DWORD bus;`
	`  DWORD stream;`
	`  DWORD timeslot;`
	`} SWI_TERMINUS;`
	Refer to the Details section for a description of these fields.

output	Array of terminus structures for the output side of the connection.
count	Number of elements in the ***input*** array and in the ***output*** array.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_ERROR	Underlying driver retrieved an unrecognized error. Call **swiGetLastError** to retrieve the MVIP device error code.
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
CTAERR_SVR_COMM	Communication error in the server environment.
SWIERR_INVALID_STREAM	One or more of the specified streams is invalid.
SWIERR_INVALID_TIMESLOT	One or more of the specified timeslots is invalid.

Details

swiMakeConnection makes connections between the corresponding elements of the output terminus array and the input terminus array.

If multiple connections are made (that is, ***count*** is greater than 1), the relative throughput delay of the connections may not all be the same. Use **swiMakeFramedConnection** if you need identical throughput delays.

Note: Under Solaris, the upper limit for the number of terminuses that can be batched in one call is 32.

The SWI_TERMINUS structure contains the following fields:

Field	Description
bus	Interface point of the switch block. Devices can reside directly on the MVIP bus. Devices can also reside on a board's local bus and may require a switch block to access the MVIP bus. Acceptable bus values are: MVIP95_MVIP_BUS MVIP95_LOCAL_BUS
stream	A grouping of timeslots that usually corresponds to a particular bit stream of time-division multiplexed (TDM) serial data on an individual track or wire of a bus.
timeslot	A particular 64 kbit/s subdivision of a TDM bus stream. Timeslots number from 0 (zero) to ***n*** where ***n*** is stream dependent.

Note: Disable an output when the connection or pattern is no longer required. Leftover connections or patterns can cause unpredictable behavior in the application.

Refer to *Making connections* on page 20 for more information.

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

See also

swiDisableOutput, **swiGetOutputState**, **swiGetSwitchCaps**, **swiResetSwitch**, **swiSampleInput**, **swiSendPattern**

Note: Refer to the *MVIP-95 switch block model* on page 12 for information about stream numbering when making duplex connections.

Example

```
#define SIMPLEX 0
#define DUPLEX  1
void myMakeConnection(SWIHD hd, SWI_TERMINUS input, SWI_TERMINUS output,
                      unsigned count, DWORD mode)
{
    unsigned i;
    DWORD duplex = 0;
    SWI_TERMINUS *outputs, *inputs;

    if (mode == DUPLEX)
        duplex = 1;

    inputs = (SWI_TERMINUS *)malloc(sizeof(SWI_TERMINUS)*count);
    outputs = (SWI_TERMINUS *)malloc(sizeof(SWI_TERMINUS)*count);

    for (i = 0; i < count; i++)
    {
        inputs[i].bus = input.bus;
        inputs[i].stream = input.stream;
        inputs[i].timeslot = input.timeslot + i;

        outputs[i].bus = output.bus;
        outputs[i].stream = output.stream;
        outputs[i].timeslot = output.timeslot + i;
    }
    swiMakeConnection(hd, inputs, outputs, count);

    if( duplex )
    {
        for (i = 0; i < count; i++)
        {
            inputs[i].stream = inputs[i].stream + 1;
            outputs[i].stream = outputs[i].stream - 1;
        }
        swiMakeConnection(hd, outputs, inputs, count);
    }
}
```

swiMakeFramedConnection

Connects inputs to outputs with identical throughput delay for all connections.

Prototype

DWORD **swiMakeFramedConnection** (SWIHD ***swihd***, SWI_TERMINUS ***input[],*** SWI_TERMINUS ***output[],*** unsigned ***count***)

Argument	Description
swihd	Switch handle.

input	Array of terminus structures for the input side of the connection:
	`typedef struct`
	`{`
	`  DWORD bus;`
	`  DWORD stream;`
	`  DWORD timeslot;`
	`} SWI_TERMINUS;`
	Refer to *swiMakeConnection* on page 76 for a description of these fields.

output	Array of terminus structures for the output side of the connection.
count	Number of elements in the ***input*** array and in the ***output*** array.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_ERROR	Underlying driver retrieved an unrecognized error. Call **swiGetLastError** to retrieve the MVIP device error code.
CTAERR_FUNCTION_NOT_AVAIL	Underlying driver does not support the configuration of specific characteristics of a local device.
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
CTAERR_SVR_COMM	Communication error in the server environment.
SWIERR_INVALID_STREAM	One or more of the specified streams is invalid.
SWIERR_INVALID_TIMESLOT	One or more of the specified timeslots is invalid.

Details

swiMakeFramedConnection is specific to MVIP-95. **swiMakeFramedConnection** makes connections between the corresponding elements of the output terminus array and the input terminus array.

The type of connection is identical to **swiMakeConnection** except that all connections made on the same switch device with this function have the same constant throughput delay.

Use **swiMakeFramedConnection** to make a single high bandwidth connection using multiple timeslots where the data must be synchronized across the timeslots. This function ensures there is identical throughput delay across all timeslots.

Note: Under Solaris, the upper limit for the number of terminuses that can be batched in one call is 32.

This function returns CTAERR_FUNCTION_NOT_AVAIL if framed connections are not supported.

Disable an output when the connection or pattern is no longer required. Leftover connections or patterns can cause unpredictable behavior in the application.

Refer to *Making connections* on page 20 for more information.

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

See also

swiDisableOutput, **swiGetOutputState**, **swiGetSwitchCaps**, **swiResetSwitch**, **swiSampleInput**, **swiSendPattern**

Example

```
#define SIMPLEX 0
#define DUPLEX  1
void myMakeFramedConnection(SWIHD hd, SWI_TERMINUS input,
                            SWI_TERMINUS output,
                            unsigned count, DWORD mode)
{
    unsigned i;
    DWORD duplex = 0;
    SWI_TERMINUS *outputs, *inputs;

    if (mode == DUPLEX)
        duplex = 1;

    inputs = (SWI_TERMINUS *)malloc(sizeof(SWI_TERMINUS)*count);
    outputs = (SWI_TERMINUS *)malloc(sizeof(SWI_TERMINUS)*count);

    for (i = 0; i < count; i++)
    {
        inputs[i].bus = input.bus;
        inputs[i].stream = input.stream;
        inputs[i].timeslot = input.timeslot + i;

        outputs[i].bus = output.bus;
        outputs[i].stream = output.stream;
        outputs[i].timeslot = output.timeslot + i;
    }
    swiMakeFramedConnection(hd, inputs, outputs, count);

    if( duplex )
    {
        for (i = 0; i < count; i++)
        {
            inputs[i].stream = inputs[i].stream + 1;
            outputs[i].stream = outputs[i].stream - 1;
        }
        swiMakeFramedConnection(hd, outputs, inputs, count);
    }
}
```

swiOpenSwitch

Opens a switching device and returns a switch handle.

Prototype

DWORD **swiOpenSwitch** (CTAHD ***ctahd***, char ****devname***, unsigned ***swno***, unsigned ***flags***, SWIHD ****swihd***)

Argument	Description
ctahd	Context handle.
devname	Pointer to the base name of the switching device, for example, agsw.
swno	Logical switch number.
flags	Flags for the open call. Refer to the Details section for a description of these fields.
swihd	Pointer to the returned switch handle.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_OPEN_FAILED	Driver open failed. Refer to the board installation manual.
CTAERR_NOT_FOUND	Driver dynamic link library or the device was not found. Refer to the board installation manual.
CTAERR_INVALID_CTAHD	***ctahd*** referenced by ***swihd*** is invalid.
CTAERR_SVR_COMM	Communication error in the server environment.

Details

swiOpenSwitch opens a switching device and returns a handle to the switch block for use in subsequent switching calls.

devname identifies the switching driver. AG and CG boards use the ***devname*** agsw. The CX 2000 board uses the ***devname*** cxsw.

swno is the board number and must match the logical number assigned to the board when performing OAM configuration.

Caution:	Since MVIP-95 device drivers can be opened only in MVIP-95 mode, NMS recommends writing all applications that may use MVIP-95 device drivers in the future to use MVIP-95 mode, even if the application is not currently using MVIP-95 device drivers.

The SWI_ENABLE_RESTORE bit is used to save the state of the switch block outputs so that the state of the switch block outputs can be restored when it is closed using **swiCloseSwitch**.

Acceptable values for ***flags*** are:

- Zero (0)

  The switch is opened in MVIP-95 mode and restoration is disabled.
- SWI_ENABLE_RESTORE
- SWI_ENABLE_ RESTORE | SWI_MVIP90

Refer to *Opening a switch handle* on page 17 for more information.

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

Example

```
void myApplicationInit(CTAHD ctahd, SWIHD hd[], int count)
{
    int i;

    for (i = 0; i < count; i++)
    {
        swiOpenSwitch(ctahd, "agsw", i, SWI_ENABLE_RESTORE, &hd[i]);
    }

}
```

swiResetSwitch

Resets the entire switch block to the idle state.

Prototype

DWORD **swiResetSwitch** (SWIHD ***swihd***)

Argument	Description
swihd	Switch handle.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_ERROR	Underlying driver retrieved an unrecognized error. Call **swiGetLastError** to retrieve the MVIP device error code.
CTAERR_FUNCTION_NOT_AVAIL	Reset switch was called on a switch that was opened in restore mode.
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
CTAERR_SVR_COMM	Communication error in the server environment.

Details

swiResetSwitch resets the entire switch block to the idle state. All the timeslots on the MVIP bus are set to a high impedance state. The timeslots on the local bus are set to a known state. (The driver vendor must publish these states in customer documentation.)

If the switch handle was opened with the SWI_ENABLE_RESTORE ***flag*** specified in **swiOpenSwitch**, this function returns CTAERR_FUNCTION_NOT_AVAIL and does not affect the switch block.

Refer to *Enabling terminus output state restoration* on page 18 for more information.

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

Note: This function does not alter the state of the clocks.

Example

```
extern SWIHD hd;

void myAppInit(CTAHD ctahd)
{
    /* Open the Switch */
    swiOpenSwitch(ctahd, "AGSW", 0, 0, &hd);

    /* Reset the Switch to defaults */
    swiResetSwitch(hd);
}
```

swiSampleInput

Retrieves the current data values present on specified switch block inputs.

Prototype

DWORD **swiSampleInput** (SWIHD ***swihd***, SWI_TERMINUS ***input[]***, BYTE ***data[]***, unsigned ***count***)

Argument	Description
swihd	Switch handle.

input	Array of terminus structures for the input of the switch block:
	`typedef struct`
	`{`
	`  DWORD bus;`
	`  DWORD stream;`
	`  DWORD timeslot;`
	`} SWI_TERMINUS;`
	Refer to *swiMakeConnection* on page 76 for a description of these fields.

data	Array that receives values present on the specified switch block inputs.
count	Number of elements in the ***input*** and ***data*** arrays.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_ERROR	Underlying driver retrieved an unrecognized error. Call **swiGetLastError** to retrieve the MVIP device error code.
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
CTAERR_SVR_COMM	Communication error in the server environment.
SWIERR_INVALID_STREAM	One or more of the specified streams is invalid.
SWIERR_INVALID_TIMESLOT	One or more of the specified timeslots is invalid.

Details

swiSampleInput reads the data available on one or more inputs of a switch block. Calling this function does not affect the switch block inputs or connections.

Note: Under Solaris, the upper limit for the number of terminuses that can be batched in one call is 32.

Refer to *Sampling data* on page 23 for more information.

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

See also

swiGetOutputState, **swiMakeFramedConnection**, **swiOpenSwitch**, **swiSendPattern**

Example

```
void myPrintInput(SWIHD hd, SWI_TERMINUS input)
{
    BYTE data;

    swiSampleInput(hd, &input, &data, 1);

    switch (input.bus)
    {
    case MVIP95_MVIP_BUS:
        printf(" %s", " mvip");
        break;
    case MVIP95_LOCAL_BUS:
        printf(" %s", "local");
        break;
    }

    printf(":%2d:%02d=%02X\n", input.stream, input.timeslot, data);
}
```

swiSendPattern

Asserts fixed patterns on specified switch block outputs.

Prototype

DWORD **swiSendPattern** (SWIHD ***swihd***, BYTE ***pattern[],*** SWI_TERMINUS ***output[],*** unsigned ***count***)

Argument	Description
swihd	Switch handle.
pattern	Array of patterns to transmit on the specified switch block outputs.

output	Array of terminus structures specifying the switch block outputs on which the patterns are to be asserted:
	`typedef struct`
	`{`
	`  DWORD bus;`
	`  DWORD stream;`
	`  DWORD timeslot;`
	`} SWI_TERMINUS;`
	Refer to *swiMakeConnection* on page 76 for a description of these fields.

count	Number of elements in the ***pattern*** and ***output*** arrays.

Return values

Return value	Description
SUCCESS	
CTAERR_DRIVER_ERROR	Underlying driver retrieved an unrecognized error. Call **swiGetLastError** to retrieve the MVIP device error code.
CTAERR_INVALID_HANDLE	***swihd*** is not a valid switch handle.
CTAERR_SVR_COMM	Communication error in the server environment.
SWIERR_INVALID_PARAMETER	Board-specific clock parameter value is invalid.
SWIERR_INVALID_STREAM	One or more of the specified streams is invalid.
SWIERR_INVALID_TIMESLOT	One or more of the specified timeslots is invalid.

Details

For each element of the specified ***pattern*** array, **swiSendPattern** sends the 8-bit pattern on the output terminus in the corresponding element of the ***output*** array.

Note: Under Solaris, the upper limit for the number of terminuses that can be batched in one call is 30.

Disable an output when the connection or pattern is no longer required. Leftover connections or patterns can cause unpredictable behavior in the application.

Refer to *Sending a pattern* on page 22 for more information.

If CTAERR_DRIVER_ERROR is returned, call **swiGetLastError** to retrieve the MVIP device error code.

See also

swiCloseSwitch, **swiDisableOutput**, **swiGetOutputState**, **swiGetSwitchCaps**, **swiMakeFramedConnection**, **swiOpenSwitch**, **swiResetSwitch**, **swiSampleInput**

Example

```
/* Send the same pattern to many output terminuses */
void mySendPattern(SWIHD hd, BYTE pattern, SWI_TERMINUS output, unsigned count)
{
    SWI_TERMINUS *outputs;
    BYTE *patterns;
    unsigned i;

    outputs = (SWI_TERMINUS *)malloc(sizeof(SWI_TERMINUS)*count);
    patterns = (BYTE *)malloc(sizeof(BYTE)*count);
    for (i = 0; i < count; i++)
    {
        outputs[i].bus = output.bus;
        outputs[i].stream = output.stream;
        outputs[i].timeslot = output.timeslot + i;
        patterns[i] = pattern;
    }
    swiSendPattern(hd, patterns, outputs, count);
    free(outputs);
    free(patterns);
}
```

7 Demonstration programs and utilities

Summary of the demonstration programs and utilities

The Switching service provides the following demonstration programs and utilities both as executable programs and as source code:

Program or utility	Description
prt2prt	Demonstrates call transfer from an incoming line to an outgoing line and uses the Switching service to make connections, send patterns, and so on.
swish	Allows interactive or text-file-driven control of MVIP switches. It provides a convenient way to manually test connections during development to verify the commands that will be given to switches from within Natural Access applications that use the Switching service.
showcx95	Displays switch connections for all boards that have MVIP switches.

For information about the *clockdemo* demonstration program, refer to the *NMS OAM System User's Manual*.

Before you start the demonstration programs, verify that:

- Natural Access is properly installed.
- The board is running.
- MVIP switching is correctly configured.

Port to port program: prt2prt

Demonstrates:

- Call transfer from an incoming line to an outgoing line.
- Using the Switching service to make connections and send patterns.

Usage

prt2prt [***options***]

where ***options*** are:

Option	Use this option to specify the...
-b ***n***	Board number ***n***. Default = 0.
-d **driver_name**	Name of the driver dynamic link library. Default = agsw.
-s ***i,o***	Incoming call timeslot and the outgoing call timeslot. Default: 0,2.
-i ***protocol***	Protocol to run on the incoming call side. Default = lps0.
-o ***protocol***	Protocol to run on the outgoing call side. Default = lps0.

Description

prt2prt performs call transfer of an incoming call to an outgoing line. It uses the Switching service to connect the voice streams of two calls so that a conversation can be carried out.

Because *prt2prt* is an actual demonstration of the Switching service, it uses MVIP-95 stream values.

Procedure

The following procedure assumes that you:

- Are testing on an AG 2000 board with a loop start hybrid on slot 0 and a DID hybrid on slot 4.
- Have downloaded the *lps0.tcp* and the *wnk0.tcp* files to the board.
- Have a 2500-type telephone connected to the DID hybrid and some way of placing calls and accepting calls over the loop start line (for example, with a central office simulator with another telephone handset connected to it).

To run *prt2prt*:

1. Start OAM. For more information about OAM, refer to the *NMS OAM Service Developer's Reference Manual*.
2. Start *prt2prt* by entering the following command:

```
prt2prt -b 0 -s 4,0 -i wnk0 -o lps0
```

   This command uses board 0 and the DID hybrid in slot 4 for the incoming calls and the loop start hybrid in slot 0 for the outgoing call. It also uses the wink start protocol on the incoming call and the loop start protocol on the outgoing call.

   prt2prt starts and the following information displays:

```
    Natural Access Port to Port Calling Demo V.5
       Board                   = 1
       InSlot,OutSlot          = 4,0
       Incoming Protocol       = wnk0
       Outgoing Protocol       = lps0
       NPorts                  = 1

       Initializing and opening the CTA context...
       Tracing disabled. Check that the daemon is running.
         Event: CTAEVN_OPEN_SERVICES_DONE, Finished
         Event: CTAEVN_OPEN_SERVICES_DONE, Finished
         Event: ADIEVN_STARTPROTOCOL_DONE, Finished
        --------
       Waiting for incoming call...
```

   prt2prt waits for an incoming call.

3. Put the telephone on stream/timeslot local:0:4 off-hook.

   You hear a relay click.

4. Dial a three-digit telephone number.

   prt2prt plays a message prompting you to dial an extension.

   The following information displays:

```
   Event: ADIEVN_INCOMING_CALL
       Incoming Call...
       Answering call...
         Event: ADIEVN_ANSWERING_CALL
         Event: ADIEVN_CALL_CONNECTED, Answered
   Call connected.
   Playing file 'ctademo', msg #12...
   Playing 1 messages from 'ctademo'...
         Event: VCEEVN_PLAY_DONE, Finished ,nbytes=1240
   Getting extension...
   Playing file 'ctademo', msg #16...
   Playing 1 messages from 'ctademo'...
```

5. Dial an extension.

   prt2prt informs you that it will dial that extension and then places the call on the outgoing timeslot.

   The following information displays:

```
   Collecting digits to dial...
         Event: ADIEVN_DIGIT_BEGIN, '5'
         Event: ADIEVN_DIGIT_END
         Event: ADIEVN_COLLECTION_DONE, Timeout
           digit string = '5'
   Got extension: 5
   Playing file 'ctademo', msg #17...
   Playing 1 messages from 'ctademo'...
         Event: VCEEVN_PLAY_DONE, Finished ,nbytes=2700
         Event: ADIEVN_STARTPROTOCOL_DONE, Finished
--------
   Placing a call to '5'...
         Event: ADIEVN_PLACING_CALL
```

6. Answer the call on the other phone that is connected to the outgoing line through the central office simulator.

```
         Event: ADIEVN_CALL_CONNECTED, Voice Begin
            Connected.
```

   prt2prt connects the two voice streams so that a conversation can take place between the two telephones.

7. Confirm that the two voice streams are connected by knocking on one mouthpiece. You will hear the knocking on the other mouthpiece.

8. Hang up the incoming side. *prt2prt* tears down the call and goes back into a waiting for call state.

   The following information displays:

```
   Hanging up...
         Event: ADIEVN_CALL_RELEASED
  Call done.
   Hanging up...
         Event: ADIEVN_CALL_RELEASED
   Call done.
         Event: ADIEVN_STOPPROTOCOL_DONE, Finished
         Event: ADIEVN_STOPPROTOCOL_DONE, Finished
         Event: ADIEVN_STARTPROTOCOL_DONE, Finished
   --------
   Waiting for incoming call...
```

Control MVIP switches: swish

Provides interactive or text-file-driven control of MVIP switches through the Switching service or Point-to-Point Switching service, or by directly using the device drivers.

Usage

swish [***options***] [***filename***]

where ***options*** are:

Option	Use this option to specify the...
-s ***context_name***	Name of a sharable context.
-i ***filename***	Default initialization file, *swish.ini*, to ignore.
-@ ***server***	Server host name or IP address. Default = localhost.

filename specifies an ASCII file containing *swish* commands to be executed before any interactive commands are executed.

Description

swish is a tool for interactive or text-file-driven control of MVIP switches. It provides a convenient way to manually try out connections during development to verify the commands that will be given to switches from within Natural Access applications that use the Switching service.

Procedure

Use *swish* in interactive mode, text-file-driven mode, or in a combination of the two.

In interactive mode, you enter *swish* commands at the *swish* command prompt to control the MVIP switches on the underlying hardware.

In text-file-driven mode, you provide *swish* with the name of the file to read for the commands to run. If the text file does not have the exit command at the end, *swish* goes into interactive mode after executing the commands in the text file. Use the ***filename****.swi* naming convention for the input text file.

Scripts that demonstrate *swish* command syntax for MVIP-related commands can be found in these directories:

Operating system	Directory
Windows	*\nms\ctaccess\demos\swish*
UNIX	*opt/nms/ctaccess/demos/swish*

To run *swish* interactively, enter the following command at the prompt:

```
swish
```

To run *swish* using input from a text file, enter the following command at the prompt:

```
swish filename
```

In interactive mode, the command prompt swish: displays. Enter the help command to get a list of commands and arguments that *swish* supports. You can also enter the name of a command without any arguments to get more information about the command.

In text-file-driven mode, *swish* executes the commands from the specified file. The syntax of the commands in the file is the same as the syntax of the commands in interactive mode.

swish supports four kinds of commands:

- Switching service commands
- Point-to-Point Switching service commands
- MVIP-90 driver commands (legacy support for older systems)
- MVIP-95 driver commands

The Switching service commands are the interface to the Switching service. The names of the commands are the same as the functions provided by the Switching service, and the arguments to the commands are also similar. Refer to the Function reference section for information about the Switching service functions.

The driver commands enable you to directly access the underlying driver by bypassing the Switching service. Some of the Switching service commands and the driver commands have the same names. *swish* uses the Switching service command by default.

To distinguish among the *swish* commands, refer to the following table:

Use this prefix...	To indicate...
ppx	Point-to-Point Switching service commands.
swi	Switching service commands.
drv	MVIP-90 driver commands (retained for legacy systems).
d95	MVIP-95 driver commands.

The Switching service commands provided by *swish* take a switch handle as an argument. This switch handle is opened by the **OpenSwitch** command. The driver commands take a driver handle as an argument and it is opened by the **LoadDriver** (Windows) or the **OpenDevice** (UNIX) command. The driver handle and the switch handle are not interchangeable: a switch handle cannot be used in a driver command and a driver handle cannot be used in a Switching service command.

The commands presented in the following table are supported by *swish*.

If you need help on a command, type the name of the command at the *swish* prompt and press **Enter**. The syntax and the options for the command are displayed.

Command	Arguments	Description
swi.OpenSwitch	***SwiHd*** = ***DLLname SwNo*** [RESTORE=...] [STD=...]	Opens a switch.
swi.CloseSwitch	***SwiHd***	Closes a switch.
swi.DisableOutput	***SwiHd ToList***	Disables outputs of switch.
swi.GetOutputState	***SwiHd ToList***	Displays connections.
swi.Caps	***SwiHd***	Displays switch capabilities.
swi.GetSwitchCaps	***SwiHd***	Displays switch capabilities.
swi.MakeConnection	***SwiHd TiList*** TO ***ToList*** [SIMPLEX|DUPLEX|QUAD]	Connects inputs to outputs.
swi.MakeFramedConnection	***SwiHd TiList*** TO ***ToList*** [SIMPLEX|DUPLEX|QUAD]	Connects inputs to outputs with identical throughput delay for all connections.
swi.ResetSwitch	***SwiHd***	Resets a switching block.
swi.SendPattern	***SwiHd pattrn*** TO ***ToList***	Sends a pattern.
swi.SampleInput	***SwiHd TiList***	Displays data memory.
swi.ConfigBoardClock	***SwiHd TYPE=STD... SOURCE=MVIP...***	Establishes the clock source for an MVIP board.
swi.ConfigBoardH100Clock	***SwiHd TYPE=H100... SOURCE=INTERNAL... H100MODE=SLAVE... FALLBACK=ENABLE... NETREFSPEED=8KHZ... FALLBACKSOURCE=INTERNAL...***	Establishes the clock source for an H.100/H.110 board.
swi.ConfigNetrefClock	***SwiHd SOURCE=INTERNAL... NETREFMODE=NETREF_1... NETREFSPEED=8KHZ...***	Defines the source of the NETREF clocks (H.100/H.110).
swi.QueryBoardClock	***SwiHd TYPE=H100...***	Returns board clocking configuration and clock status information.
swi.QueryTimingReference	***SwiHd SOURCE=INTERNAL...***	Returns status of potential TDM bus clock timing reference.
swi.EnableSwitch	***SwiHd***	Enables switch block.

Command	Arguments	Description
swi.DisableSwitch	***SwiHd***	Disables switch block.
swi.ConfigStreamSpeed	***SwiHd SPEED=... Streams***	Configures stream speed.
swiGetStreamsBySpeed	***SwiHd SPEED=...***	Returns streams by speed.
swi.ConfigLocalStream	***SwiHd***	Configures device on a local stream.
swi.ConfigLocalStream	***SwiHd***	Configures device on a local stream.
swi.ConfigLocalTimeslot	***SwiHd***	Configures device on a local stream and timeslot.
swi.GetBoardInfo	***SwiHd***	Returns board information.
swi.GetDriverInfo	***SwiHd***	Returns driver information.
swi.GetLocalStreamInfo	***SwiHd***	Returns information about a device on a local stream.
swi.GetLocalTimeslotInfo	***SwiHd***	Returns information about a device on a local stream and timeslot.
swi.CallDriver	***SwiHd command*** [***arg1*** ... ***argN***]	Makes a direct call to the MVIP device driver.
swi.GetLastError	***SwiHd***	Returns the last MVIP device error.
swi.ConfigCarrier	***SwiHd TrkNo*** [frame=D4...] [code=ZCS...]	Loads configuration data for a digital carrier (trunk).
swi.CarrierStatus	***SwiHd TrkNo***	Displays carrier status.
swi.QueryHybridIds	***SwiHd STList***	Retrieves the hybrid IDs.
d95.MakeConnection	***DevHd TiList*** TO ***ToList*** [SIMPLEX|DUPLEX|QUAD]	Connects inputs to outputs.
d95.MakeFramedConnection	***DevHd TiList*** TO ***ToList***	Connects inputs to outputs with identical throughput delay for all connections.
d95.ResetSwitch	***DevHd***	Resets a switching block.
d95.SendPattern	***DevHd pattrn*** TO ***SToList***	Sends a pattern.

Command	Arguments	Description
d95.DisableOutput	***DevHd SToLst***	Disables outputs of switches.
d95.SampleInput	***DevHd STiLst***	Displays data memory.
d95.WatchInput	***DevHd STiLst***	Monitors SToLst continuously.
d95.WatchOutputl	***DevHd SToLst***	Monitors SToLst continuously.
ppx.CreateConnection	***PpxHd = CxName DefaultPattern***	Creates an empty connection.
ppx.OpenConnection	***PpxHd = CxName***	Opens a previously created connection.
ppx.CloseConnection	***PpxHd***	Closes a connection.
ppx.CloseX	***Switch Number***	Closes handle to opened switch.
ppx.DestroyNamed Connection	***PpxHd***	Destroys a named connection.
ppx.SetTalker	***PpxHd Talker***	Sets the talker for a connection.
ppx.AddListeners	***PpxHd ListenerList***	Adds listeners to a connection.
ppx.RemoveListeners	***PpxHd ListenerList***	Removes listeners from a connection.
ppx.SetDefaultPattern	***PpxHd pattern***	Sets the default pattern for a connection.
ppx.Connect	***TiList*** TO ***ToList*** [SIMPLEX | DUPLEX | QUAD]	Makes a connection.
ppx.Disconnect	***ToList*** FROM ***TiList*** [SIMPLEX | DUPLEX | QUAD]	Disconnects a connection.
ppx.Begin	(None)	Starts collecting commands.
ppx.BeginCancel	(None)	Cancels a begin command.
ppx.Submit	(None)	Sends commands to server.
ppx.ShowDB	(None)	Sends the contents of the database to a file.

Note: The **d95** commands are applicable only to MVIP-95 drivers. All stream and timeslot values should be specified in MVIP-95 syntax.

For more information about the...	Refer to the...
swi functions	Specific function in the function reference.
d95 commands	*MVIP-95 Device Driver Standard Manual*
ppx functions	*Point-to-Point Switching Service Developer's Reference Manual*

Example

```
# This SWISH script connects the DSPs to the Line-interfaces on
# a fully populated AG 2000 board.

# Open the switch in MVIP-95 mode with switch state restoration disabled.
swi.OpenSwitch ag2000 = agsw 0 STD=MVIP95 RESTORE=DISABLE

# Reset the switch
swi.ResetSwitch ag2000

# Connect the DSPs to the Line-interfaces,
swi.MakeConnection ag2000 LOCAL:0:0..7 to LOCAL:5:0..7 QUAD

# The same command can be accomplished by typing the following set of commands:
swi.MakeConnection ag2000 LOCAL:0:0..7 to LOCAL:5:0..7
swi.MakeConnection ag2000 LOCAL:4:0..7 to LOCAL:1:0..7
swi.MakeConnection ag2000 LOCAL:2:0..7 to LOCAL:7:0..7
swi.MakeConnection ag2000 LOCAL:6:0..7 to LOCAL:3:0..7

# Close the switch. The connections remain because we opened the switch in
# restoration disabled.
swi.CloseSwitch ag2000
quit
```

Show switch connections: showcx95

Displays switch connections.

Usage

showcx95 [-@***server***][-b***board_number***][***driver_name...***]

where ***driver_name*** is the name of the switching driver for the board you are querying. The following table lists options for using this command:

This command...	Queries...
showcx95	All boards in the system and displays the switch connections.
showcx95 ***driver_name***	Boards with the given switching driver and displays the switch connections.
showcx95 -b***board_number***	The indicated board or boards and displays the switch connections.
showcx95 -@***server***	All boards in the system specified by the server name or IP address. Default = localhost.

Description

showcx95 displays the switch connections for all boards that have MVIP switches. If a pattern is being sent on a timeslot, the pattern value is displayed.

Example

If you have an AG board configured as board 0 with trunk channel 1 connected to local DSP resources (both voice and signaling) and you run the following commands in *swish*:

```
SWISH: openswitch ag = agsw 0
SWISH: resetswitch ag
SWISH: makeconnection ag local:0:4..7 to local:5:0..3
SWISH: makeconnection ag mvip:0:0 to local:1:0 duplex
```

The output displays as:

```
SHOWCX95 Version 1.1                    Jan 15 1998

AGSW 0
 M{00/01}:00                            <-> L{01/00}:00
 L-01:02                                <-      0x7f
 L-01:04..07                            <-      0x7f
 L-03:00                                <-      0x00
 L-03:02                                <-      0x00
 L-03:04..07                            <-      0x00
 L-05:00..03                            <-     L-00:00..04
 L-05:04..07                            <-      0x7f
 L-07:00..07                            <-      0x00
```

In this output example, M means MVIP bus and L means local bus.

This output example shows these types of connections:

```
      L-01:02                       <-      0x7f
```

Pattern 0x7f is sent to timeslot Local:01:02.

```
      L-05:00..03                   <-     L-00:00..04
```

Timeslots Local:00:00..04 are writing to timeslots Local:05:00..03.

```
      M{00/01}:00                   <-> L{01/00}:00
```

Timeslot MVIP:00:00 is writing to timeslot Local:01:00.
Timeslot Local:00:00 is writing to timeslot MVIP:01:00 (a duplex connection).

The syntax of *showcx95* v1.1 for duplex connections is:

```
     L|M{fwd slot1/rev slot1}  <->  L|M{fwd slot2/rev slot2}
```

that represents ***fwd slot1*** -> ***fwd slot2*** and ***rev slot1*** <- ***rev slot2***.

8 Errors

Alphabetical error summary

All Natural Access functions return a status code of SUCCESS (0) or an error code indicating that the function failed and the reason for the failure.

Switching service error codes are defined in the *swidef.h* include file. The error codes are prefixed with SWIERR.

The following table lists the Switching service errors. All errors are 32 bits.

Error name	Hex	Decimal	Description
SWIERR_CONNECTION_NOT_SUPPORTED	0x40011	262161	Switch block does not support the requested connection. A permanent limitation in the switch block prevents the connection.
SWIERR_DEVICE_ERROR	0x40002	262146	MVIP device driver encountered an error while using the services of another device driver.
SWIERR_DLL_INVALID_DEVICE	0x40001	262145	Dynamic link library (DLL) could not find the requested device.
SWIERR_INTERNAL_CONFLICT	0x40010	262160	Switch component of a switch matrix conflicts with another switch component (the state of the switch matrix is ambiguous).
SWIERR_INVALID_CLOCK_PARM	0x40007	262151	Value of a clock configuration parameter is invalid. Enter a valid value.
SWIERR_INVALID_COMMAND	0x40000	262144	Device driver does not support the requested operation.
SWIERR_INVALID_MINOR_SWITCH	0x4000B	262155	Value of the switch parameter is invalid.
SWIERR_INVALID_MODE	0x4000A	262154	Device driver does not support the setting of the mode of an output terminus to the mode specified (for example, **swiMakeFramedConnection** is not supported in all hardware or by MVIP-90 drivers). Enter a valid mode. If you are using **swiMakeFramedConnection** on a board with the H.100 or H.110 bus, ensure that the statement SwitchConnectMode = AllConstantDelay is present in the board keyword file.
SWIERR_INVALID_PARAMETER	0x4000C	262156	Parameter needed by the called function is set to an invalid value. Enter a valid value.

Error name	Hex	Decimal	Description
SWIERR_INVALID_SPEED	0x40008	262152	Invalid stream speed capacity was entered. This functionality works only in MVIP-95. Enter one of the following acceptable speed values: MVIP95_2MBPS_STREAM_SPEED MVIP95_4MBPS_STREAM_SPEED MVIP95_8MBPS_STREAM_SPEED
SWIERR_INVALID_STREAM	0x40004	262148	Value of stream in a terminus element is out of range. Enter a valid value for stream in the terminus element.
SWIERR_INVALID_TIMESLOT	0x40005	262149	Value of timeslot in a terminus element is out of range. Enter a valid value for timeslot in the terminus element.
SWIERR_MISSING_PARAMETER	0x40006	262150	Insufficient number of parameters were provided for the driver to complete the command.
SWIERR_MVIP_BUS_NOT_ENABLED	0x40012	262162	Switching command was called on a switch block on which the MVIP bus was not enabled. CLOCKING.HBUS.CLOCKMODE = Master_A, Master_B, or Slave in the board keyword file for the switch block.
SWIERR_NO_PATH	0x4000E	262158	Device driver is unable to complete the connection because there is blocking or some other temporary switch limitation encountered.
SWIERR_NOT_CONFIGURABLE	0x40009	262153	Device does not support configuration of the requested parameters, values, or both.
SWIERR_SWITCH_VERIFY_ERROR	0x4000F	262159	Verification of the switch operation failed.
SWIERR_UNSUPPORTED_MODE	0x4000D	262157	Mode is not supported by either the device driver or the hardware below the driver.

Numerical error summary

The following table lists the Natural Access Switching service errors:

Hex	Decimal	Error name
0x40000	262144	SWIERR_INVALID_COMMAND
0x40001	262145	SWIERR_DLL_INVALID_DEVICE
0x40002	262146	SWIERR_DEVICE_ERROR
0x40004	262148	SWIERR_INVALID_STREAM
0x40005	262149	SWIERR_INVALID_TIMESLOT
0x40006	262150	SWIERR_MISSING_PARAMETER
0x40007	262151	SWIERR_INVALID_CLOCK_PARM
0x40008	262152	SWIERR_INVALID_SPEED
0x40009	262153	SWIERR_NOT_CONFIGURABLE
0x4000A	262154	SWIERR_INVALID_MODE
0x4000B	262155	SWIERR_INVALID_MINOR_SWITCH
0x4000C	262156	SWIERR_INVALID_PARAMETER
0x4000D	262157	SWIERR_UNSUPPORTED_MODE
0x4000E	262158	SWIERR_NO_PATH
0x4000F	262159	SWIERR_SWITCH_VERIFY_ERROR
0x40010	262160	SWIERR_INTERNAL_CONFLICT
0x40011	262161	SWIERR_CONNECTION_NOT_SUPPORTED
0x40012	262162	SWIERR_MVIP_BUS_NOT_ENABLED
0x40013	262163	SWIERR_UNKNOWN_ERROR

9 Examples

Switching application example

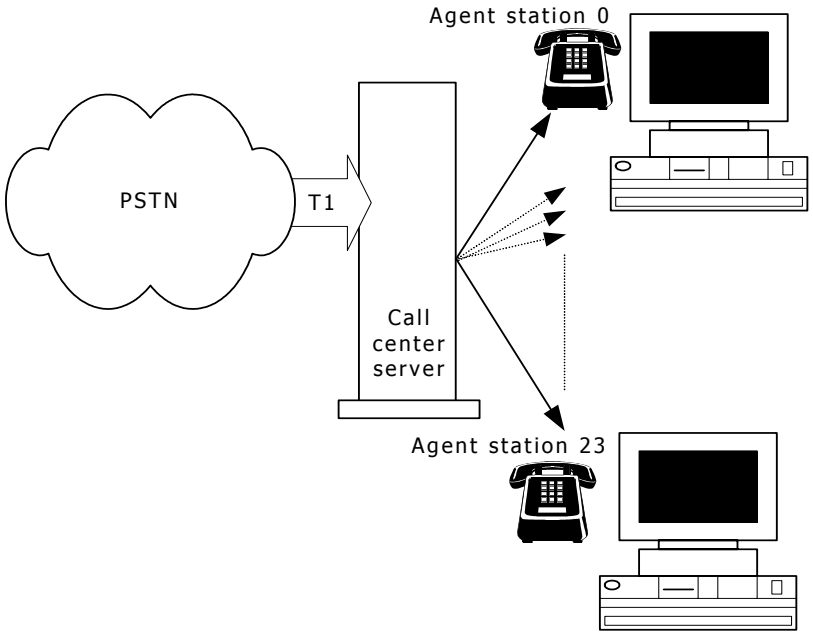
This topic describes how to use the Natural Access Switching service to develop a call center application. The following information is provided:

- Scenario of the example
- Hardware used
- Sample program

Scenario

There are 24 incoming lines with 24 analog operator stations, as shown in the following illustration:

Note: A real application would have more incoming lines than operators.



This program:

1. Accepts a call on an incoming T1 line.
2. Plays a *please hold* message, and sends silence to the caller.

   Note: A real application may do some preparatory work, such as database look-up, before transferring a call to an operator.
3. Plays a *you have a call....* message to the operator (assuming that the operator is already listening).
4. Connects the voice paths of the incoming line and an operator station.
5. Monitors the incoming line for hang-up.
6. Breaks down both ends of the call when either end hangs up.
7. Returns to step 1.

Hardware

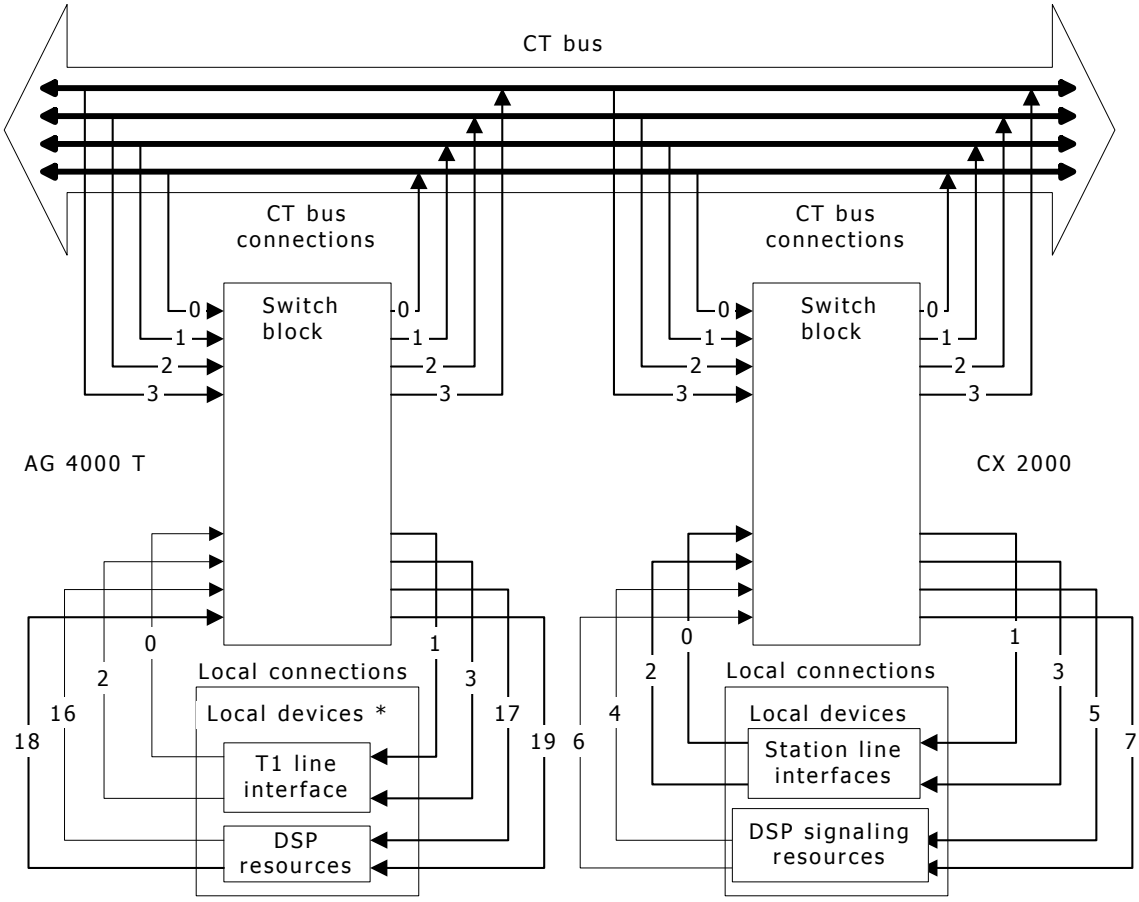
For purposes of this example, the hardware is:

- One AG 4000 T board with as many as 96 digital line interfaces and 96 DSP ports, one for each incoming line.
- One CX 2000 board with up to 48 analog operator station interfaces.

Note: A real call center application may use different hardware configurations.

Sample program

The following illustration shows the initial state of the switch blocks in the system when there are no connections. Stream connections and only timeslot 0 are shown.



** Only trunk 1 is shown.*

1. Open the switching device and access the switch handle for the AG 4000 board and the CX 2000 board. This example assumes that the switches are open in MVIP-95 mode, and that their states are restored on exit.

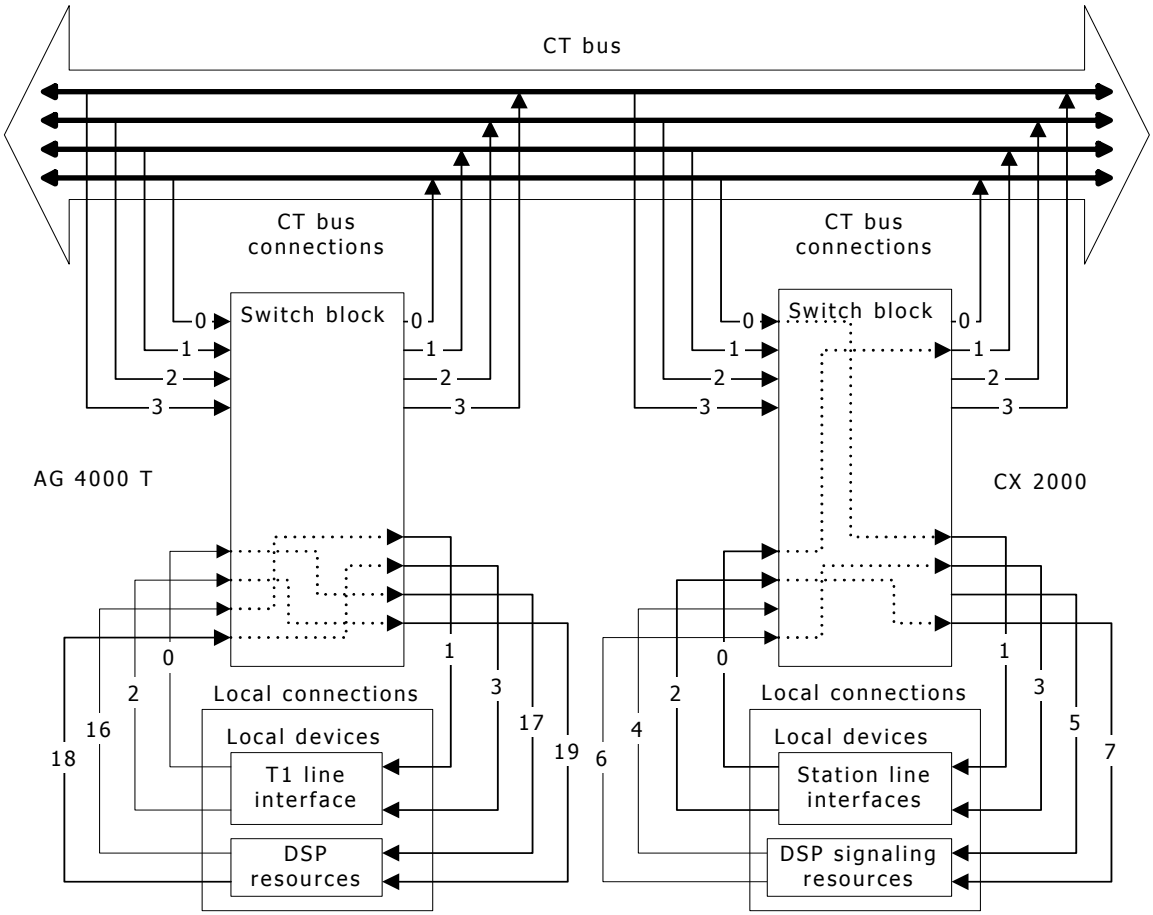
```
void myInitialize(CTAHD ctahd, SWIHD *t1hd, SWIHD *cxhd)
{
    SWI_TERMINUS inputs[24], outputs[24];
    unsigned count;

    /* Open a switch handle to the AG 4000 board */
    swiOpenSwitch(ctahd, "AGSW", 0, SWI_ENABLE_RESTORE, t1hd);

    /* Open a switch handle to the CX 2000 board */
    swiOpenSwitch(ctahd, "CXSW", 0, SWI_ENABLE_RESTORE, cxhd);

    /* Connect the DSPs to the line interfaces on the AG 4000: */

    /* Connect the voice streams: */
    /* Connect network XMIT to DSP RCV */
    for (count = 0; count < 24; count++)
    {
        inputs[count].bus = MVIP95_LOCAL_BUS;
        inputs[count].stream = 0;
        inputs[count].timeslot = (DWORD)count;

        outputs[count].bus = MVIP95_LOCAL_BUS;
        outputs[count].stream = 17;
        outputs[count].timeslot = (DWORD)count;
    }
    /* Make output address(es) read from input address(es) */
    swiMakeConnection(*t1hd, inputs, outputs, count);

    /* Connect DSP XMIT to network RCV */
    for (count = 0; count < 24; count++)
    {
        inputs[count].bus = MVIP95_LOCAL_BUS;
        inputs[count].stream = 16;
        inputs[count].timeslot = (DWORD)count;

        outputs[count].bus = MVIP95_LOCAL_BUS;
        outputs[count].stream = 1;
        outputs[count].timeslot = (DWORD)count;
    }
    swiMakeConnection(*t1hd, inputs, outputs, count);

    /* Connect the signaling streams: */
    /* Connect network XMIT to DSP RCV */
    for (count = 0; count < 24; count++)

    {
        inputs[count].bus = MVIP95_LOCAL_BUS;
        inputs[count].stream = 2;
        inputs[count].timeslot = (DWORD)count;


        outputs[count].bus = MVIP95_LOCAL_BUS;
        outputs[count].stream = 19;
        outputs[count].timeslot = (DWORD)count;
    }
    swiMakeConnection(*t1hd, inputs, outputs, count);

    /* Connect DSP XMIT to network RCV */
    for (count = 0; count < 24; count++)
    {
        inputs[count].bus = MVIP95_LOCAL_BUS;
        inputs[count].stream = 18;
        inputs[count].timeslot = (DWORD)count;

        outputs[count].bus = MVIP95_LOCAL_BUS;
        outputs[count].stream = 3;
```

```
        outputs[count].timeslot = (DWORD)count;
    }
    swiMakeConnection(*t1hd, inputs, outputs, count);
}
    /* Connect the DSPs to the line interfaces on the CX 2000 (signaling only)*/
    /* Connect the signaling streams: */
    /* Connect network XMIT to DSP RCV */
    for (count = 0; count < 24; count++)
    {
        inputs[count].bus = MVIP95_LOCAL_BUS;
        inputs[count].stream = 2;
        inputs[count].timeslot = (DWORD)count;

        outputs[count].bus = MVIP95_LOCAL_BUS;
        outputs[count].stream = 7;
        outputs[count].timeslot = (DWORD)count;
    }
    swiMakeConnection(*t1hd, inputs, outputs, count);

    /* Connect DSP XMIT to network RCV */
    for (count = 0; count < 24; count++)
    {
        inputs[count].bus = MVIP95_LOCAL_BUS;
        inputs[count].stream = 6;
        inputs[count].timeslot = (DWORD)count;

        outputs[count].bus = MVIP95_LOCAL_BUS;
        outputs[count].stream = 3;
        outputs[count].timeslot = (DWORD)count;
    }
```

2. Using switching, nail up the CX 2000 lines to the MVIP bus on streams 0 and 1.

```
/* Nails up CX 2000 lines to the MVIP bus */
void myConnectAgCx(SWIHD cxhd)
{
    SWI_TERMINUS outputs[24], inputs[24];
    unsigned count;

    /* Connect Voice lines: */

    /* Connect CX2000 local:0:0..23 to mvip:1:0..23 */
    for (count = 0; count < 24; count++)
    {
        inputs[count].bus = MVIP95_LOCAL_BUS;
        inputs[count].stream = 0;
        inputs[count].timeslot = (DWORD)count;

        outputs[count].bus = MVIP95_MVIP_BUS;
        outputs[count].stream = 1;
        outputs[count].timeslot = (DWORD)count;
    }
    swiMakeConnection(cxhd, inputs, outputs, count);

    /* Connect CX2000 mvip:0:0..23 to local:1:0..23 */
    for (count = 0; count < 24; count++)
    {
        inputs[count].bus = MVIP95_MVIP_BUS;
        inputs[count].stream = 0;
        inputs[count].timeslot = (DWORD)count;

        outputs[count].bus = MVIP95_LOCAL_BUS;
        outputs[count].stream = 1;
        outputs[count].timeslot = (DWORD)count;
    }
    swiMakeConnection(cxhd, inputs, outputs, count);
}
```

The following illustration shows the state of the switches in the system:



The program returns to this state after each call.

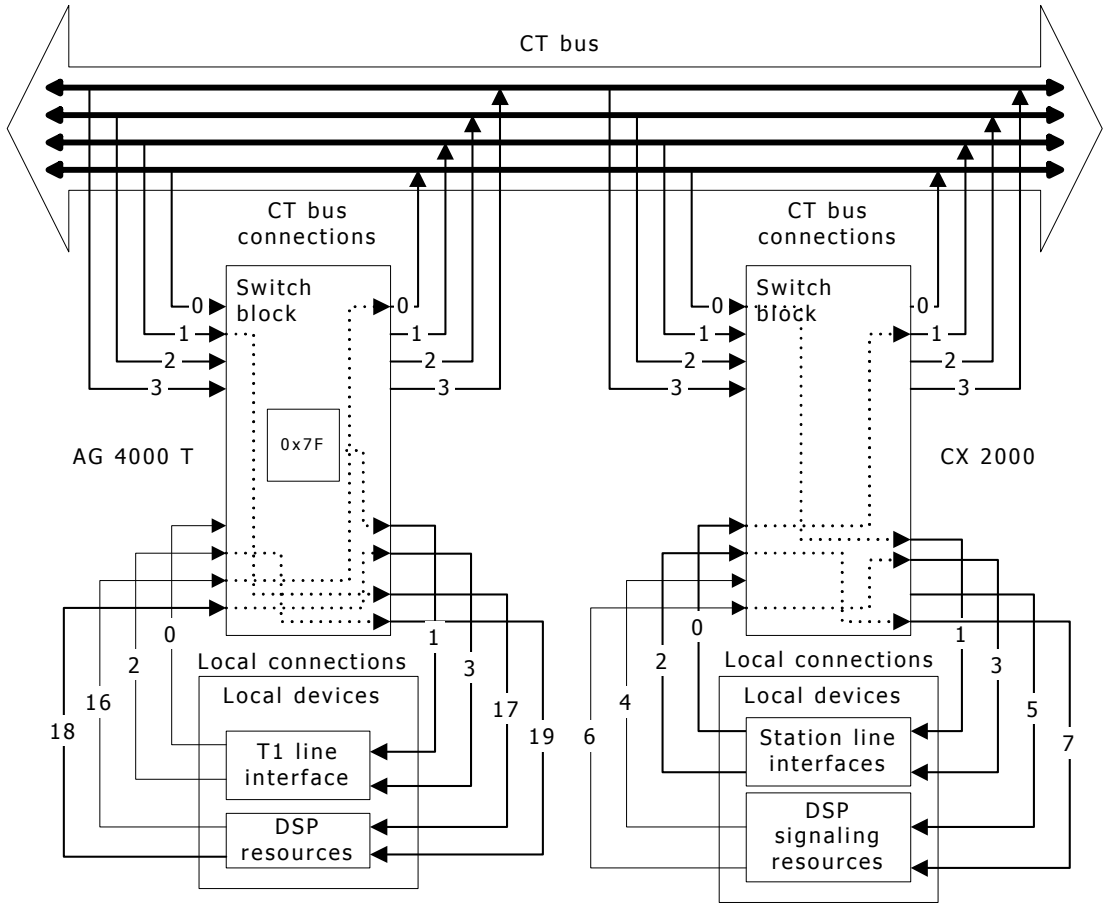
3. Wait for incoming calls on the incoming lines using the Natural Call Control service.
4. When a call comes in, use the Natural Call Control service to perform call control. Once the T1 call is connected, play *Please hold* using the Voice Message service and send a silence pattern to the incoming line using the Switching service.

```
/* Send Silence to incoming T1 call */
void mySendSilence(SWIHD t1hd, DWORD timeslot)
{
    SWI_TERMINUS output;
    BYTE pattern = 0x7F; /* Silence pattern for MU-law */

    /* Send silence pattern to incoming call via local:1:timeslot */
    output.bus = MVIP95_LOCAL_BUS;
    output.stream = 1;
    output.timeslot = timeslot;
    swiSendPattern(t1hd, &pattern, &output, 1);
}
```

5. Connect the DSP port assigned to the incoming call to the CX 2000 operator station. (This step assumes the battery and the bit detector are enabled on the CX 2000 ports.) Notify the operator of the incoming call using the Voice Message service.

```
/* Connect T1 DSP to CX 2000 station, DUPLEX connection via the CT bus,
 * in preparation for playing the "here comes a call" message.
 */
void myConnect4KCx(SWIHD t1hd, DWORD timeslot)
{
    SWI_TERMINUS output, input;

/* Connect AG4000 T1 local:16:timeslot to mvip:0:timeslot */
    input.bus = MVIP95_LOCAL_BUS;
    input.stream = 16;
    input.timeslot = timeslot;

    output.bus = MVIP95_MVIP_BUS;
    output.stream = 0;
    output.timeslot = timeslot;
    swiMakeConnection(t1hd, &input, &output, 1);

/* Connect AG4000 T1 mvip:1:timeslot to local:17:timeslot */
    input.bus = MVIP95_MVIP_BUS;
    input.stream = 1;
    input.timeslot = timeslot;

    output.bus = MVIP95_LOCAL_BUS;
    output.stream = 17;
    output.timeslot = timeslot;
    swiMakeConnection(t1hd, &input, &output, 1);
}
```

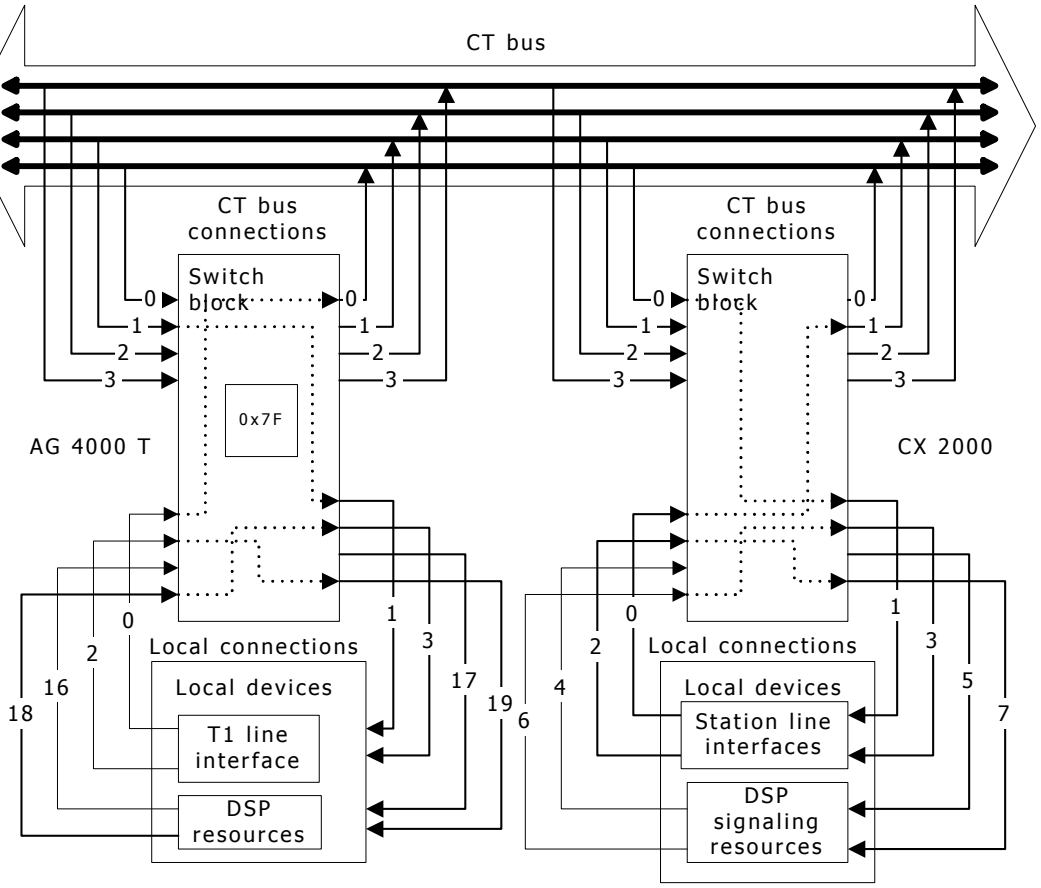
The following illustration shows the state of the switches in the system:



6. Connect the voice paths of the AG 4000 T1 line and the CX 2000 operator station so that the caller and the operator can carry on a conversation. To accomplish this task, connect the AG 4000 board's voice streams to the CX 2000 board's operator voice streams.

```
/* Connect Voice paths of CX 2000 station and
 * the incoming call, DUPLEX connection.
 */
void myConnectVoice(SWIHD t1hd, DWORD timeslot)
{
    SWI_TERMINUS output, input;

    /* Connect AG 4000 T1 local:0:timeslot to mvip:0:timeslot */
    input.bus = MVIP95_LOCAL_BUS;
    input.stream = 0;
    input.timeslot = timeslot;

    output.bus = MVIP95_MVIP_BUS;
    output.stream = 0;
    output.timeslot = timeslot;

    swiMakeConnection(t1hd, &input, &output, 1);

    /* Connect AG 4000 T1 mvip:1:timeslot to local:1:timeslot */
    input.bus = MVIP95_MVIP_BUS;
    input.stream = 1;
    input.timeslot = timeslot;

    output.bus = MVIP95_LOCAL_BUS;
    output.stream = 1;
    output.timeslot = timeslot;

    swiMakeConnection(t1hd, &input, &output, 1);
}
```

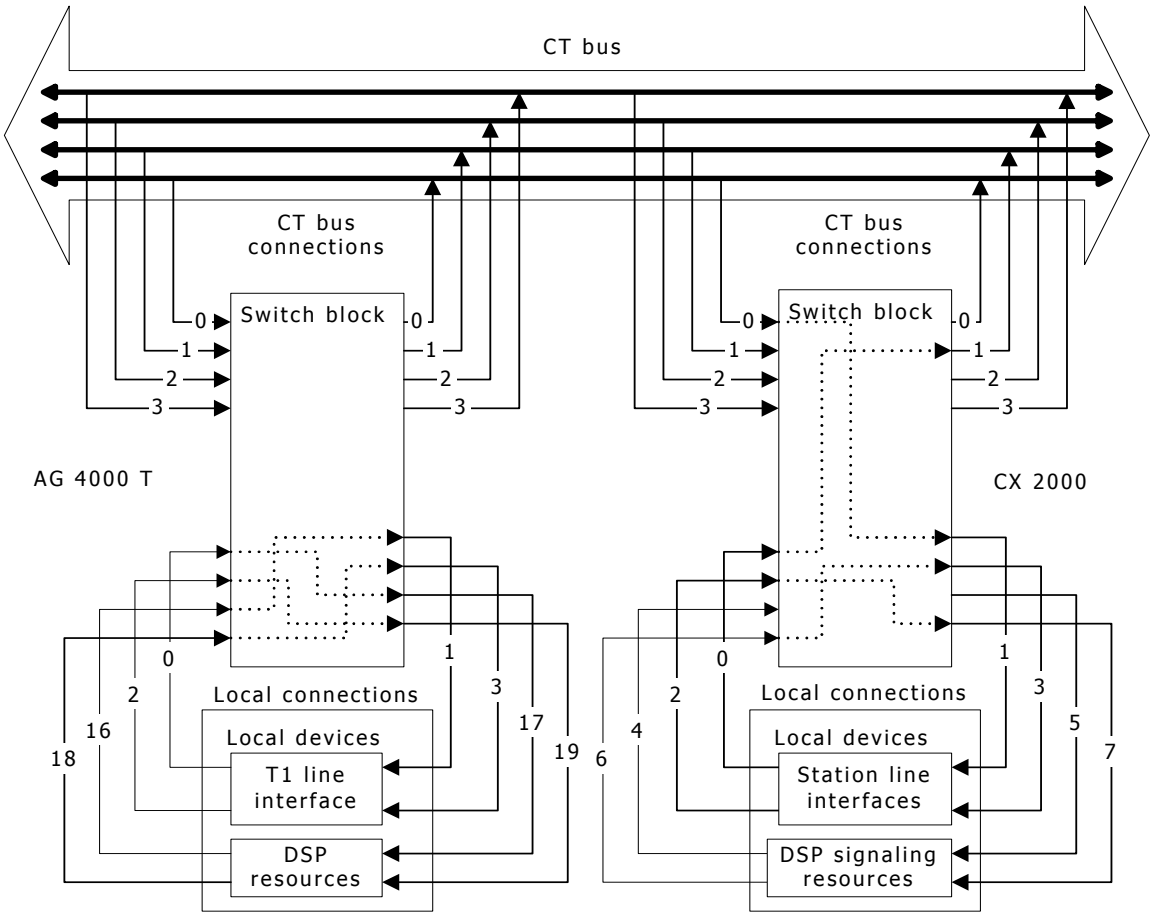
The following illustration shows the state of the switches in the system:



7. If either end hangs up, disconnect the call using the Natural Call Control service. Reconnect the AG 4000 board's DSP streams to the AG 4000 board's line interface streams.

```
/* Reset paths to get ready for next incoming call.
 */
void myResetT1Line(SWIHD t1hd, DWORD timeslot)
{
    SWI_TERMINUS output, input;

    /* Disable outputs of switch to mvip:0:timeslot */
    output.bus = MVIP95_MVIP_BUS;
    output.stream = 0;
    output.timeslot = timeslot;

    swiDisableOutput(t1hd, &output, 1);

    /* Disable outputs of switch to local:1:timeslot */
    output.bus = MVIP95_LOCAL_BUS;
    output.stream = 1;
    output.timeslot = timeslot;

    swiDisableOutput(t1hd, &output, 1);

    /* Reconnect voice streams */
    input.bus = MVIP95_LOCAL_BUS;
    input.stream = 0;
    input.timeslot = timeslot;

    output.bus = MVIP95_LOCAL_BUS;
    output.stream = 17;
    output.timeslot = timeslot;

    swiMakeConnection(t1hd, &input, &output, 1);

    input.bus = MVIP95_LOCAL_BUS;
    input.stream = 16;
    input.timeslot = timeslot;

    output.bus = MVIP95_LOCAL_BUS;
    output.stream = 1;
    output.timeslot = timeslot;

    swiMakeConnection(t1hd, &input, &output, 1);
```

The application returns to the initial state, as shown in the following illustration:



8. At the end of the program, call **swiCloseSwitch** to close all the switches and restore the state of the switch blocks.

```
/* Close open switches */
void myShutdown(SWIHD t1hd, SWIHD cxhd)
{
    swiCloseSwitch(t1hd);
    swiCloseSwitch(cxhd);
}
```

H.110 clock configuration example

In this example, there are four NMS CompactPCI boards in the system. The first board is the primary master that drives the H.110 A clock from the PSTN network connection. The second board is the secondary master that drives the H.110 B clock from the H.110 A clock. The second board uses a PSTN network connection as an alternate clock source when a fallback to the B clock occurs. The third board drives the NETREF_1 signal from its own PSTN network connection. The fourth board drives the NETREF_2 signal from its own PSTN network connection.

If the primary clock master fails, fallback to a secondary master (B clock) can be programmed to be automatic. In this example, the secondary master uses its own PSTN interface to drive the B clocks. The secondary master can also be programmed to use either of the NETREF signals if so desired.

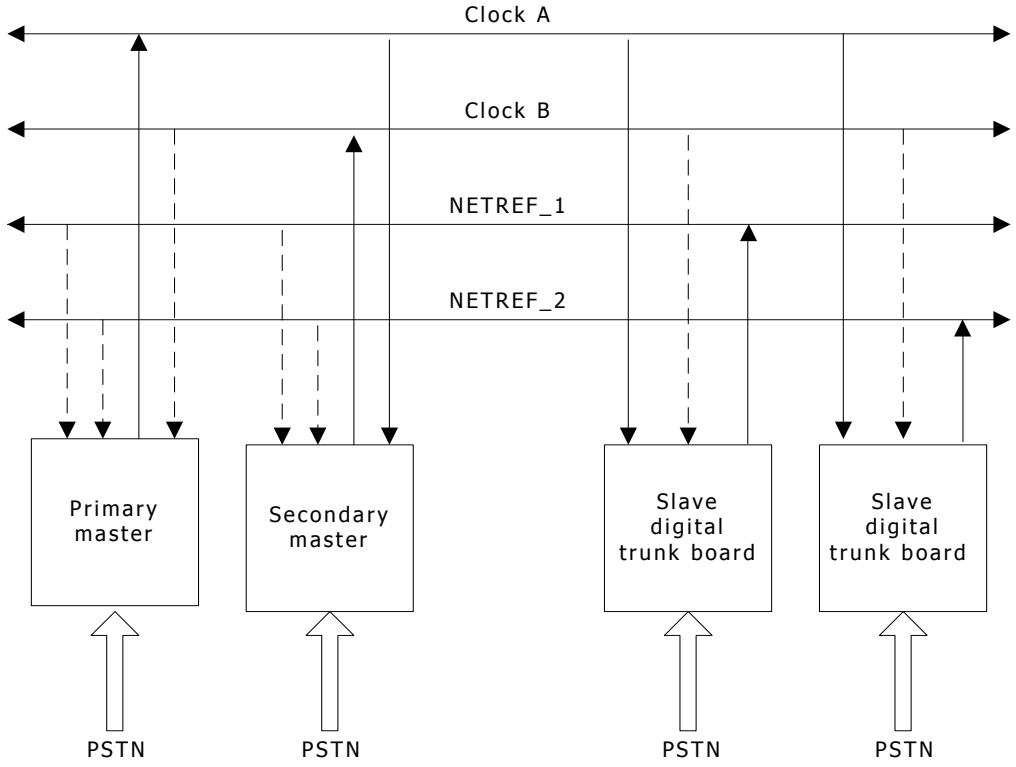
Note: This example is applicable to the NMS CompactPCI AG, CG, and CX boards.

This example:

- Configures the board clock of the first CompactPCI board to synchronize to the PSTN clocks and become the H.110 A clock master.
- Configures the board clock of the second CompactPCI board to synchronize to the A clocks and become the H.110 B clock master. Its fallback source is its network connection.
- Configures the third and fourth boards to slave off the A clocks.
- Configures the third and fourth boards to drive NETREF_1 and NETREF_2 respectively.
- Sets the automatic fallback feature to allow for a fallback from the primary to the secondary master.

For information about configuring the clock fallback feature, refer to the *NMS OAM System User's Manual*. For more information about H.110 A clocks and B clocks, refer to the *ECTF: H.110 Revision 1.0 Hardware Compatibility Specification: CT Bus*.

The following illustration shows the example of clock fallback:



1. Configure the board clock of the first CompactPCI board to synchronize to the PSTN clocks and become the primary clock master (A clock).

```
SWI_CLOCK_ARGS boardclock;

/* Make first board sync off the PSTN digital trunk 1 */
boardclock.size = sizeof(SWI_CLOCK_ARGS);
boardclock.clocktype = MVIP95_H100_CLOCKING;
boardclock.clocksource = MVIP95_SOURCE_NETWORK;
boardclock.network = 1;
boardclock.ext.h100.h100clockmode= MVIP95_H100_MASTER_A;
boardclock.ext.h100.autofallback= MVIP95_H100_ENABLE_AUTO_FB;
boardclock.ext.h100.netrefclockspeed= MVIP95_H100_NETREF_8KHZ;
boardclock.ext.h100.fallbackclocksource= MVIP95_SOURCE_NETWORK;
boardclock.ext.h100.fallbacknetwork = 2;

swiConfigBoardClock(firstt1, &boardclock);
```

2. Configure the board clock of the second CompactPCI board to synchronize to the PSTN clocks and become the secondary clock master (B clock).

```
SWI_CLOCK_ARGS boardclock;

boardclock.size = sizeof(SWI_CLOCK_ARGS);
boardclock.clocktype = MVIP95_H100_CLOCKING;
boardclock.clocksource = MVIP95_SOURCE_H100_A;
boardclock.network = 0;
boardclock.ext.h100.h100clockmode= MVIP95_H100_MASTER_B;
boardclock.ext.h100.autofallback= MVIP95_H100_ENABLE_AUTO_FB;
boardclock.ext.h100.netrefclockspeed= MVIP95_H100_NETREF_8KHZ;
boardclock.ext.h100.fallbackclocksource= MVIP95_SOURCE_NETWORK;
boardclock.ext.h100.fallbacknetwork = 1;

swiConfigBoardClock(secondt1, &boardclock);
```

3. Configure the board clocks of the third and fourth CompactPCI boards to slave to the H.110 A clocks.

```
SWI_CLOCK_ARGS boardclock3;
SWI_CLOCK_ARGS boardclock4;


boardclock3.size = sizeof(SWI_CLOCK_ARGS);
boardclock3.clocktype = MVIP95_H100_CLOCKING;
boardclock3.clocksource = MVIP95_SOURCE_H100_A;
boardclock3.network = 0;
boardclock3.ext.h100.h100clockmode= MVIP95_H100_SLAVE;
boardclock3.ext.h100.autofallback= MVIP95_H100_ENABLE_AUTO_FB;
boardclock3.ext.h100.netrefclockspeed= MVIP95_H100_NETREF_8KHZ;

boardclock4.size = sizeof(SWI_CLOCK_ARGS);
boardclock4.clocktype = MVIP95_H100_CLOCKING;
boardclock4.clocksource = MVIP95_SOURCE_H100_A;
boardclock4.network = 0;
boardclock4.ext.h100.h100clockmode= MVIP95_H100_SLAVE;
boardclock4.ext.h100.autofallback= MVIP95_H100_ENABLE_AUTO_FB;
boardclock4.ext.h100.netrefclockspeed= MVIP95_H100_NETREF_8KHZ;


swiConfigBoardClock(thirdt1, &boardclock3);
swiConfigBoardClock(fourtht1, &boardclock4);
```

4. Configure the third and fourth CompactPCI boards to drive the NETREF_1 and NETREF_2 signals on the H.110 bus from their PSTN connection.

```
SWI_NETREF_CLOCK_ARGSnetref_1;
SWI_NETREF_CLOCK_ARGSnetref_2;

netref_1.size = sizeof(SWI_NETREF_CLOCK_ARGS);
netref_1.network = 1;
netref_1.netref_clock_mode = MVIP95_H100_NETREF_1;
netref_1.netref_clock_speed = MVIP95_H100_NETREF_8KHZ;


netref_2.size = sizeof(SWI_NETREF_CLOCK_ARGS);
netref_2.network = 1;
netref_2.netref_clock_mode = MVIP95_H100_NETREF_2;
netref_2.netref_clock_speed = MVIP95_H100_NETREF_8KHZ;

swiConfigNetrefClock(thirdt1, netref_1);
swiConfigNetrefClock(fourtht1, netref_2);
```

Index