PMC-Sierra

PM8610, PM8611, PM8620, PM8621

NSE/SBS
NARROWBAND CHIPSET DRIVER

DRIVER MANUAL

PROPRIETARY AND CONFIDENTIAL

PRELIMINARY

ISSUE 1: AUGUST, 02

LEGAL INFORMATION

Copyright

Disclaimer

None of the information contained in this document constitutes an express or implied warranty by PMC-Sierra, Inc. as to the sufficiency, fitness or suitability for a particular purpose of any such information or the fitness, or suitability for a particular purpose, merchantability, performance, compatibility with other parts or systems, of any of the products of PMC-Sierra, Inc., or any portion thereof, referred to in this document. PMC-Sierra, Inc. expressly disclaims all representations and warranties of any kind regarding the contents or use of the information, including, but not limited to, express and implied warranties of accuracy, completeness, merchantability, fitness for a particular use, or non-infringement.

In no event will PMC-Sierra, Inc. be liable for any direct, indirect, special, incidental or consequential damages, including, but not limited to, lost profits, lost business or lost data resulting from any use of or reliance upon the information, whether or not PMC-Sierra, Inc. has been advised of the possibility of such damage.

CONTACTING PMC-SIERRA

PMC-Sierra
8555 Baxter Place Burnaby, BC
Canada V5A 4V7

Tel: +1-604-415-6000
Fax: +1-604-415-6200

Document Information: document@pmc-sierra.com
Corporate Information: info@pmc-sierra.com
Technical Support: apps@pmc-sierra.com
Web Site: http://www.pmc-sierra.com

OVERVIEW

Scope

This document is the driver manual for the NSE/SBS Narrowband Chipset (PM8610, PM8611, PM8620, PM8621) driver software. It describes the features and functionality provided by the chipset driver, the software architecture, and the external interface of the chipset driver software. The document also describes how the chipset driver can be ported to a different platform.

Objectives

The main objectives of this document are as follows:

- To provide a detailed list of the chipset driver's features
- To describe the software architecture of the driver (e.g., data structures, state diagrams and function descriptions)
- To describe the external interface of the driver; this interface illustrates how the chipset driver interacts with the underlying hardware devices, the RTOS, and the external application software.

References

The main references for this document are as follows:

- Narrowband Chipset System Architecture, Issue 1, PMC-2000413 (PMC-Sierra, Inc.)
- SBI Bus Serializer Data Sheet, Issue 5, PMC- 2000168 (PMC-Sierra, Inc.)
- NSE Data Sheet, Issue 5, PMC- 2000170 (PMC-Sierra, Inc.)
- SBS and SBSLITE Device Driver Manual, Issue 3, PMC-2011471 (PMC-Sierra, Inc.)
- NSE-20G and NSE-8G Device Driver Manual, Issue 2, PMC-2010053 (PMC-Sierra, Inc.)
- ANSI – T1.105 – 1995, "Synchronous Optical Network (SONET) – Basic Description including Multiplex Structure, Rates, and Formats", 1995
- ITU – G.707 – 2000, "Network Node Interface for the Synchronous Digital Hierarchy (SDH)", 2000
- NSE/SBS Open Path Algorithm API Design Specification, Issue 1, PMC-2010601 (PMC-Sierra, Inc)
- A Survey of Rollback-recovery Protocols in Message-Passing Systems, Elnozahy, M., Alvisi, L., Wang, Y., and Johnson, D., CMU-CS-99-148, Carnegie Mellon University, 1999.

- CHESS-NB Designing a Non-blocking Fabric for 1:2 Multicast, Issue 2, PMC-2020050 (PMC-Sierra Inc.)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

The following sections of the Narrowband Chipset Device Driver Manual describe the Narrowband Chipset device driver. The chipset driver is written in ANSI-C programming language to promote greater driver portability to other embedded hardware and Real Time Operating System environments.

Section 2 gives an overview of the Narrowband chipset and the main features provided by the chipset driver to the user from an application perspective. Section 3 defines the software architecture of the Narrowband Chipset driver. It also includes a discussion of the driver's external interface and its main components. The Data Structure information in Section 4 describes the elements of the driver that either configure or control its behavior. Included here are the constants, variables, and structures that the Narrowband Chipset device driver uses to store initialization, configuration, and status information. Section 5 provides a detailed description of each function that is a member of the Narrowband Chipset driver Application Programming Interface (API). This section outlines function calls that hide device-specific details and application callbacks that notify the user of significant events.

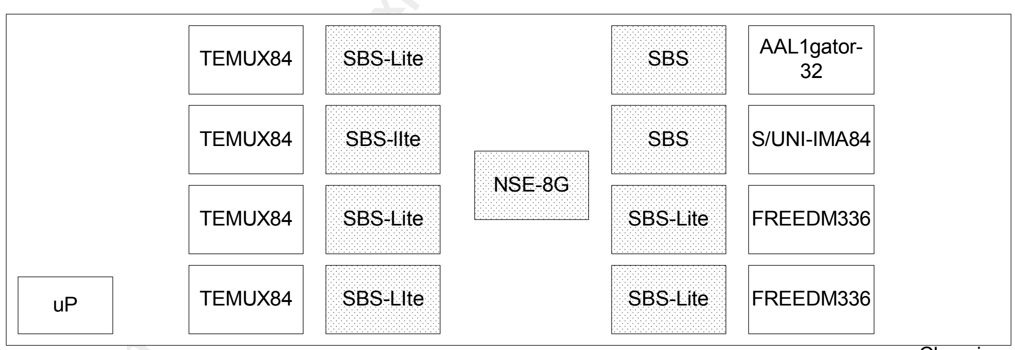
For your convenience, Section 8 of this manual provides a brief guide for porting the Narrowband Chipset driver to your hardware and RTOS platform. In addition, an extensive Appendix (page 178) and Index (page 257) provides you with useful reference information.

2 NARROWBAND CHIPSET OVERVIEW

The NSE/SBS narrowband chipset forms a time-space-time switching fabric capable of cross connecting traffic down to DS0 granularity. It scales the existing SBI family products from OC-3 rates to OC-12 and OC-48 rates for the Any-Service-Any-Port type (ASAP) multi-service equipment. The chipset not only supports SBI bus devices but also TeleCombus devices. In SBI mode, it provides DS0 granularity time-space-time switching fabric and in TeleCombus mode, it provides a VT1.5/VT2 (TU-11/TU-12) granularity. The chipset also provides a seamless integration to the CHESS family of PMC-Sierra Inc. The addition of NSE/SBS to the CHESS-enabled devices allows user to perform VT-level cross connect functions. Targeted applications for the chipset include OC-48 ADM, and channelized OC-48/4xOC-12 ASAP architecture for carrier class products.

The Narrowband Chipset Driver (CSD) presents a unified interface for the chipset under all different configurations and provides a synchronized access and coordinated control over the underlying devices (PM8620/1 NSE-20/8G, and PM8610/1 SBS/SBSLITE) for Narrowband switching applications. The main functionality includes configuration of chipset devices, connection setup/maintenance/teardown, 1:1 and 1:N port protection, and also UPSR protection. In addition, the chipset driver is designed to handle various NSE/SBS switching fabric configurations including multi-stage fabric architecture that supports higher bandwidth traffic (Note: The CSD software is designed and implemented to accommodate 1- or 3-stage fabrics; however, the OPA library is implemented to support only 1-stage fabric. Therefore, *only* 1-stage fabric is currently supported).

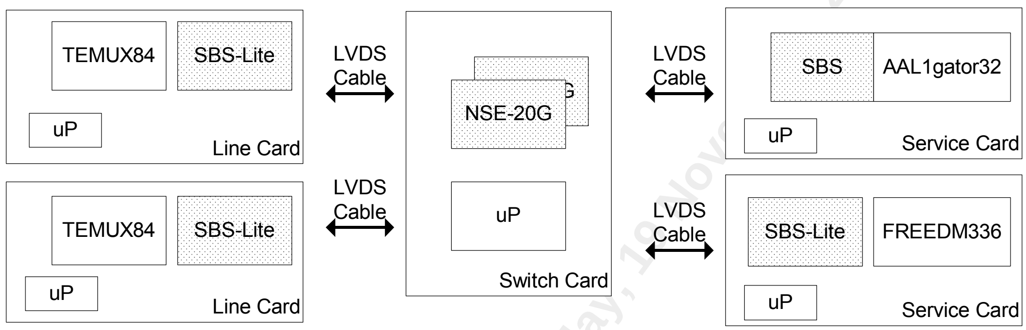
Figure 1: NSE/SBS Switching Fabric - Centralized System Model



The chipset driver provides a high level of abstraction to the user for using the NSE/SBS switching fabric. It is built on top of the following software components- the NSE and SBS device drivers for accessing the underlying NSE and SBS devices, and the narrowband Open Path Algorithm library (OPA Library). Central to this library is the open path algorithm (OPA) engine that provides the intelligence for establishing calls and generating connection map settings for all individual devices in the fabric.

The chipset driver is designed to be flexible and can easily be adapted to various system configurations. Some of the typical system configurations are shown in the following. A typical centralized configuration (Figure 1) has all SBS(s) and NSE(s) under the control of one microprocessor that also runs the OPA library; a distributed configuration (Figure 2) has SBS(s) controlled by one microprocessor and then the NSE(s) by a different microprocessor, which may also host the OPA library optionally. Additional microprocessor can be deployed to host the OPA library for dedicated processing.

Figure 2: NSE/SBS Switching Fabric - Distributed System Model



In-band communication links via LVDS between SBSs and NSEs are provided to facilitate inter-device (or inter-card) communication. In a distributed configuration, one of the intended applications is to distribute switch fabric settings from the NSE side to the remote SBSs. The chipset driver provides functionality to receive and transmit messages using these links. However, the definition of the in-band link message content/format and the design of any additional communication protocol running on top of the link are beyond the scope of the chipset driver and are left to the user application.

Centralized and Distributed System Configurations

In a centralized configuration, the chipset driver API interacts with all NSE and SBS devices that are controlled by one microprocessor. In a distributed configuration, multiple instances of the chipset driver can be deployed to run under multiple microprocessors. The configuration of each of the instances varies depending on whether it is a line or switch card. The interface for all configurations remains the same for the applicable features.

In a distributed configuration, each CSD instance is configured according to the devices under its control. For instance, a typical line card may consist of some PHY layer devices and one or more SBS devices. This CSD will then be configured to run just the SBS device driver and all the physical SBS devices on the line card. In a typical switch card that consists of NSE devices, the CSD instance will be configured to run with the NSE device driver and/or the OPA. All NSE devices will be registered (added) as physical devices attached to the card and the remote SBS devices will be registered as logical devices.

- It is important to note that typically, only one CSD instance should be configured to run the OPA. (One exception is in the case of a 1+1 redundant fabric system where a second (standby) copy of the OPA is allowed to run in parallel with the working copy.) The OPA schedules calls in the fabric and keeps the record of all existing calls. It therefore acts as the central repository for all the records. Since this CSD/OPA has to be aware of all devices in the fabric, devices that are not local to this CSD/OPA must be added as "logical" devices. This allows the CSD/OPA to properly recognize the existence of the SBS or NSE devices in the fabric, though not directly under its control.

Scalability

The CSD is designed to support 1- or 3-stage fabrics for applications that require higher bandwidth (Figure 3 illustrates a 3-stage, and 2-depth fabric composed of NSE-20G and SBS lite devices). The maximum bandwidth for a 3-stage fabric is 640 Gbps for NSE-20G fabric and 96 Gbps for NSE-8G fabric. There is no real theoretical limitation to a large switching core other than physical limitation such as board size or heat dissipation. Please refer to Table 1 (NSE-20G) and Table 2 (NSE-8G) for other possible configurations for a switching fabric with DS0 granularity. The current CSD implementation supports *only* 1-stage fabric.

Figure 3: Stage-3 Switch Fabric: 64 SBSLITEs, 6 NSE-20G, Bandwidth = 40 Gbps

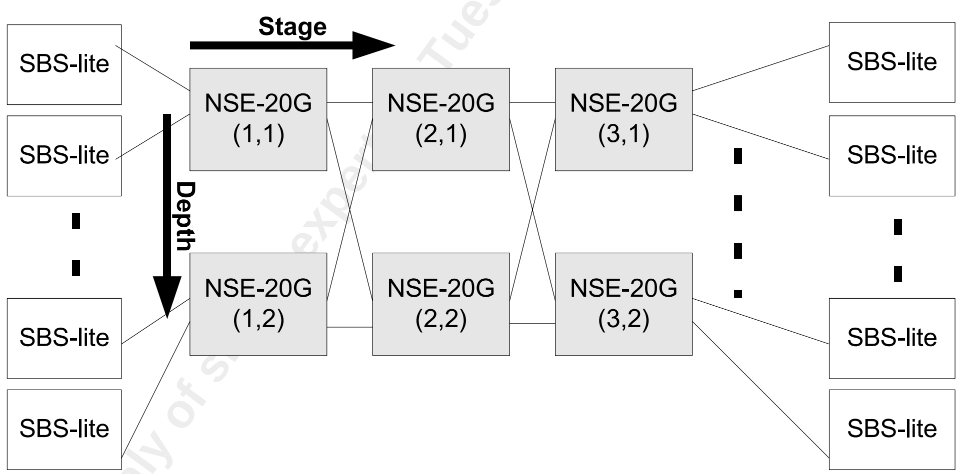


Table 1: Narrowband Chipset Scalable Fabric with NSE-20G

Stage	Depth	NSE-20G	SBS	Bandwidth (Gbps)
1	1	1	32	20
3	2	6	64	40
3	4	12	128	80

3	8	24	256	160
3	16	48	512	320
3	32	96	1024	640

Table 2: Narrowband Chipset Scalable Fabric with NSE-8G

Stage	Depth	NSE-8G	SBS	Bandwidth (Gbps)
1	1	1	12	8
3	2	6	24	16
3	3	9	36	24
3	4	12	48	32
3	6	18	72	48
3	12	36	144	96

On the line/service card side, the CSD supports the addition/deletion of new chipset components into the system without affecting the operation of the existing fabric. For instance, a new line card with SBS devices can be added to the CSD when there is additional link requirements. Such an action does not affect the operation of the existing links. Similarly, a faulty line card can be taken out of service without affecting the rest of the system. The assumption is that the chipset system is built on a backplane technology with hot-swap capability.

TeleCombus and SBI Bus Mode Switching

In TeleCombus mode, the switching granularity is VT1.5/VT2. The CSD takes advantage of the column-switching mode available in the hardware to support the switching operation. The advantage is simpler and faster connection setup since connection setting is specified for the entire column, not just on a per-DS0 basis. Each SBS in the chipset can be set up for quad 19.44 MHz TeleCombus or single 77.76 MHz TeleCombus operation.

In SBI mode, the finest switching granularity is DS0. Each SBS device can be configured to work in a quad SBI (19.44 MHz) or single SBI336 (77.76 MHz) mode. In addition, CAS processing can be enabled/disabled on a per-tributary basis for DS0s with CAS bytes associated with them. Alternately, the CSD can set up the chipset to operate in a column-switching mode, similar to the TeleCombus mode if no DS0 routing is expected.

TeleCombus and SBI Bus Tributary Naming Convention

In SBI mode, the tributary payload type on the SBS has to be properly defined for the chipset to function properly. SBI336 column multiplexed 4 SBI buses together. It complies with most of the SBI bus specification. Each SBI bus consists of 3 SPEs and each SPE can consist of T1, E1, TVT1.5, TVT2, DS3/E3, and fractional T1/E1 type payloads. Traffic type cannot be mixed within one SPE but can be mixed independently across SPEs. For instance, the first SPE can carry 28 T1s (Note: T1 and TVT1.5 can be mixed within one SPE), the second SPE can carry 21 E1s and the third can be defined for one DS3.

In either bus mode, the user specifies the VTs to be switched across the fabric. For instance, the user need only specify the source and destination SBS, SBI bus number, SPE number, and the T1 number to route the entire T1 across the fabric. In TeleCombus mode, SONET specification is used to label the virtual tributaries specified by the STS-3 (ranges from 1 to 4) and STS-1 (ranges from 1 to 3) numbers, the VT group number ranges from 1 to 7, and the VTx number where x = 1.5, 2, 3, and 6. The tributary number for VT1.5, VT2, VT3, and VT6 varies from 1 to 4, 3, 2, and 1 respectively (Figure 4). Note that there can only be one type of VT defined in one particular VT group.

SDH naming convention may also be employed in place of SONET. The AUG-1, AU-3, and TUG-2 replace the STS-3, STS-1 and VT group numbers. The VTs are replaced by TU-11, TU-12 and TU-2 that correspond to the VT1.5, VT2, and VT6 in SONET (Note: There is no SDH equivalent of SONET's VT3). This naming convention applies to frames with AU-3 structure. For AU-4 structured frame, all are the same except the TUG-3 number replaces the AU-3 number.

Table 3: SONET vs. SDH Virtual Tributary Naming Convention in TeleCombus

SONET	SDH AU-3 structured frame	SDH AU-4 structured frame
STS-3 number	AUG-1 number	AUG-1 number
STS-1 number	AU-3 number	TUG-3 number
VT group number	TUG-2 number	TUG-2 number
Virtual Tributary number:	Tributary Unit (TU) number	Tributary Unit (TU) number

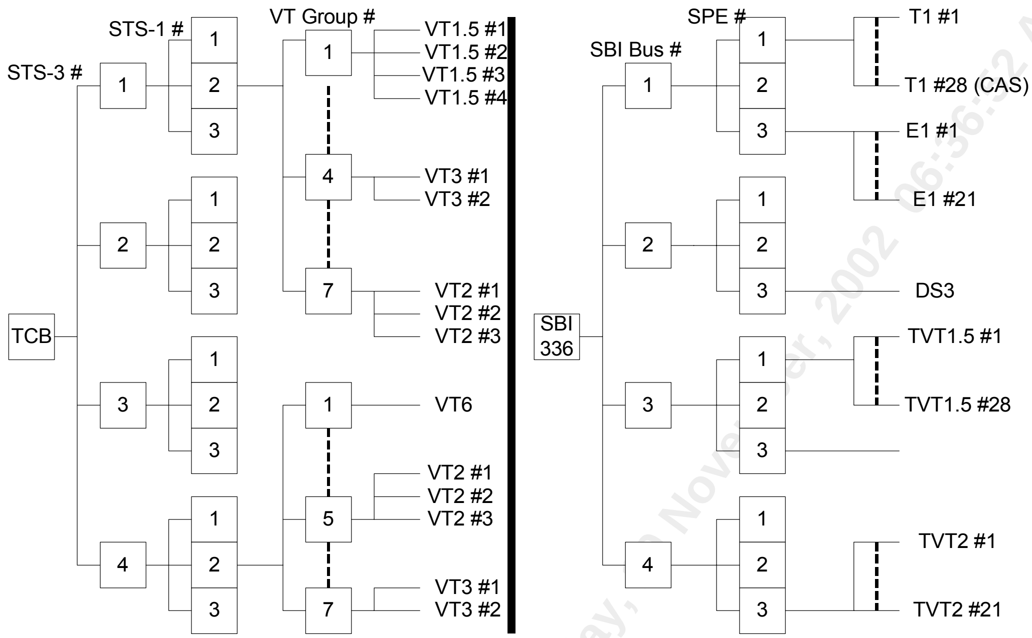
The API functions for status retrieval and PRGM refer to the STS-1 path number. The "order of transmission" of the bytes in a SONET/SDH frame is followed and Table 4 provides the translation between the tributary numbering and the STS-1 path number. It can also be interpreted as how the columns from different STS-1 data streams are interleaved with each other. In other words, the column of (1,1) is followed by that of (2,1), then the column of (3,1), and etc.

Table 4: TeleCombus and SBI336 Bus STS-1 Path Numbering

TeleCombus (STS3 num, STS1 num) SBI336 Bus (SBI num, SPE num)	STS-1 Path number

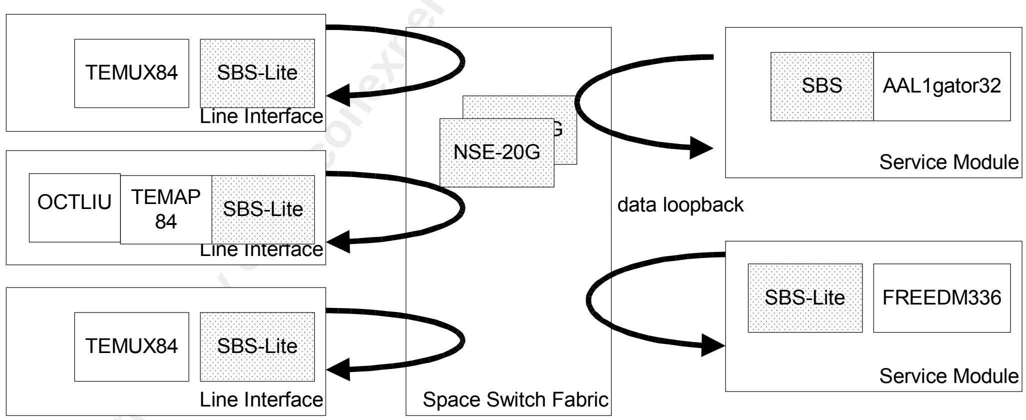
TeleCombus (STS3 num, STS1 num) SBI336 Bus (SBI num, SPE num)	STS-1 Path number
(1,1)	1
(2,1)	2
(3,1)	3
(4,1)	4
(1,2)	5
(2,2)	6
(3,2)	7
(4,2)	8
(1,3)	9
(2,3)	10
(3,3)	11
(4,3)	12

Figure 4: NSE/SBS Tributary Naming Convention for TeleCombus and SBI336 Bus



Chipset Loopback State

Figure 5: NSE/SBS Switching Fabric – Loopback State



The loopback state (Figure 5) can be viewed as the "point of reference" for the state of the switching fabric. The CSD initializes the system to this state by default at the beginning. An API is also available to upper layer application to aid in bringing the fabric to this loopback state at any time. Doing so wipes out all the existing connections.

Fabric Wiring Topology

The CSD is designed to take into account arbitrary wiring topologies between SBS devices and the NSE core (Figure 6) in the system. However, designer should try to use the default, or standard SBS/NSE wiring in the system. Non-standard wiring topologies increase the processing overhead of the CSD.

In the case of a 3-stage fabric, the physical wiring between all NSE devices cannot be arbitrary and must follow the “PMC-standard” NSE/NSE wiring scheme. This wiring standard is outlined as follows: For the NSE switching core, the interconnection between all NSE devices is defined below. The mapping specifies the core (stage 1) NSE connection associated with a given edge (stage 0 or stage 2) NSE connection.

depth(core) = NSEport(edge) / portsPerNSE

NSEport(core) = depth(edge) * portsPerNSE + NSEport(edge) / portsPerNSE

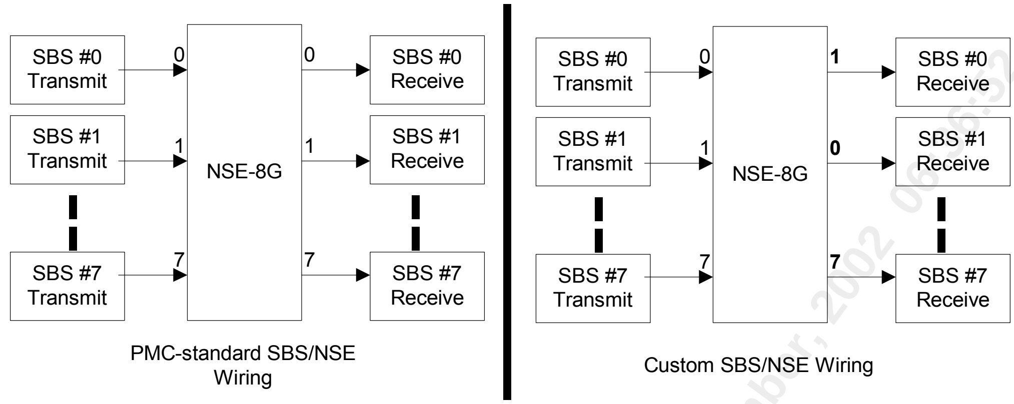
where portsPerNSE = NSEports / Depth. All divisions are integer division.

For the SBS-to-NSE connections, a SBS device must connect to the same port number of the two NSE devices that are at the same depth.

For instance, if it is a 3-stage fabric made up of NSE-20Gs with depth 2, here is the “standard" connection. NSE(x,y) specifies the NSE devices in the fabric with x denotes the stage and y denotes the depth. SBS(z) specifies the SBS devices attached to the 3-stage NSE core.

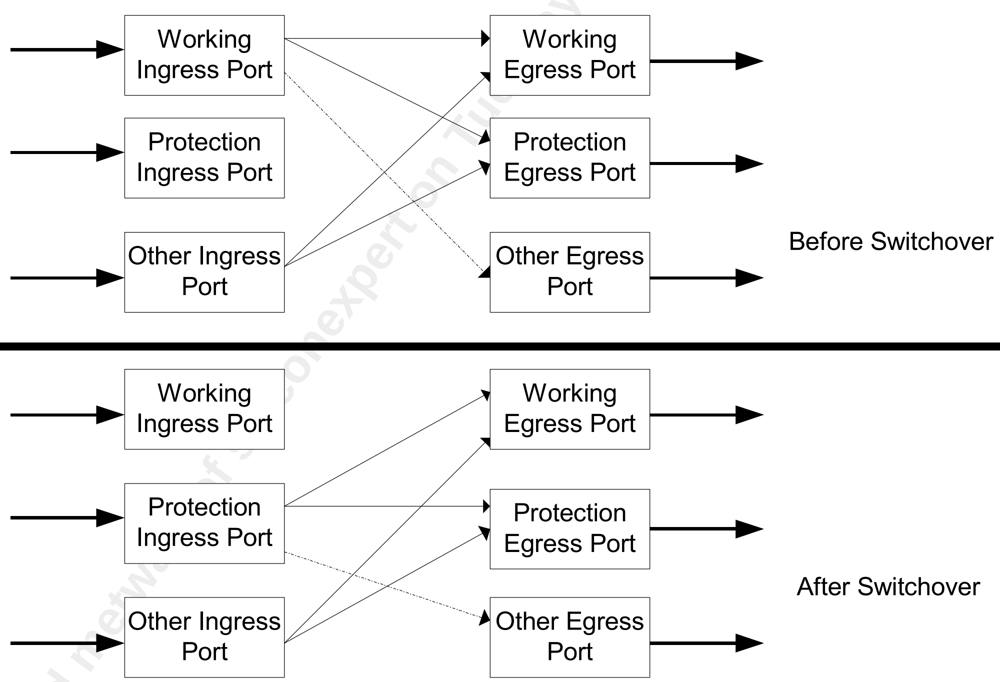
NSE(1,0) incoming ports(0-15) connects to NSE(0,0) outgoing ports(0-15)
NSE(1,0) incoming ports(16-31) connects to NSE(0,1) outgoing ports(0-15)
NSE(1,0) outgoing ports(0-15) connects to NSE(2,0) incoming ports(0-15)
NSE(1,0) outgoing ports(16-31) connects to NSE(2,1) incoming ports(0-15)
NSE(1,1) incoming ports(0-15) connects to NSE(0,0) outgoing ports(16-31)
NSE(1,1) incoming ports(16-31) connects to NSE(0,1) outgoing ports(16-31)
NSE(1,1) outgoing ports(0-15) connects to NSE(2,0) incoming ports(16-31)
NSE(1,1) outgoing ports(16-31) connects to NSE(2,1) incoming ports(16-31)
SBS(0) transmit LVDS link connects to NSE(1,0) port 5
SBS(0) receive LVDS link connects to NSE(3,0) port 5
SBS(1) transmit LVDS link connects to NSE(1,1) port 20
SBS(1) receive LVDS link connects to NSE(3,1) port 20

Figure 6: NSE/SBS Switching Fabric – Default vs. Custom LVDS Wiring



1+1 and 1:N Port Protection

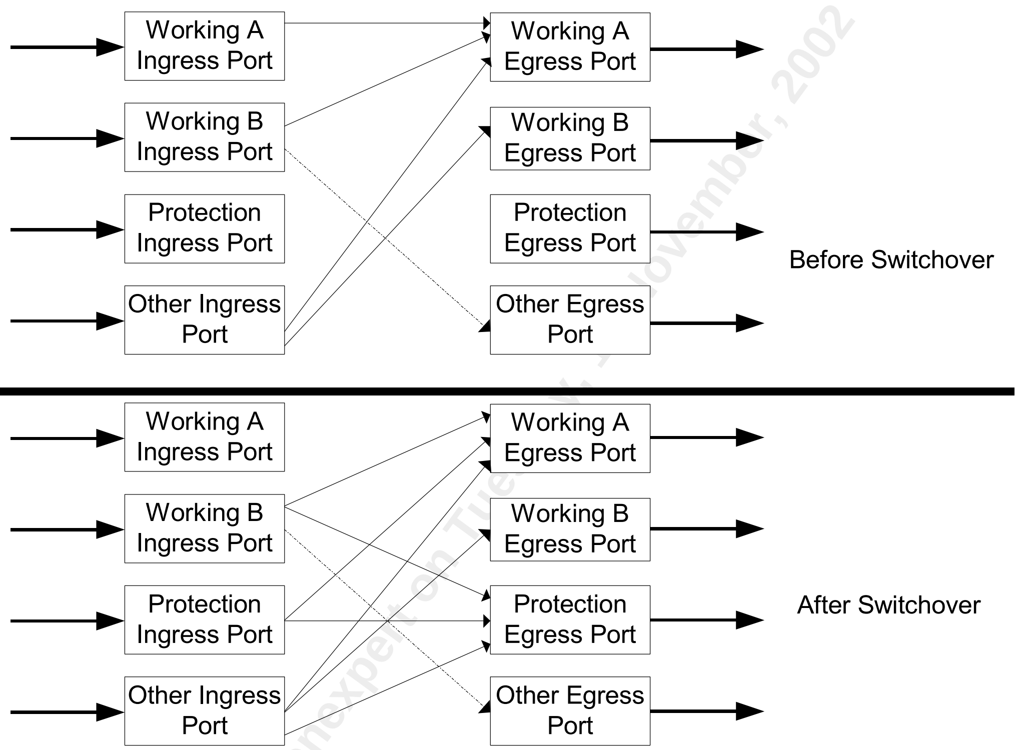
Figure 7: 1+1 Port Protection Before and After a Switchover



The chipset driver supports 1+1 and 1:N port protection. A 1+1 port protection (Figure 7) involves a pair of logical ports. One port is labeled as the working and the other as protection. These labels are arbitrary and active traffic could pass through either port after a switchover. Traffic is multicast to both the working and protection egress ports at all times. Functions are available to group/ungroup logical ports and perform switchovers for the 1+1 port protection.

For 1:N port protection (Figure 8), one logical port is reserved to protect N working ports. Functions are available to reserve a logical port for protection purpose for a group of working ports. At a switch over, external equipment is responsible for redirecting traffic from one of the working ingress port to the protection port.

Figure 8: 1:N Port Protection (with N = 2) : Before and After a Switchover to Protect Working A Ingress Port



Unidirectional Path Switching Ring (UPSR)

UPSR (Unidirectional Path Switching Ring) is supported by the chipset driver. Traffic can be added to or dropped from a UPSR. Figure 9 illustrates a typical UPSR operation. When adding traffic to a UPSR, the traffic is added to both the outer and inner loops (timeslots do not have to be the same in both loops). In the case of dropping traffic from the UPSR, traffic is either sourced from the outer or inner loop depending upon the state of the switchover. Note that the two SBS devices associated with a UPSR have to be declared as a UPSR protection pair prior to the add/drop operation.

Traffic can also be added to a UPSR without protection, i.e., traffic is only added to either the outer or inner loop in a user-specified timeslot without duplicating the same traffic on the other loop. Likewise, unprotected traffic can also be dropped from either loop. A path level switchover is not applicable in this case.

Figure 9: UPSR Protection - 2-fiber

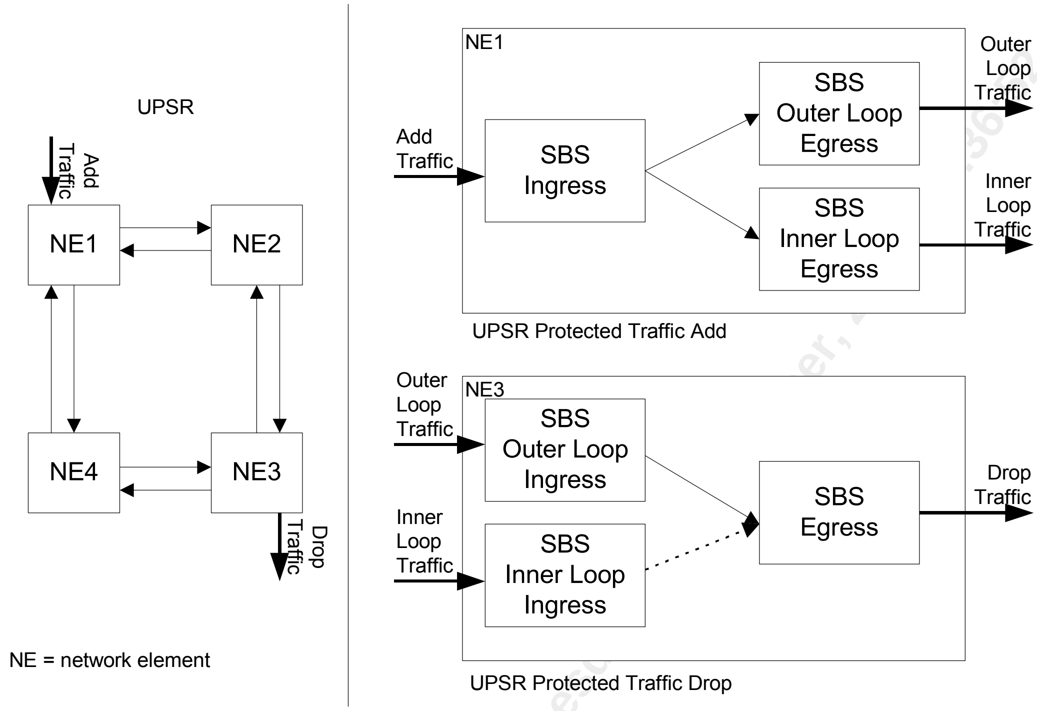
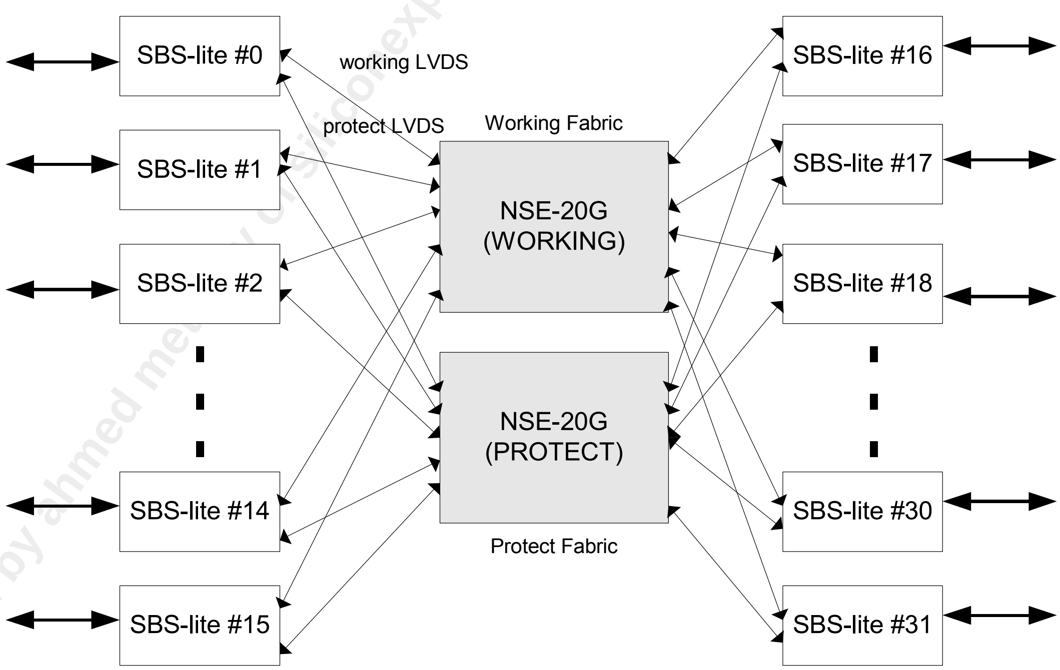


Figure 10: Stage-1 Narrowband Switch Fabric: 32 SBSLITEs, 1 NSE-20G for Working Fabric and 1 for Protect Fabric



Working and Protection Fabric

Figure 10 shows a working NSE-20G fabric with 32 SBSLITEs protected by another NSE-20G device. The traffic can be selected to be using the working or protect LVDS links on the SBS/SBSLITE devices. This is to protect against any hardware failure that may occur in the fabric card. The switching over to the protect fabric card is easily achieved by one of the following options: (a) software control, or (b) hardware control via the RWSEL pin. In either case, user is responsible for synchronizing the connection pages between the working and the protect fabric cards.

Standard and Doubled Fabric

Our discussion has so far been limited to standard fabric only. A limitation with the standard fabric is its ability to handle multicasting without blocking. This limitation can be partially (guaranteed non-blocking for 2:1 casts) alleviated by introducing more SBS or NSE devices to the fabric. This is sometimes referred to as fabric “speed-up”.

Figure 11 shows a doubled SBS fabric. In this configuration, SBS devices are doubled up to provide twice the number of timeslots available in a standard fabric. Each SBS pair has an external multiplexer attached on the egress side to control the traffic selection. Traffic is either selected from the “A” device or the “B” device at anytime depending on the control signal of the multiplexer. The control signal is one of the TeleCombus signals from the “A” SBS device. Depending on the path termination mode (MST, LPT, or HPT) employed by the system, different TeleCombus signal is used for the multiplexer control. If the system is configured to run in either MST or HPT mode, the OPL signal should be employed; otherwise, the OTAIS signal should be used. User determines which control signal, OPL or OTAIS, when initializing the CSD via the MIV. Please note that the OPL signal is shown in the figure. For a more in depth discussion on doubled fabric, please refer to the “CHESS-NB Designing a Non-blocking Fabric for 1:2 Multicast (PMC-2020050).

Figure 12 shows a doubled SBS and NSE fabric. It is designed to circumvent the reduced bandwidth supported by a doubled SBS fabric. Since the SBS devices are doubled up in a doubled SBS fabric, the overall aggregate bandwidth is halved. The doubled SBS and NSE fabric reclaims the lost bandwidth in a doubled SBS fabric.

Figure 11: Doubled SBS Fabric – 10Gbps Aggregate Bandwidth using NSE-20G

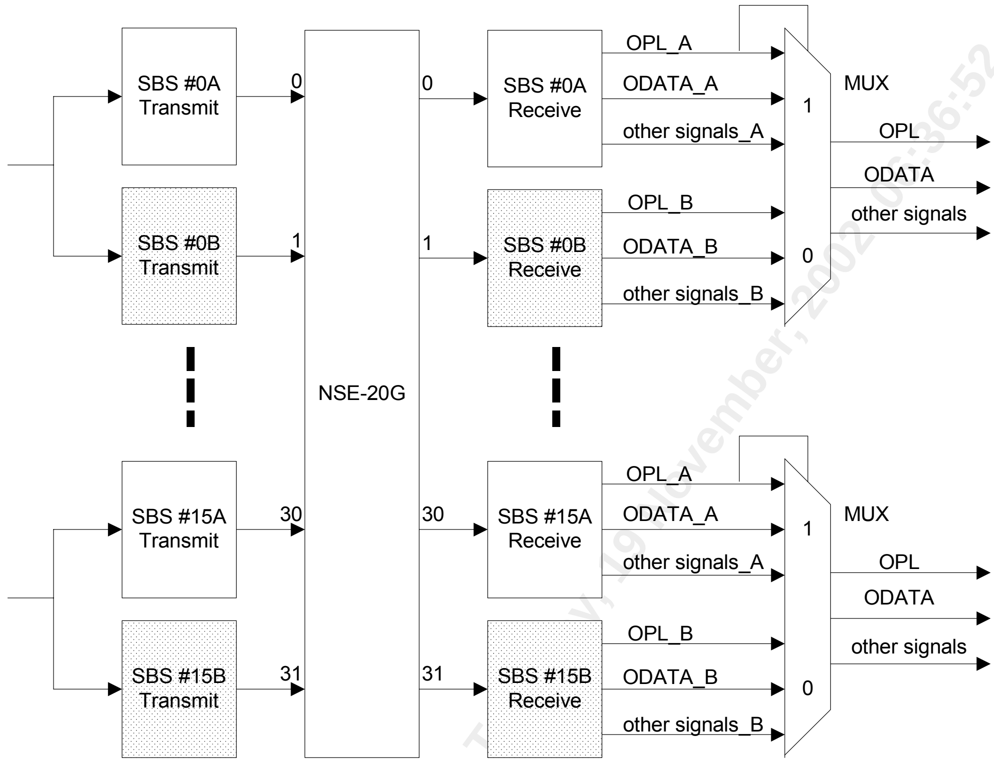
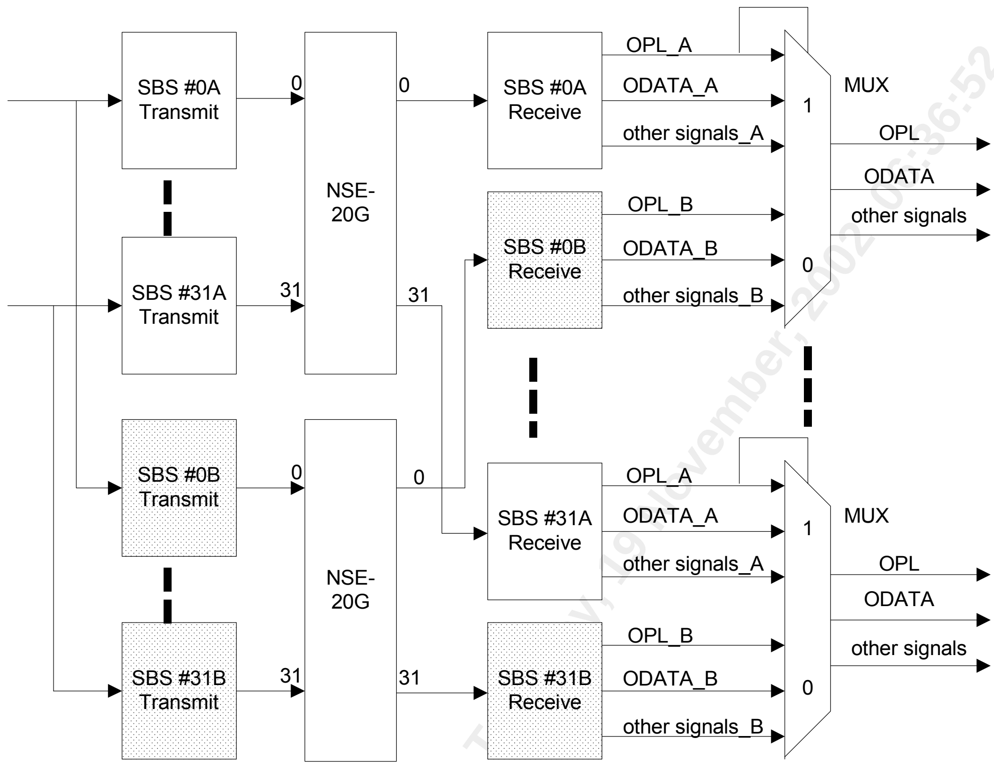


Figure 12: Doubled SBS and NSE Fabric – 20Gbps Aggregate Bandwidth using Two NSE-20G



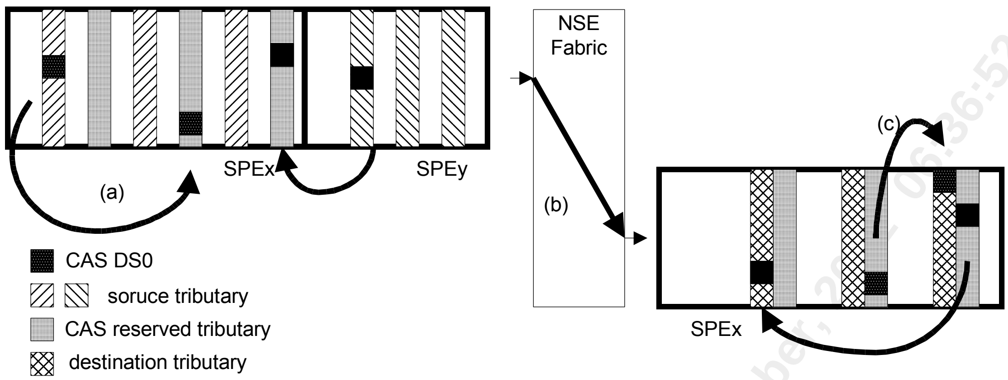
CAS Traffic Routing

When routing CAS traffic through the fabric, the DS0s has to be first packed into a tributary with CAS processing enabled, (The assumption is the DS0s fills a T1/E1 virtual tributary. Multiple tributaries are required if there are more DS0s than a tributary can hold) and then switched as a whole across the NSE space switch fabric to the destination SBS. This is deemed necessary because only SBS has the ability to process CAS. The concept of a CAS route across the fabric is introduced as a result when routing CAS traffic across the fabric.

A CAS route can be viewed as a channel setup for CAS traffic between the two SBSs. All CAS DS0 traffic is first routed to the CAS reserved tributaries (Figure 13) before going through the NSE space switch to the same CAS reserved tributaries in the output SBS. The traffic is then routed to the destination tributary from the reserved tributary on the output SBS side.

The CSD automatically handle all aspects of CAS traffic routing and is transparent to the user. Currently, the CSD does not support a mixture of T1 and E1 CAS traffic in the fabric.

Figure 13: CAS Traffic Routing Across NSE/SBS Fabric



In-band Link Communication

In-band links are dedicated point-to-point communication links over LVDS. This in-band link provides a clear channel for communication between devices located remotely from each other. It is in place to facilitate shelf-to-shelf or rack-to-rack intercommunication. Sent at every frame (125us), each message is 36 bytes long, with 2 bytes of header, 32 bytes for payload, and 2 bytes for the CRC-16 trailer. The header bytes provide some near-realtime control signals between devices to synchronize page switching, indicate switchover between working or protected links and exchange three user defined signals (hardware) and 8 Auxiliary signals (software). The User and Auxiliary signals can be used to indicate interrupts or can be used for handshaking between the end point microprocessors. The CSD provides API for sending, receiving messages, and manipulating the header bytes via the in-band links.

SBS Egress Bus Integrity

Egress bus integrity on SBS has to be preserved at all time or the operation of downstream devices can be adversely affected. Egress bus integrity includes the proper setup of all the transport overhead (TOH), J1, stuff, V1/2/3/4/5 and the payload columns/bytes in the frame that are vital to the well being of the bus signals. For example, improper bus signals can be generated as a result of a misplaced J1 byte in the outgoing bus. Leveraging the MSU programming capability on the egress side (applicable only to SBS revision B or later devices), the CSD automatically handles all the bus integrity issues by fixing up all the bus-related columns/bytes. This is done when virtual tributaries (VTs) or DS0s are setup using `nbcsFmgtMapTrib` or `nbcsFmgtMapDs0`. It is noteworthy to point out that, once the payload types are defined, these columns/bytes can be setup in advance before any VT/DS0 connections are setup. The advantage is to reduce the connection time because these special columns/bytes are only setup once. Aside from disconnecting a valid circuit, API `nbcsFmgtUnMapTrib` and `nbcsFmgtUnMapDs0` can be used to setup these special columns/bytes in advance. The connection map settings can then be retrieved/populated by `nbcsFmgtGetChgMap`. This unmapping is important to define the “unused” DS0s or tributaries so that they have proper egress bus signal for the downstream device. Extreme care should be taken on handling the unused DS0s (inside a tributary) or tributaries (inside a SPE). Without unmapping the “unused” DS0s/tributaries, they may inadvertently draw input from undesirable input timeslots and affect the egress bus signal.

3 SOFTWARE ARCHITECTURE

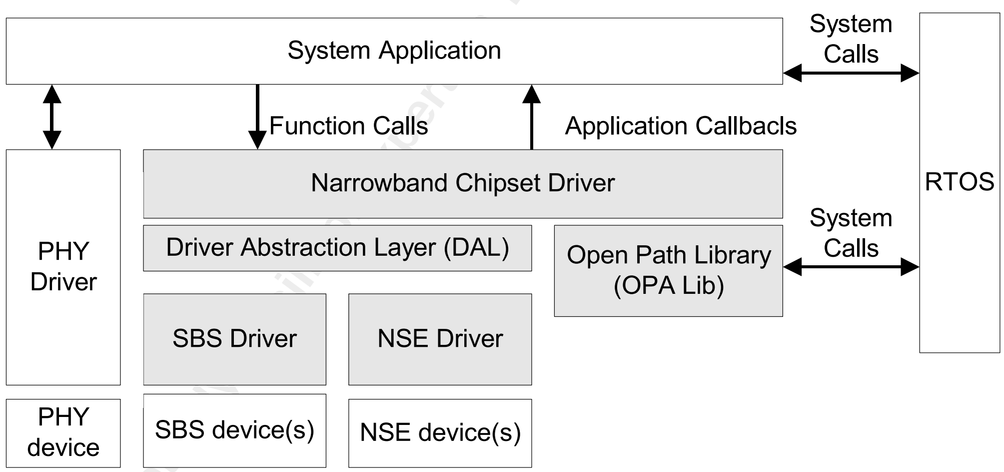
This section of the manual describes the software architecture of the Narrowband Chipset device driver. It includes a discussion of the driver's external interfaces and its main components.

3.1 Driver External Interfaces

Figure 14, Figure 15, and Figure 16 illustrate the external interfaces defined for the Narrowband Chipset device driver in different system configurations. The CSD can be initialized to work with centralized, distributed or various other system configurations.

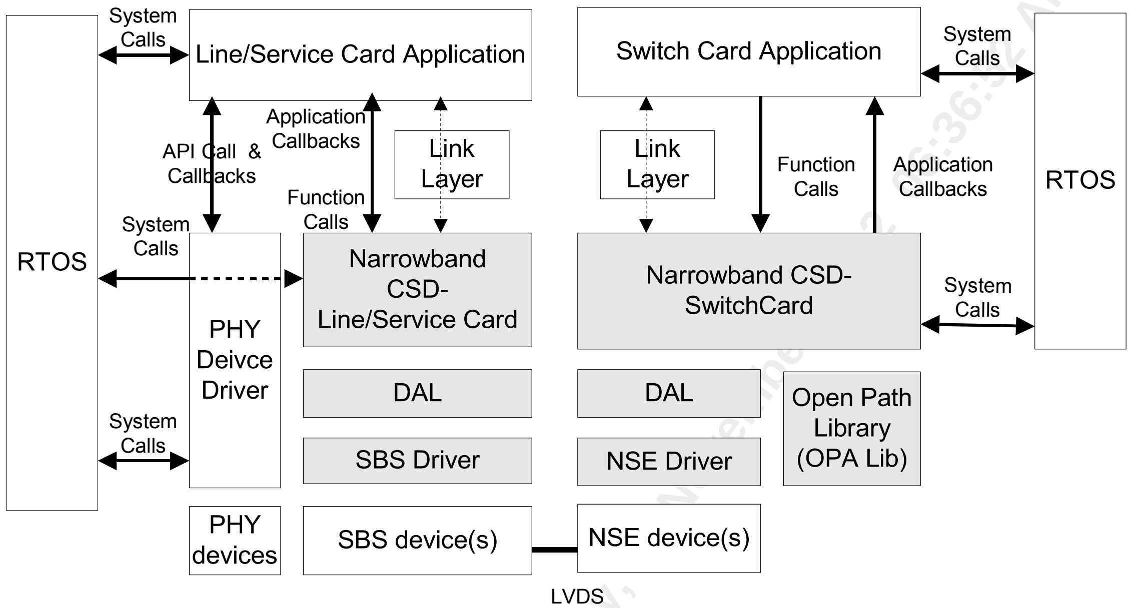
The interface between the CSD and the upper layer application is consistent regardless of configuration. This is an attempt to present a unified interface to the upper layer regardless of whether it is configured to run in a line card or a switch card. There are, of course, some API functions and callbacks that are not available on the switch card or line card side when the functionality does not belong. For instance, the switch card application cannot access PRGM functionality that is provided by SBS devices. Below is a description of how the CSD adapts to some typical system configurations

Figure 14: Driver External Interfaces – Typical Centralized Configuration



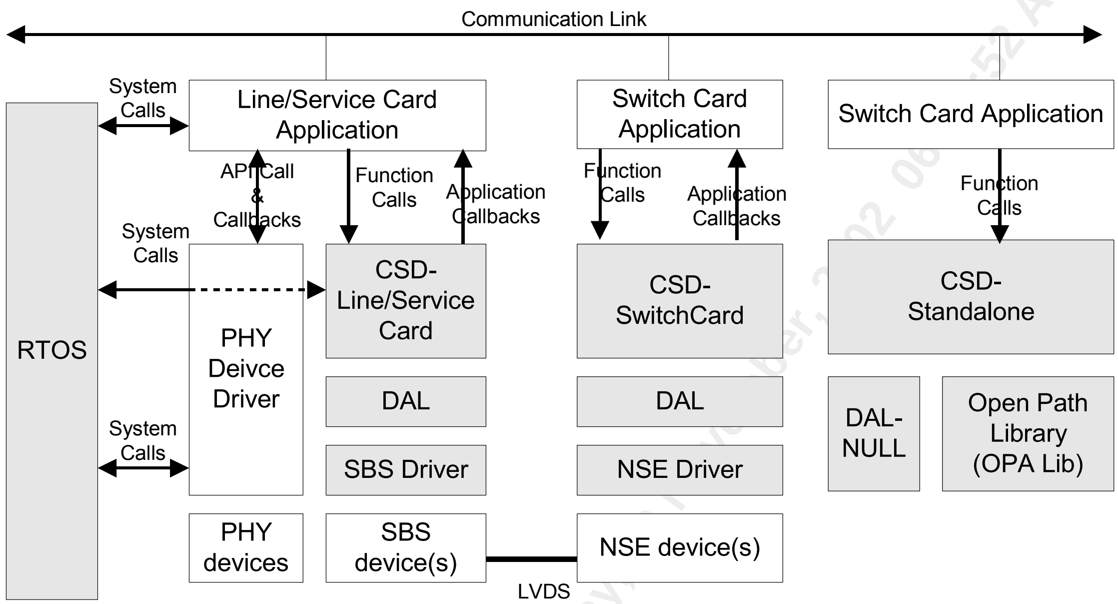
In a typical centralized configuration, all devices are under the control of a single microprocessor. It is the most straightforward configuration since all devices, SBS, NSE, and other supporting devices, are assumed to be under the control of a single microprocessor. The CSD can easily be configured to accommodate such configuration.

Figure 15: Driver External Interfaces – Typical Distributed Configurations



In a typical distributed configuration, multiple instances of the chipset driver can be deployed to run under multiple microprocessors. Each CSD instance is most likely configured differently to accommodate different system configurations. For instance, a typical line card may consist of some PHY layer devices and one or more SBS devices, the CSD will then be configured to run just the SBS device driver and all the physical SBS devices attached to the line card. On the contrary, a typical switch card that consists of NSE devices requires the CSD to run with the NSE device driver. All the NSE devices will be registered (added) as physical devices that are attached to the card. Figure 16 shows another variation in a distributed system model with the CSD/OPA physically detached from any physical SBS and NSE devices. An external link (e.g., Ethernet) is required to act as the communication channel between all boards.

Figure 16: Driver External Interfaces – Typical Distributed Configurations (Standalone OPA)



The flexibility of the CSD lends itself to easy adaptation to all of the different system configurations shown (or any other custom system configurations). The most important concept is that there should only be one active OPA in the working system of a distributed system. In other words, only one CSD instance should be configured to run the OPA. The OPA schedules calls in the fabric and keeps the record of all existing calls. It therefore acts as the central repository for all the records. Since this CSD/OPA has to be aware of all devices in the fabric, devices that are not local to this CSD/OPA must be added as "logical" devices (The status of the device is declared when adding the device via `nbcsAdd`). This allows the CSD/OPA to properly recognize the existence of the SBS or NSE devices in the fabric, though not directly under its control. (In the case of a 1+1 redundant fabric system, a second (standby) copy of the OPA is allowed to run in parallel with the working copy.)

Application Programming Interface

The driver's Application Programming Interface (API) is a list of high-level functions that can be invoked by application programmers to configure, control, and monitor Narrowband Chipset devices. The API includes functions that:

- Manage the chipset devices
- Perform diagnostic tests
- Perform run-time system diagnostics with PRBS generators and monitors
- Configure and control system interface/clock
- Retrieve status and counts information

- Control LVDS links operation
- Manage the switching fabric
- Configure the map setting in the space and time switches
- Configure and access the in-band link communication channels

The chipset driver's API functions use the services of the other driver components to provide this system-level functionality to the application programmer.

The chipset driver's API also consists of callback routines that are used to notify the application of significant events that take place within the chipset device(s) and module.

Real-Time OS (RTOS) Interface

The chipset driver's Real-Time Operating System (RTOS) interface provides functions that let the chipset driver use the RTOS's memory, interrupt, and pre-emption services. These RTOS interface functions perform the following tasks for the chipset driver:

- Allocate and de-allocate memory
- Manage buffers for the ISR and the DPR
- Take and give semaphores
- Enable and disable pre-emption

The RTOS interface also includes service callbacks. These are functions installed by the driver using RTOS service calls such as installing interrupts. These service callbacks are invoked when an interrupt occurs.

Driver Abstraction Layer (DAL)

The driver abstraction layer provides abstraction to the underlying device drivers. The interface of this layer models after the functionality of a generic time stage switch and a space stage switch. This layer isolates the CSD from the device drivers and lends itself to easy porting of the CSD to various hardware configurations or even to new devices that provide similar time:space:time switching capabilities. Please refer to Appendix for more details regarding the DAL.

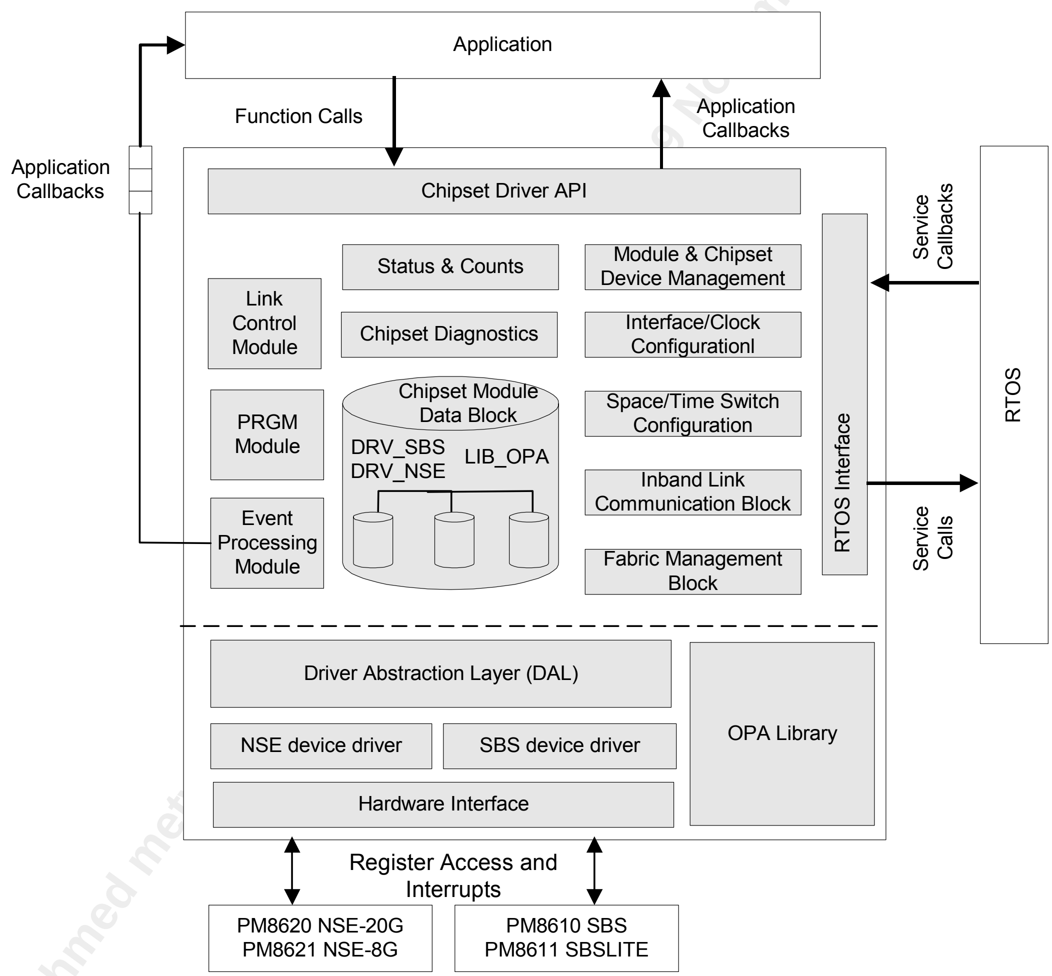
3.2 Main Components

Figure 17 illustrates the top-level architectural components of the Narrowband Chipset device driver:

- Module data-block
- Module and chipset device management
- Interface/clock configuration
- Event processing module

- Status and counts
- In-band link communication module
- Fabric management
- Space/time switch configuration
- Link control
- PRBS generator and monitor (PRGM)
- Device diagnostics

Figure 17: Driver Architecture – Internal Components



Chipset Module Data-Block

The Chipset Module Data-Block (CSMDB) is the top-layer data structure created by the Narrowband Chipset driver. The CSMDB stores context information about the driver module, such as:

- Module state
- Maximum number of devices
- The NSE software module
- The SBS software module

The NSE and SBS software modules manage the underlying NSE and SBS device driver. Each module keeps track of the maximum number of devices allowed, the current count of registered devices, etc.…

Module and Chipset Device Management

The module and chipset device management block performs the following:

- Module management services, such as initializing the driver and then allocating memory and other RTOS resources that are needed by the driver
- Chipset device management services, such as providing basic read/write routines and initializing a device in a specific configuration, as well as enabling the device's general activity

For more information about the module and device states, see the state diagram on page 48. For typical module and device management flow diagrams, see pages 50 and 51 respectively.

Event Processing

Event processing is closely associated with application callbacks (which is a mechanism employed to notify the upper layer application when an event occurs). A set of events is defined with callback ability. The user can choose to enable/disable a particular event and no callback are issued if that event is disabled. Events mostly originate from the underlying device drivers.

Two modes are supported, namely interrupt-driven or polling. When the driver is in interrupt mode, registered callbacks are issued to the application. If the driver is in polling mode, application has to call a function to periodically check the occurrence of events and the issue of callbacks.

Status and Counts

User calls `nbcsStatsGetCounts` to retrieve counts for the specified device or group. It is the responsibility of the user to invoke the count routines often enough to avoid counter rollover or saturation. It is up to the application code to derive time-based calculations such as errored seconds.

The status routine, nbcsStatsGetStatus, is responsible for retrieving the status information from underlying devices such as clock monitoring.

Interface/Clock Configuration

The chipset works with either SBI/SBI336 or TeleCombus devices. The function, nbcsIntfCfgBus, configures the SBS devices in the chipset driver for either bus system. The SBS devices can be configured to handle 4xSBI bus @ 19.44 MHz or 1xSBI336 @ 77.76 MHz. In addition, parallel 77.76 MHz SBI bus output on the transmit side can be selected. Doing so disables all the LVDS serial output. (Note: SBSLITE does not support the 4xSBI mode nor the parallel bus output mode). Other parameters that can be configured include bus parity, even or odd.

For TeleCombus, the configuration is similar. User has a choice of either 4x19.44 MHz TeleCombus or 1 x 77.76 MHz TeleCombus. For additional bus parameters, user can configure bus parity, J1 byte position, H1 and H2 pointer value, etc… Function nbcsIntfCfgPyld, configures the payload type of the SBI/TeleCombus once the bus type is defined. For SBI bus, this function configures what type of traffic the SPE carries. It can be T1, E1 or DS3/E3. For TeleCombus, this function configures the VT types, VT1.5, VT2, VT3, or VT6 being carried in the SPE. If the system operates in SBI bus mode, the function, nbcsIntfCfgTrib, further configures the attributes of the tributaries. For each tributary, user can enable the output on the bus (applicable only in outgoing but not incoming bus), enable the CAS processing, enable the justification request, and defines the tributary as a transparent virtual one (TVT).

All the CSUs (clock synthesis units) and DLLs are accessible via nbcsIntfCfgCsu provided by this logical block. User can reset or put any units in the chipset to low power mode.

The C1 frame pulse delay of a given device/group can be programmed by calling API nbcsIntfCfgC1FrmDly.

LVDS Serial Link Control

This block provides functions nbcsLkcInsertLcv, nbcsLkcInsertTp, nbcsLkcForceOfa, nbcsLkcForceOca, and nbcsLkcCenterFifo for forcing line code violation, inserting test pattern, out-of-frame and character alignment, and centering FIFOs respectively. In addition, serial links in NSE can be put into low power mode when unused. Invoking API function nbcsLkcCntl for SBS devices, user can select the active link between the working and the protection link if the software link control option is enabled. This parameter, along with J0 byte insertion, and termination mode are all accessible from this block using nbcsLkcCfg.

Space/Time Switch Configuration

This logical block exposes two functions, nbcsStswMapSlot and nbcsStswGetSrcSlot to write and read the connection maps of the underlying switches, time or space, directly. It is necessary to access the individual switch when the user has to set up new calls across the fabric. The sequence is usually to make request for new connections, retrieves all the new settings, and then configures the switch(es) with the new settings. These functions can also be used to set up the switching fabric directly, bypassing the OPA all together.

In addition, the driver provides the API nbcsStswSetPage to switch page(s) of a single device or a group of devices in the system. The API function nbcsStswGetPage retrieves the active page of a device. By utilizing the in-band link PAGE bits, both API functions can be invoked even for a remote SBS device (from the switch card side) in a distributed configuration.

The API function nbcsStswTogglePage is designed to perform a system-wide page switching for all registered NSE/SBS devices. This operation is pointless when not synchronized with the incoming C1 frame pulse. As a result, this function utilizes the underlying C1 frame pulse interrupt to coordinate the page switching in the system to guarantee a hitless page switching operation. The in-band link PAGE bits are used to switch SBS pages remotely; hence, user should enable the in-band link page switching operation in all the remote SBSs.

API nbcsStswCopyPage copies the connection map contents from active to inactive page within the same switch or from inactive to inactive page across different device of the same type.

Fabric Management Module

The fabric management module provides services for call management that includes call setup, teardown, fabric setting retrieval, 1+1, 1:N port protection, UPSR protection and etc. This block interacts with the OPA library, which is responsible for calculating the new fabric setting for new connections, and providing 1+1 and 1:N port protection services.

The module accepts call setup requests in standard formats, STS-3/3c (SDH AU-4), STS-1, T1/E1, VTs, DS3/E3, fractional T1/E1s, and DS0s. For T1/E1 VTs, DS3/E3, STS-3/3c or fractional rate tributaries, nbcsFmgtMapTrib and nbcsFmgtUnMapTrib are used to setup and tear down connections. For DS0 connections, nbcsFmgtMapDS0 and nbcsFmgtUnMapDS0 are used instead. Routing CAS traffic is achieved by the same nbcsFmgtMapDS0 with the CAS flag set to logic one. User subsequently calls nbcsFmgtUnMapDS0 to tear down CAS DS0 or non-CAS DS0 connections. The function nbcsFmgtRsvpCasRoute reserves the total number of virtual tributaries set aside for CAS routing.

Arbitrary SBS wiring is supported for the fabric. User, by calling nbcsFmgtDefWiring, provides a wiring table describing the underlying physical wiring between all SBS devices and the NSE core. The wiring has to be properly defined before other operation can be carried out.

Calling nbcsFmgtSetProtect sets up 1+1 and 1:N port protection. The parameters for this function include the type of port protection to set up; and all the ports (working and protection) involved. Port protection is removed by calling nbcsFmgtClearProtect. Upper layer application calls nbcsFmgtSwitchProtect to initiate the switching over from working to protection ports and vice versa. Depending on whether auto setting update is activated, the new setting will or will not be populated to the (inactive) connection page. In a distributed system model, user has to retrieve raw device settings by calling nbcsFmgtGetChgMap for distribution to the proper devices.

In many cases, raw device settings have to be retrieved from the OPA for distributing to the devices. There are several scenarios that require a change in device setting. The most common ones are new call connection request, and port protection switchover. The incremental setting change is required and can be retrieved by nbcsFmgtGetChgMap. A snapshot of the entire fabric can also be retrieved by nbcsFmgtGetMap.

Central to the page switching operation which is required any time new settings are to be activated, C1 frame pulse detection is provided by calling nbcsEventDetectC1FP which enables the underlying C1 frame pulse interrupt. Upper layer application should then listen to the callback function cbackC1FP to handle the C1 frame pulse reception. The most stringent requirement is in the case of TeleCombus, where this page swapping operation has to be completed 27 microseconds after the arrival of the C1 frame pulse. The in-band link PAGE[1:0] header bits are expected to be used in a distributed system environment.

The fabric can be brought to the initial state (loopback state) by calling nbcsFmgtSetLpbkMode. This resets the context of the OPA and essentially wipes out all current connections.

In-band Link Communication Module

In-band links are dedicated point-to-point communication links over LVDS. It is useful for communication between remote SBSs residing in line cards and NSEs that are populated in a core-switching card. The chipset driver allows the user to send and receive messages via the in-band links. Functions are available to send/receive in-band link messages, and configure/retrieve in-band communication parameters such as FIFO thresholds, timeouts, and etc.

Function nbcsIlcCntl enables/disables the in-band link controller associated with the specified LVDS link. When disabled, the in-band link controller is put in a “bypass” mode, no messages are written or inserted.

The in-band link has a receive FIFO depth of 8 messages. When the number of messages reaches the capacity, the chipset driver notifies (via callback if enabled) the application the condition requesting a readout from the FIFO. User can also set the threshold for messages in the FIFO ranging from 1 to 8 and is notified when the number of messages reaches that threshold. In addition, a timeout mechanism (with timeout constant 125, 250, 375, or 500 microseconds) in the FIFO is designed to notify user of any stale messages stored in the FIFO more than the specified timeout constant. User can call `nbcsIlcGetRxNumMsg` to query the total number of messages stored in the FIFO and then calls API `nbcsIlcGetRxMsg` to retrieve Rx FIFO messages. Each message is associated with a CRC error bit and a logic high for this status signals a CRC check failure for that message. For in-band link header bytes, user calls `nbcsIlcGetRxHdr` to retrieve all the header bytes USER[2:0], PAGE[1:0], LINK[1:0] and AUX[7:0]. It is noteworthy to point out that the CSD uses the PAGE[1:0] bits extensively to query and update the connection page of the remote SBS(s). User should refrain from using the PAGE bits. The rest of the header bytes can be used freely.

For the message transmission operation, user can retrieve the header bytes being sent by `nbcsIlcGetTxHdr` and alter the header bytes to be sent by `nbcsIlcSetTxHdr`. The Tx FIFO, similar to the receive one, also has a capacity of 8 messages. User can write multiple messages to the FIFO for transmission. API `nbcsIlcGetTxFifoLvl` is available to query the free capacity of the Tx FIFO for additional messages. User can then call `nbcsIlcTxMsg` to transmit the maximum number of messages admissible by the Tx FIFO.

PRGM Diagnostics

Pseudo-random bit sequence (PRBS) generator is provided at STS-1 granularity for all outgoing LVDS serial links for off-link verification. In addition, a PRBS monitor is provided at STS-1 granularity for all incoming LVDS serial links for off-link verification. This block is only applicable to SBS devices in the chipset. The `nbcsPrgmCfgPyld` API is available to configure the payload type. The `nbcsPrgmCfg` API is designed to configure the linear feedback shift register(LFSR), and the invert PRBS sequence mode or sequential mode on a per STS-1 basis and to enable/disable the PRBS generator and monitor on each STS-1 on the working and protect links in the SBS.

User can invoke `nbcsPrgmResync` and `nbcsPrgmForceErr` for PRBS monitor resynchronization and bit error insertion.

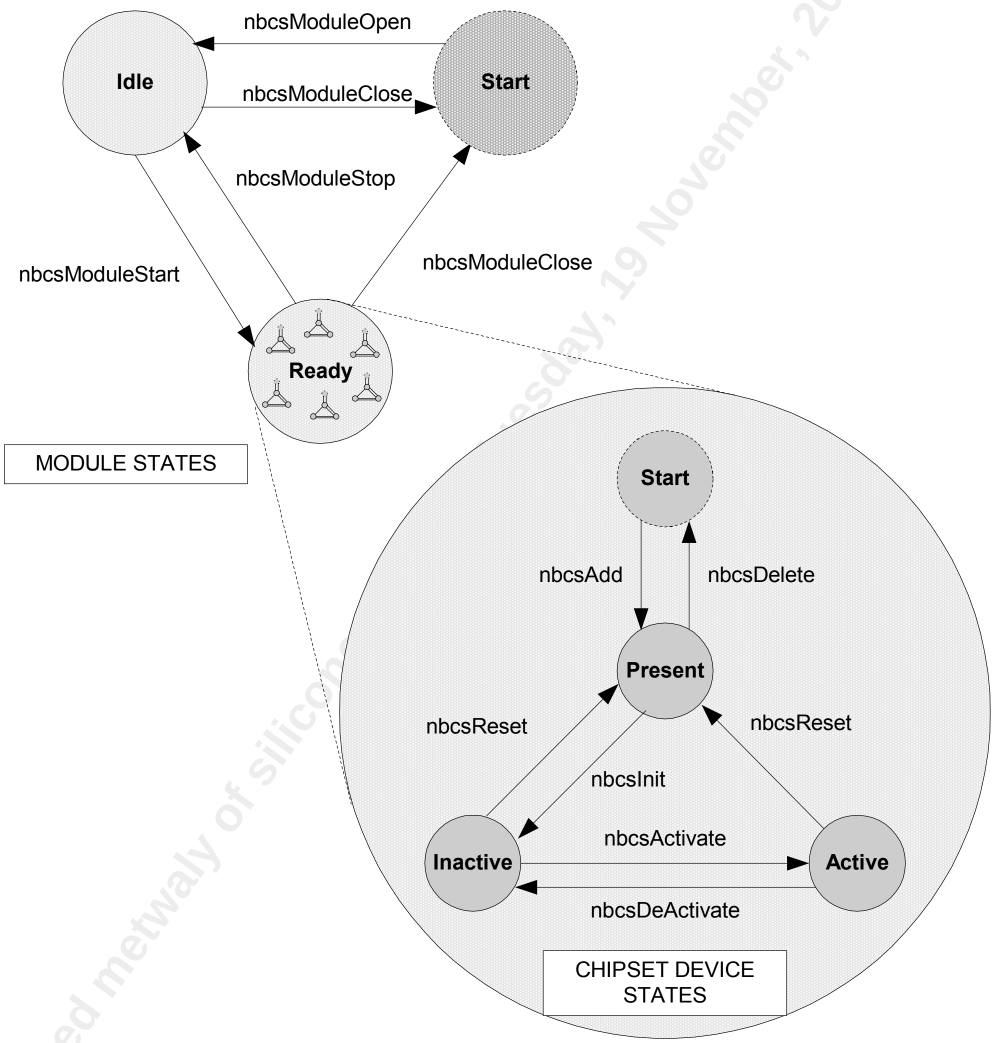
Chipset Device Diagnostics

The chipset device diagnostics API can be used to isolate/identify problems within a specified chipset device and its interfaces. The `nbcsDiagTestReg` and `nbcsDiagTestRam` API conduct the register and RAM tests for the chipset driver. User can call `nbcsDiagLpbk` for device loopback.

3.3 Software States

Figure 18 shows the software state diagram for the Narrowband Chipset driver. State transitions occur on the successful execution of the corresponding transition functions shown below. State information helps maintain the integrity of the CSMDB by controlling the set of operations allowed in each state.

Figure 18: Driver Software States



Module States

The following is a description of the Narrowband Chipset module states. Please see Section 5.1 for a detailed description of the API functions that are used to change the module state. The module states are:

Start

The chipset driver module has not been initialized. In this state the chipset driver does not hold any RTOS resources (e.g., memory and timers), has no running tasks, and performs no actions.

Idle

The chipset driver module has been initialized successfully. The Module Initialization Vector (MIV) has been validated; the CSMDB has been allocated and loaded with current data; the per-device data structures have been allocated; and the RTOS has responded without error to all the requests sent to it by the driver.

Ready

This is the normal operating state for the chipset driver module, which means that all RTOS resources have been allocated and that the chipset driver is ready for underlying devices to be added. The chipset driver module remains in this state while devices are in operation.

Chipset Group and Device States

The following is a description of the Narrowband Chipset device states. See section 5.1 for a detailed description of the API functions that are used to change the chipset device state.

Start

The chipset device has not been initialized. In this state the device is unknown to the driver and performs no actions. There is a separate flow for each device that can be added, and they all start here.

Present

The chipset device has been successfully added. All devices are detected and properly registered with essential information recorded in the chipset driver module. In this state, the device performs no actions.

Inactive

In this state the chipset device is configured; however, all data functions – including interrupts, status and counts functions – have been de-activated.

Active

This is the normal operating state for the chipset device. In this state, either interrupt servicing or polling is enabled.

Indeterminate (group state only)

A group is in this state if not all of the devices in the group are in a consistent state. While in this state, some API functions are still accessible, as described in later sections.

3.4 Operation Processing Flows

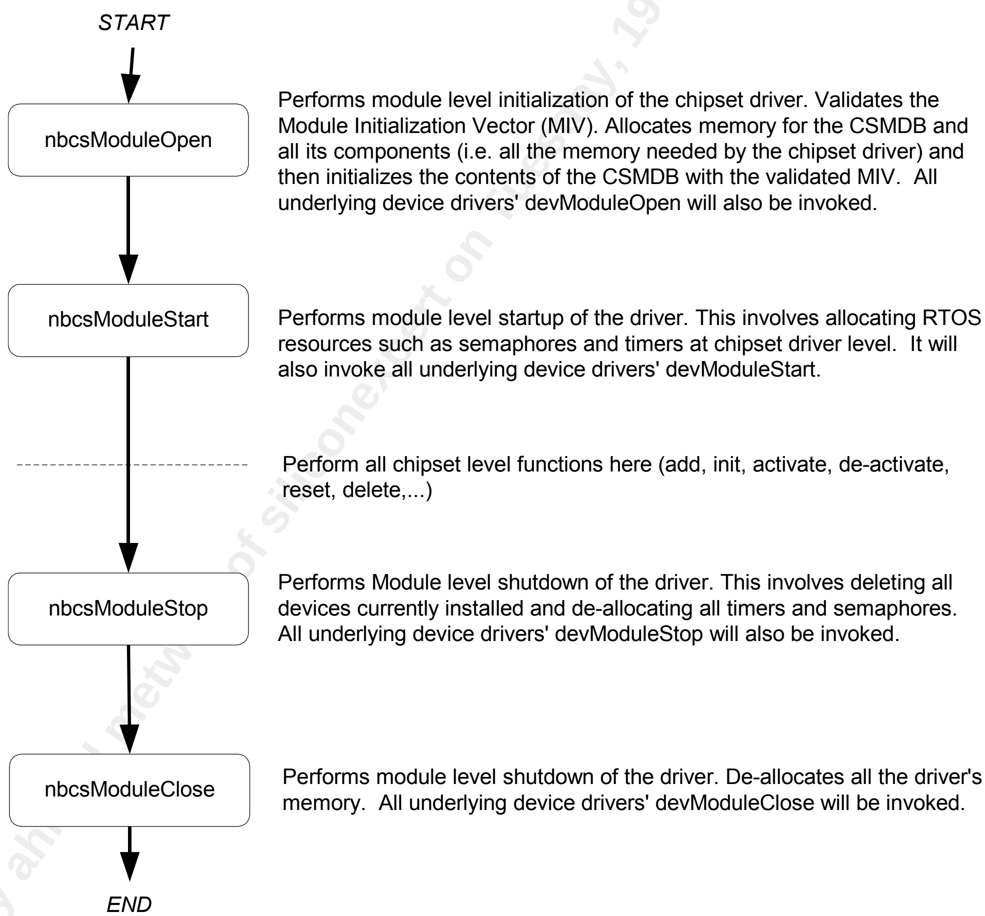
This section of the manual describes the main processing flows of the Narrowband Chipset driver components.

The flow diagrams presented here illustrate the sequence of operations that take place for different driver functions. In addition, the diagrams also serve as a guide to the application programmer by illustrating the sequence in which the application must invoke the driver API.

Module Management

The following diagram illustrates the typical function call sequences that occur when either initializing or shutting down the Narrowband Chipset driver module.

Figure 19: Module Management Flow Diagram

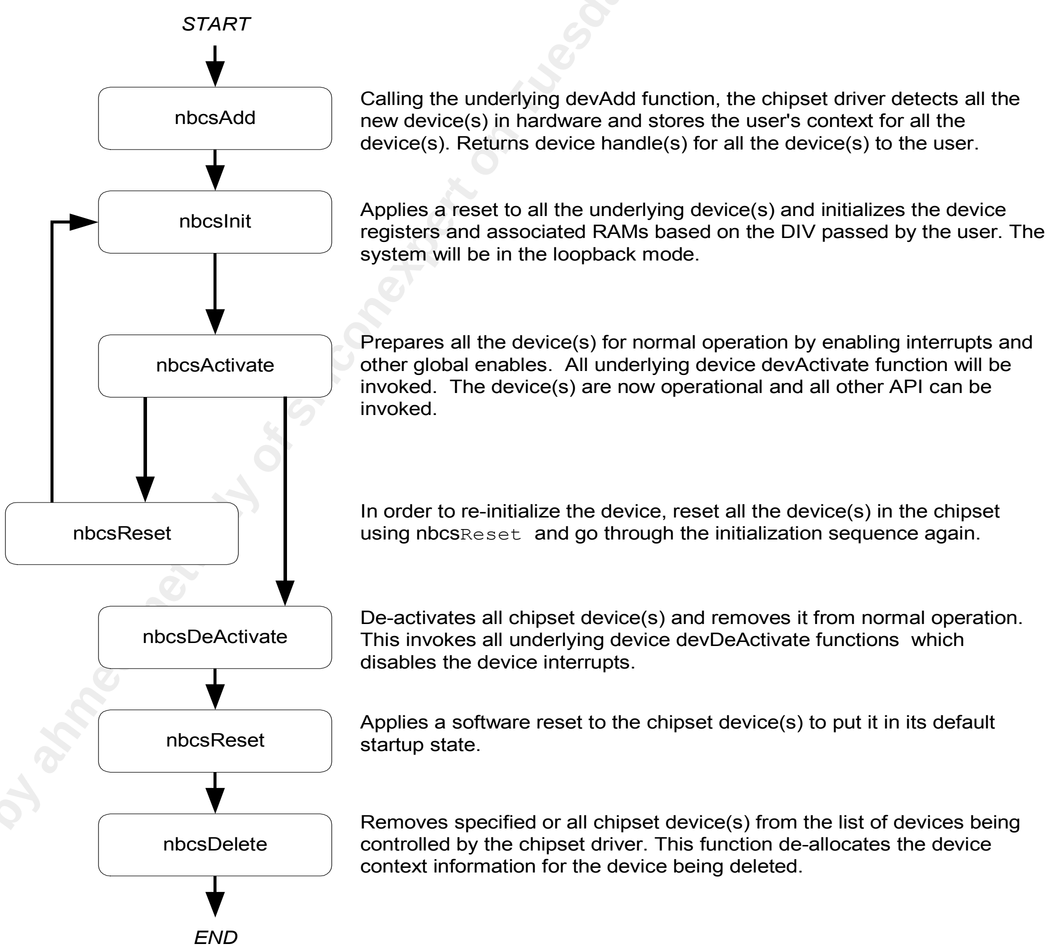


Chipset Device Management

The following figure shows the typical function call sequences that the chipset driver uses to add, initialize, re-initialize, and delete the Narrowband Chipset device. The Chipset driver components (devices) can be added or deleted individually or as a group. User can add/delete individual devices by referring to the device state diagram. For instance, if a particular device has to be taken out from the ACTIVE state, user can use `nbcsReset` to bring the device to PRESENT state and then `nbcsDelete` to remove the device from the chipset driver. Similarly, a new device can be added to a system at any time by calling `nbcsAdd`, and then `nbcsInit` to bring the new device to the INACTIVE state.

The normal sequence is to add all chipset devices individually first, and then initialize all the devices with the corresponding DIV. Then, the device should be activated by calling `nbcsActivate`. The same applies to `nbcsDeActivate, which` moves the chip state from ACTIVE to INACTIVE. Activating, deactivating, or resetting individual (SBS) devices are encouraged only when that devices is a new addition or if the device needs to be taken out from the chipset.

Figure 20: Chipset Device Management Flow Diagram



Group Management

The group concept allows user to carry out operations on a group of devices conveniently. User can freely group devices into meaningful groups so that they can be processed as a unit. For instance, user can instantiate a "core fabric" group that contains all NSE devices in a multi-stage fabric system.

Groups are defined by the use of the `nbcsGroupAdd` function call. At the time the group is defined, and at all times after that, the group state is determined by the states of the constituent devices. If all devices within a certain group are in the same state, then the group is in that state as well. If not all devices within a group are in the same state, then the group state is indeterminate.

The user can choose to use only group-based functions to initialize the chipset . If so, the group management flow diagram is identical to the device management flow diagram shown in Figure 18, with the exception that function `nbcsGroupDelete` has to be employed to delete a group. As the various group management functions are called, all devices within the specified group are transitioned to the appropriate states.

However, devices may be members of multiple groups. Because of this, group states do not always transition in the same manner as in the device state diagram. In Figure 21, say one group (A) has been established through the use of `nbcsGroupAdd` followed by `nbcsInit`, and a different, disjoint group (B) of devices has been established through the use of `nbcsGroupAdd`, `nbcsInit`, and `nbcsActivate`. Now all devices in group A, as well as group A itself, are considered to be in the INACTIVE state. All devices in group B, as well as group B itself, are considered to be in the ACTIVE state.

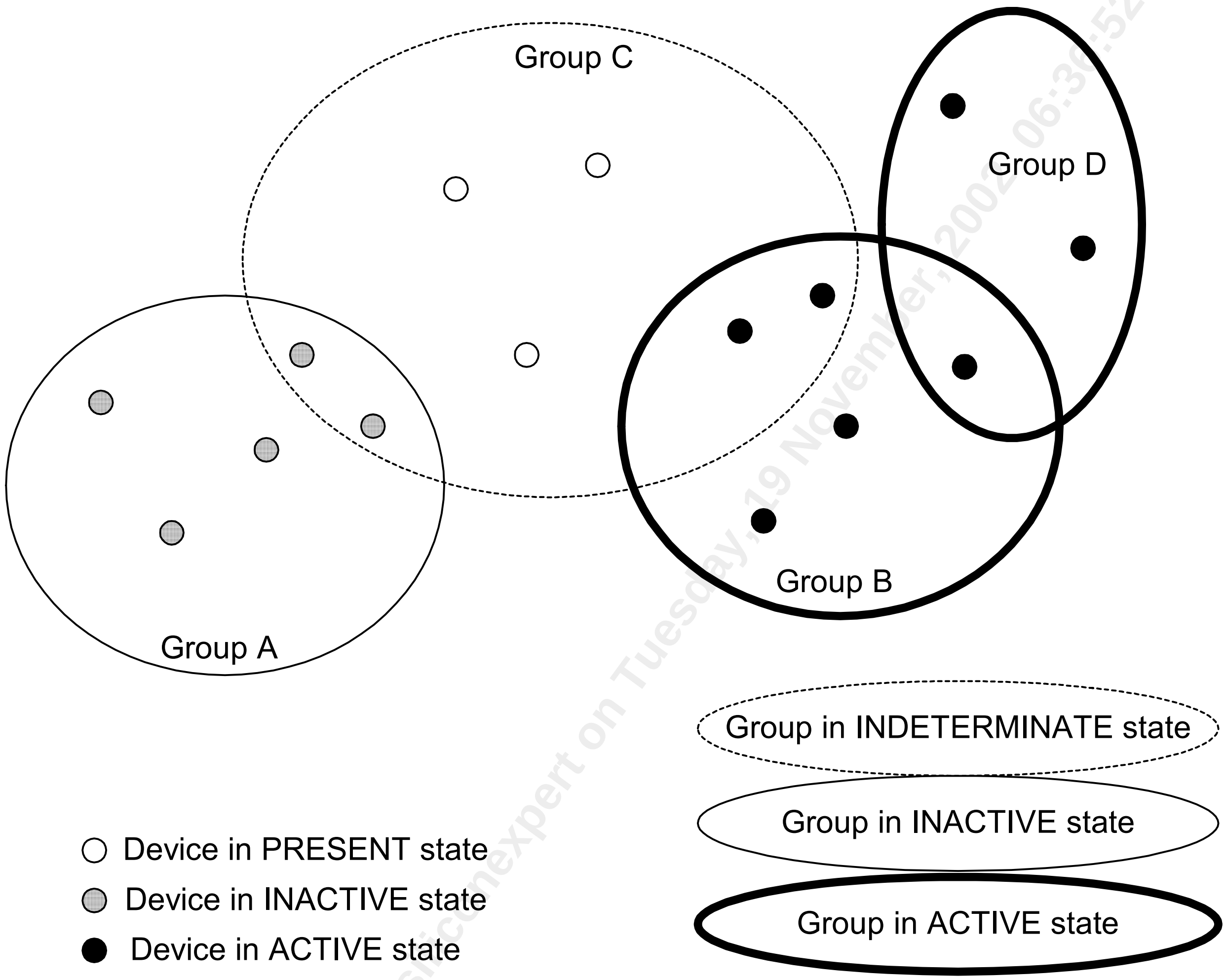
If a new group, say group C, is formed, consisting of some of group A, some of group B, and some new devices, `nbcsGroupAdd` can be called to create the group. Group C now has state of INDETERMINATE; the devices within group C have states of ACTIVE, INACTIVE, or PRESENT. Note that the USER would not be able to call `nbcsInit` on group C, as group C is not in the PRESENT state. Rather, if the USER wants the devices with PRESENT state to transition into the INACTIVE state, the USER must call `nbcsInit` on each such device.

On the other hand, say a new group (D) is created, using only the single function `nbcsGroupAdd`, out of part of group B and some new ACTIVE devices that were previously brought into the ACTIVE state with device management functions. This new group is in the ACTIVE state, since all of its devices are ACTIVE.

New devices can be added to and deleted from an existing group by calling `nbcsGroupAdd` and `nbcsGroupDelete` respectively. The state of a group can be retrieved by API function `nbcsGroupGetState`.

Thus, it is most expedient to use the Group Management functions (`nbcsGroupAdd`, `nbcsGroupDelete`, etc.) on non-overlapping groups of devices, in order to initialize the devices conveniently. Later, other potentially overlapping groupings can be made, to facilitate the commands sent during normal operation. However, the user is cautioned against using the group functions to cause state transitions on overlapping groups.

Figure 21: Example of overlapping groups



Typical CSD Startup Sequence

After the CSD module has been started and initialized properly and all the devices being added to the CSD (using module and device management APIs), there are some additional information regarding the system that needs to be furnished for proper operation.

1) (Defining the physical wiring of SBSs and the NSE core). It is essential to define how all the SBS (both ingress and egress direction) devices are wired to the NSE core. User calls API function `nbcsFmgtDefWiring` to define the wiring topology.

2) (Defining the payload types) API function `nbcsIntfCfgPyld` is used to configure the payload type of the SBS, for both the ingress and egress sides. The same function is used regardless of what bus mode the system is set to, SBI336 or TeleCombus. In the case of SBI bus, further tributaries configuration can be carried out using API `nbcsIntfCfgTrib.` This function is not applicable when the system is configured to TeleCombus mode.

3) (Put the system in loopback state). The loopback state is the point of reference for the system, which then evolves from this point with the addition of connections, etc… All the traffic is being looped back at this state. User should call the API function `nbcsFmgtSetLpbkMode, which` updates the offline page of all the local devices and resets the OPA library to clear all connections to support the loopback mode. User should then perform a synchronized page switch to put the settings in effect. Please refer to the subsequent section for more details on this operation.

4) (Setting up egress bus integrity for SBS/SBSLITE revision B devices only). As soon as the payload types are defined for all SBS devices, the egress bus integrity can be set up prior to any call activities. Preserving the bus integrity for each outgoing SPE is essential to the downstream device. For example, the J1 byte in each SPE has to be set up properly; the V1 byte has to be valid in a T1 or E1 tributary inside the SPE, etc. The function `nbcsFmgtUnMapTrib` is used to setup the integrity (as long as the SPE types have been properly defined). The side effect is all data in the payload is overwritten by zero and the loopback state will be disturbed. This step is optional.

SBS and NSE devices can be present locally or remotely depending on the system configuration, In a typical centralized system configuration, all devices are local and are under the control of a microprocessor that also runs the CSD. In a distributed system configuration, SBS and NSE devices that make up the switching fabric may be scattered across line, core or standalone processor cards and are under the control of multiple microprocessors. The CSD can be configured to run in all the cards with any combination of SBS/NSE devices present locally to the card.

In a typical centralized configuration, NSE/SBS devices are present and the OPA is expected to perform the routing; hence, the following fields in the `MIV sbsDrvPresent`, `nseDrvPresent`, and `opaLibUse` should all be set to 1. In a typical line card, it is not expected to contain any physical NSE devices nor will it run the OPA routing; hence, the following fields in the `MIV sbsDrvPresent`, `nseDrvPresent`, and `opaLibUse` should be set to 1, 0, and 0 respectively. Likewise, in a typical core card configuration, the fields `sbsDrvPresent`, and `nseDrvPresent`, should be set to 0 and 1 respectively. The field `opaLibUse` can be either 0 or 1 depending on whether the OPA routing is run locally or elsewhere. In a typical standalone system that is intended to host the CSD with the OPA, `sbsDrvPresent`, `nseDrvPresent`, and `opaLibUse` should then be set to 0, 0 and 1 respectively since the SBS and NSE device drivers are absent. There should only be one CSD module in the system with the field `opaLibUse` set to 1.

SBS devices that are not physically attached to the CSD-distributed-core or CSD-standalone do have to be added even in the distributed core or standalone CSD. This creates a logical proxy of the actual remote SBS on the side of the distributed-core/standalone CSD. However, note that the NSE devices do not have to be added on the remote CSD side.

It is trivial in the case of a centralized configuration for the initialization sequence. In the case of a distributed model, further clarification is needed. Defining the physical wiring is only needed in the CSD with the OPA installed, i.e., in a distributed core CSD or the standalone CSD. This step essentially provides the CSD/OPA the necessary information of the actual physical wiring. For step (2), the payload type configuration has to be called in the CSD-remote, the CSD-distributed-core and the CSD-standalone.

The following is an example of the API sequences required for 2 SBSs and 1 NSE. Two examples, centralized and distributed, are shown.

(A) Centralized

1) `nbcsModuleOpen()`
2) `nbcsModuleStart()`
3) `nbcsAdd(sbs1)`
4) `nbcsAdd(sbs2)`
5) `nbcsAdd(nse)`
6) `nbcsFmgtDefWiring()`
7) `nbcsIntfCfgPyld()`
8) `nbcsIntfCfgTrib()` if in SBI mode
9) `nbcsFmgtSetLpbkMode()`
10) `nbcsFmgtUnMapTrib() opt`

(B) Distributed

Distributed-remote#1	Distributed-remote#2	Distributed-core
1) `nbcsModuleOpen()`	1) `nbcsModuleOpen()`	1) `nbcsModuleOpen()`
2) `nbcsModuleStart()`	2) `nbcsModuleStart()`	2) `nbcsModuleStart()`
3) `nbcsAdd(sbs1)`	3) `nbcsAdd(sbs2)`	3) `nbcsAdd(nse)`
4) `nbcsAdd(sbs1)opt`	4) `nbcsAdd(sbs1)opt`	4) `nbcsAdd(sbs1)`
5) `nbcsAdd(sbs2)opt`	5) `nbcsAdd(sbs2)opt`	5) `nbcsAdd(sbs2)`
6) `nbcsFmgtDefWiring()opt`	6) `nbcsFmgtDefWiring()opt`	6) `nbcsFmgtDefWiring()`
7) `nbcsIntfCfgPyld()`	7) `nbcsIntfCfgPyld()`	7) `nbcsIntfCfgPyld()`
8) `nbcsIntfCfgTrib()opt`	8) `nbcsIntfCfgTrib()opt`	8) `nbcsIntfCfgTrib()opt`
9) `nbcsFmgtSetLpbkMode()`	9) `nbcsFmgtSetLpbkMode()`	9) `nbcsFmgtSetLpbkMode()`
		10) `nbcsFmgtUnMapTrib()`

Connection Setup and Teardown

Regardless of what the system configuration is, this section lists out the sequence of events that has to happen for a coordinated page switch for the fabric: The next section then goes into greater details on how this sequence of operation is handled in different configurations.

1) Call the fabric management API to request call connections

2) Retrieve all the changed SBS and/or NSE devices settings to support the new connection

3) Write new settings to inactive pages for all affected SBS and/or NSE devices

4) Determine the pending active pages of all the devices.

5) Switch the active page number of all the affected SBS and NSE devices (if any) by toggling the pages of all the devices. This operation has to be synchronized with the C1 frame pulse.

6) Update (synchronize) the settings between the active and inactive page in all SBS and/or NSE devices. This step is required only if the page setting automatic update feature (this is the field `pageAutoSync` configurable via the MIV) for the system is off; otherwise, the settings between the active and inactive pages are synchronized automatically.

7) User can later call the unmapping function in the fabric management API to remove the connection. Then, follow the same logic as if it is a call setup, i.e., repeat step (2) to (6) after a call removal. Settings <u>may</u> be changed after a call disconnect.

Centralized Configuration

In the centralized configuration, the CSD has a large amount of autonomy to perform the page switching. Setting up calls across the fabric requires invocation of a handful of APIs.

1) User calls `nbcsFmgtMapTrib/nbcsFmgtMapDS0` to request new call connections.

2) Populates all the incremental settings of both NSE and SBS devices to the hardware by calling `nbcsFmgtGetChgMap`.

3) Invoke `nbcsStswTogglePage` to toggle all the connection pages in the chipset synchronously with the C1 frame pulse.

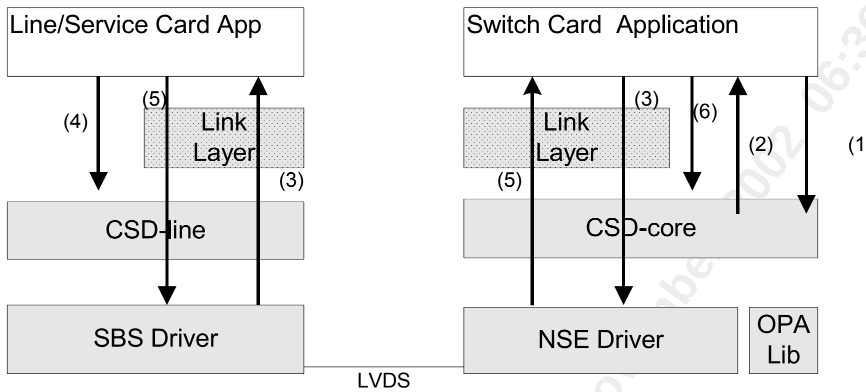
4) (if applicable) Keep the new settings by calling `nbcsStswCopyPage` to synchronize the settings between the active and inactive pages of all the devices.

5) User can call `nbcsFmgtUnMapTrib/nbcsFmgtUnMapDS0` to remove the connection if necessary.

6) (if applicable) Keep the new settings by calling `nbcsStswCopyPage` to synchronize the settings between the active and inactive pages of all the devices.

Distributed Configuration

Figure 22: Chipset Driver Call Setup Flow Diagram – Distributed Model



The following lists out the steps to take for setting up new connections.

1) On the switch card side, user calls `nbcsFmgtMapTrib/nbcsFmgtMapDS0` to request new call connections.

2) `R`etrieves the incremental settings of all remote SBS devices from the CSD by calling `nbcsFmgtGetChgMap`. This function also populates new settings to the local NSE device(s).

3) Distribute the settings to remote SBSs by ILC (via a link layer protocol) or other means such as Ethernet.

4) The remote SBS application receives the new settings and calls `nbcsStswMapSlot` to update the settings for all SBSs.

5) The remote SBS application then sends acknowledgement back to NSE side. With acknowledgement from all remote SBSs, the switch card application is assured of all SBS settings being transmitted correctly and proceeds with performing a synchronized page switch. (Note: the link layer including any of the acknowledgement protocol is beyond the scope of the CSD.)

6) Invoke `nbcsStswTogglePage` to toggle all the connection pages in the chipset synchronously with the C1 frame pulse.

7) (if applicable) Synchronize all the settings between the active and inactive pages of all the devices by calling API `nbcsStswCopyPage`.

8) User can call `nbcsFmgtUnMapTrib`/`nbcsFmgtUnMapDS0` to remove the connection if necessary.

9) Follow the sequence as if it is a call setup, i.e., step (2) to (7)

It is imperative to point out that the CSD uses the in-band link controller to access and change connection page numbers for remote SBSs from the NSE side. In other words, API `nbcsStswTogglePage`, `nbcsStswGetPage`, and `nbcsStswSetPage` are operational only (hence the aforementioned sequence for call management) if the field `pageSwapCntl` configurable from MIV is set to be controlled by the PAGE header bits in the in-band link controller. If the system configuration uses other means to synchronize page switch for the entire fabric such as an external hardware line, user is advised to call API `nbcsEventDetectC1FP` to enable the C1 frame pulse detection and embeds the necessary logic (such as toggling this hardware line in our example) in the callback function `cbackNbcsC1FP` to orchestrate the switch.

1+1 Port Protection in Distributed System

Here is the sequence for setting up the 1+1 port protection and triggering a switch over. Note that the protection port payload types has to be identical to that of the working port. This also applies to the case of 1:N protection. The payload types of the protection port has to be identical to that of the working port:

1. User calls `nbcsFmgtSetProtect` to specify what protection, 1+1 or 1:N to setup, and supplies it with the working and protection port(s).
2. Define the payload types for both the working and protect port to be identical.
3. When a switch over is deemed necessary (determined by some upper layer signaling protocol), user can call `nbcsFmgtSwitchProtect` to initiate a switchover.
4. Upon receiving a success from the chipset driver, the application should then call `nbcsFmgtGetChgMap` to retrieve new settings. The rest is similar to the call/setup and teardown case in previous section.

Adding New Line/Service Card

In the event when a line/service card needs be added, it can be achieved without affecting the rest of the cards and traffic flow.

1) Call `nbcsAdd` and then `nbcsInit` with a valid DIV

2) Call `nbcsFmgtDefWiring` to define the new wiring topology.

3) Call `nbcsIntfCfgPyld` and then `nbcsIntfCfgTrib` if necessary to configure the payload type and the tributaries.

4) (optional) Call `nbcsActivate` to bring the new device to ACTIVATE. The new card is now ready to source or sink traffic. This is to bring the new device to the same state as the other devices so that their device states are synchronized.

Replacing Working Line/Service Card

In the event when a line/service card needs be replaced, it can be taken out without affecting the rest of the cards and traffic flow.

1. First removes all connections to the SBS to be removed
2. Call `nbcsReset` and then `nbcsDelete` to remove the SBS from the fabric.
3. Add a SBS device in the new line card using the sequence described in previous section.
4. Assuming the old SBS is 1+1 protected by another SBS that has been active since the working SBS is taken out of service, call `nbcsStswCopyPage` to synchronize the setting between the protected SBS and the new SBS.
5. Establish the 1+1 protection between the new SBS and the protection SBS by calling `nbcsFmgtSetProtect` and perform a switchover by calling `nbcsFmgtSwitchProtect` to restore traffic flow in the new working SBS.

3.5 Event Processing

The Narrowband Chipset driver supplies all the callback functions for the underlying device drivers. When an underlying event occurs and is detected by the hardware, an interrupt is generated and serviced by the corresponding device driver which invokes a callback to the upper layer application, in this case, the narrowband chipset driver. The CSD then processes the callbacks and forwards them to the application code for events that are enabled.

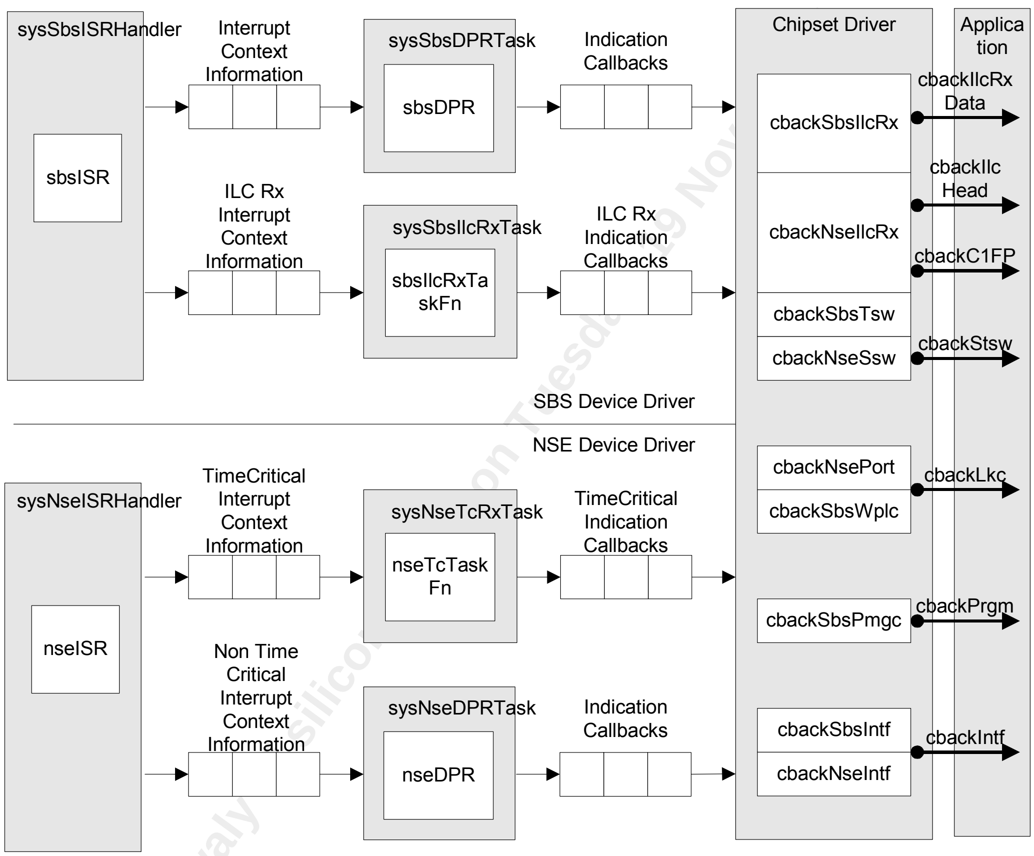
The following is an overview of the interrupt service model used in the device drivers to service device interrupts. Please refer to the NSE and SBS device driver user manual documents for more details in interrupt processing. Basically, the device driver services the device interrupts using an Interrupt-Service Routine (ISR) that traps interrupts. In the ISR, the device driver reads the master interrupt status registers to find out what the interrupt cause(s) is and sends the necessary information to the deferred processing routines (DPR) that actually process the interrupt conditions and clears them. This architecture enables the ISR to execute quickly and then exit. Most of the time-consuming processing of the interrupt conditions is deferred to the DPR by queuing the necessary interrupt-context information to the DPR task. The DPR function runs in the context of a separate task within the RTOS. The DPR then processes the interrupt information and takes appropriate action based on the specific interrupt condition detected. As the nature of this processing can differ from system to system, DPR calls different indication callbacks for different interrupt conditions.

The CSD receives these callbacks from the underlying device drivers, processes them, sorts them according to their categories and then issues callbacks to the upper application layers. Application has the option to enable and disable any events. After initialization, the CSD by default enables all events and reports them to the application code unless they are turned off by the application. Some events are recommended to be on at all times under normal circumstances such as in-band link events. They are by default, turned on by the CSD.

Events that are reported to the application via callbacks are as follows: C1 frame pulse received, in-band link data available, in-band link header bits changed, interface events such as parity error, LVDS link events such as out-of-frame alignment and FIFO error, space/time switch events such as page changed, and PRBS generator and monitor events from SBS devices.

Figure 23 illustrates the interrupt service model used in the Narrowband Chipset driver design. Users can customize these callbacks to suit their system. Please see page 160 for example implementations of the callback functions.

Figure 23: NSE/SBS Chipset Driver Event Processing Model – Interrupt-Mode



Calling nbcsPoll

Instead of employing an interrupt service model for the underlying devices, the user can use a polling service model in the Narrowband Chipset driver to process the device's event.

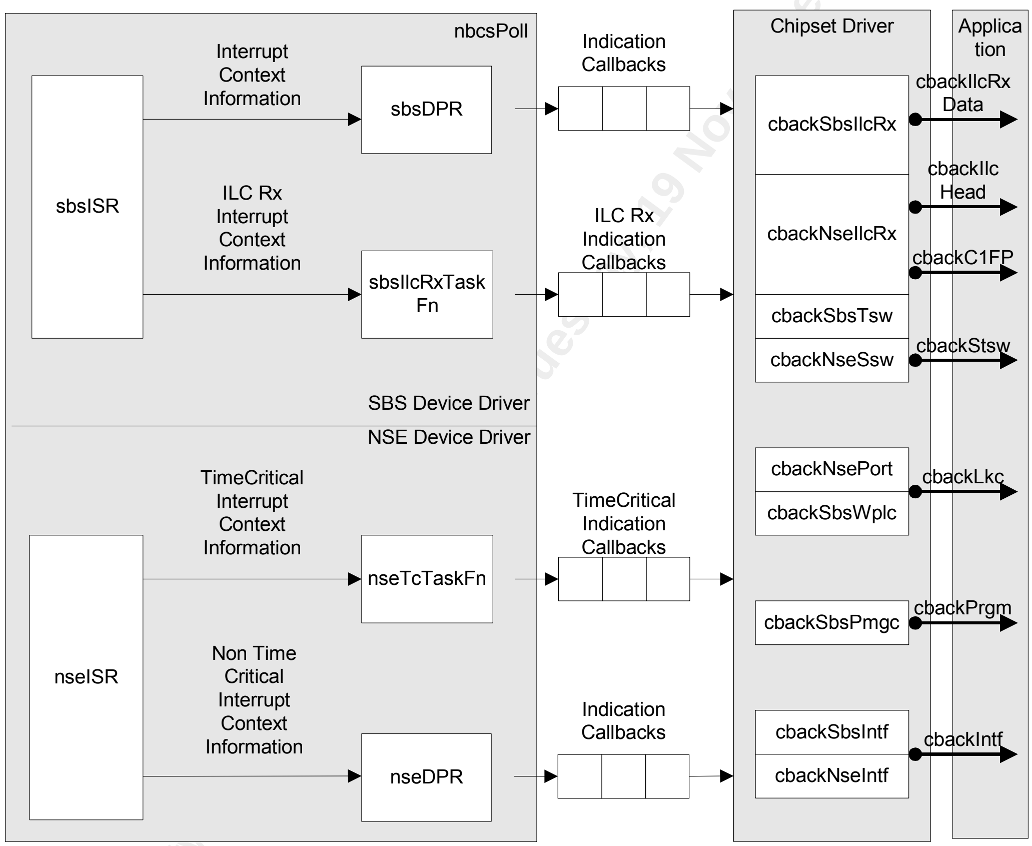
Figure 24 illustrates the polling service model used in the Narrowband Chipset driver design.

The mode, polling or interrupt, is selected via the MIV at the module initialization. In polling mode, the application is responsible for calling `nbcsPoll` often enough to service any pending error or alarm conditions. When `nbcsPoll` is called, the underlying polling functions of the NSE and SBS device driver are called internally.

The respective device ISR (interrupt service routine) functions read from the master interrupt-status register of the SBS and NSE. If at least one valid event is found then the corresponding ISR invokes the its DPR (deferred processing routine) directly. The event eventually is reported via the registered callback functions to the application.

It is imperative to point out that some time critical API will not function properly when the driver is set up in polling mode. For instance, the `nbcsStswTogglePage` will not operate correctly since it relies on switching pages of all registered devices in a timely fashion by monitoring the received C1 frame pulse. The same applies to API `nbcsEventDetectC1FP`.

Figure 24: NSE/SBS Chipset Driver Event Processing Model – Polling Mode



3.6 CSD API Availability

The availability of a CSD API depends largely on whether a device is registered as a local or remote device. In general, API that requires the actual physical device to perform the task returns an error code if the device is not present locally. The exception is all module/device/group management APIs. The fabric management APIs are also available regardless of the device status if the OPA is present locally.

4 DATA STRUCTURES

This section of the manual describes the elements of the driver that configure and control its behavior. Included here are the constants, variables, and structures that the Narrowband Chipset device driver uses to store initialization, configuration, and status information. For more information on our naming convention, please see Appendix A (page 178).

4.1 Constants

The following constants are used throughout the driver code:

- <Narrowband Chipset ERROR CODES>: contains error codes returned by the API functions and used in the global error number field of the Chipset Module Data Block (CSMDB) and Chipset Device Data Block (CSDDB). For a complete list of error codes, see Appendix B (page 178).
- NBCS_MAX_SBS and NBCS_MAX_NSE: define the maximum number of SBS and NSE devices that can be supported by the driver. This constant must not be changed without a thorough analysis of the consequences to the driver code.
- NBCS_MAX_SBS_INIT_PROFS and NBCS_MAX_NSE_INIT_PROFS: define the maximum number of profiles for SBS and NSE devices that can be supported by the driver.
- NBCS_MAX_GROUP: define the maximum number of groups that can be supported by the driver.
- NBCS_MOD_START, NBCS_MOD_IDLE, and NBCS_MOD_READY: are the three possible module states (stored in the CSMDB as stateModule).
- NBCS_START, NBCS_PRESENT, NBCS_ACTIVE, and NBCS_INACTIVE: are the four possible device states (stored in the CSDDB as stateDevice).
- eNBCS_TCBTRIB_TYPE: NBCS_TCBVT_VT15, NBCS_TCBVT_VT2, NBCS_TCBVT_VT3 NBCS_TCBVT_VT6, NBCS_TCB_DS3E3 and NBCS_TCB_STST3C: The first four are the four possible virtual tributary types VT1.5, VT2, VT3, and VT6 in a virtual group. For SDH, select VT1.5 for VC-11, VT2 for VC-12, and VT6 for VC-2. NBCS_TCB_DS3E3 is to specify the payload type as DS3 or E3. The last one is for specifying payload type to be STS-3c or STS-3 in SONET or STM-1 in SDH format.
- eNBCS_SBITRIB_TYPE: NBCS_T1_PYLD, NBCS_E1_PYLD , NBCS_DS3_E3_PYLD, NBCS_FRAC_RT_PYLD: are the four possible tributary types for T1, E1 DS3/E3 and fractional rate.
- eNBCS_BUSTYPE: NBCS_BUS_SBI, NBCS_BUS_TCB denote the SBI bus or TeleCombus mode for the system.
- eNBCS_IO_BUSMODE: NBCS_IO_BUS_QUAD, or NBCS_IO_BUS_SINGLE: denote the two possible bus modes namely quad bus (4 x 19.44 MHz) or single bus (1 x 77.76 MHz) in either SBI or TeleCombus mode.

- eNBCS_PORTPROTECT: NBCS_PORTPROTECT_NONE, NBCS_PORTPROTECT_1PLUS1, NBCS_PORTPROTECT_1FORN and NBCS_PORTPROTECT_UPSR for 1:1, and 1:N port protection and UPSR protection. Note that NBCS_PORTPROTECT_NONE is not for the user. It is reserved for the internal use of the driver.
- eNBCS_MULTIFRM_MODE: NBCS_MF_4 and NBCS_MF_48 for multi-frame consists of 4 frames and 48 frames respectively.
- eNBCS_ACCESSMODE_STSW: NBCS_STSW_UNICAST, NBCS_STSW_TIME_INPORT, NBCS_STSW_TIME_OUTPORT, NBCS_STSW_INPORT, NBCS_STSW_OUTPORT and NBCS_STSW_MAP for various mapping mode in space/time switch configuration
- eNBCS_ILC_FIFO_TIMEOUT: NBCS_ILC_FIFO_125US, NBCS_ILC_FIFO_250US, NBCS_ILC_FIFO_375US and NBCS_ILC_FIFO_500US for selecting the FIFO timeout constant in the ILC RxFIFO.
- eNBCS_LPBK: NBCS_O2ILPBK, NBCS_T82R8LPBK, and NBCS_T2RLPBK for SBS loopback
- eNBCS_DEVTYPE: NBCS_NSE20G, NBCS_NSE8G, NBCS_SBS, NBCS_SBSLITE and NBCS_SBSNSE_GROUP for device/group type identification.
- eNBCS_SWHMODE: NBCS_SWH_BYTE and NBCS_SWH_COLUMN for selecting the fabric switching mode, byte or column.
- eNBCS_WPLINK_CNTL: NBCS_LINK_CNTL_SW and NBCS_LINK_CNTL_HW for controlling the working and protect link control, hardware or software, in all the SBS devices.
- eNBCS_CONMAP_CNTL: NBCS_MAP_CNTL_SW, NBCS_MAP_CNTL_HW and NBCS_MAP_CNTL_ILC for selecting the connection map control via software, hardware pin, or ILC in all the SBS devices.
- eNBCS_FABRIC_TYPE: NBCS_FABRIC_STD, NBCS_FABRIC_DOUBLE_SBS, and NBCS_FABRIC_DOUBLE_SBSNSE for selecting the type of the underlying NSE/SBS fabric.
- eNBCS_FABRIC_SETTING: NBCS_SWITCHOVER_SETTING, and NBCS_CALL_SETTING for selecting the type of settings to retrieve.
- eNBCS_CALLTYPE: NBCS_CALL_MCAST, NBCS_CALL_UPSRDROP for selecting the type of calls to be set up/torn down.
- eNBCS_TMODE: NBCS_TMODE_MST, NBCS_TMODE_HPT and NBCS_TMODE_LPT for selecting different path termination mode, namely MST, HPT and LPT.
- eNBCS_CHKPT_TYPE: NBCS_CHKPT_OPA, and NBCS_CHKPT_CSD for distinguishing different type of checkpoints. This type is for internal use of the driver.

4.2 Structures Passed by the Application

These structures are defined for use by the application and are passed as arguments to functions within the driver. These structures are described below.

Module and Device Management

Chipset Module Initialization Vector: MIV

This structure contains module-level initialization parameters for the chipset driver. The user passes this structure as an input parameter in the nbcsModuleOpen function call.

- The variables maxNseDevs, maxSbsDevs, maxSbsInitProfs, maxNseInitProfs, and maxGroups define the maximum number of NSE and SBS devices that the chipset driver, the maximum number of initialization profiles for both devices and the maximum number of groups the chipset driver permits in the session. The numbers are used to calculate the amount of memory allocated for the chipset driver.
- cbackC1FP, cbackIlcRxData , cbackIlcHead, cbackIntf, cbackLkc, cbackStsw, and cbackPrgm are used to pass the addresses of application functions that are used by the chipset driver to inform the application code of pending events. If these fields are set to NULL, the application will not be notified of the events.

Table 5: Narrowband Chipset Module Initialization Vector: sNBCS_MIV

Field Name	Field Type	Field Description
perrModule	INT4 *	(pointer to) errModule (see description in the CSMDB)
maxNseDevs	UINT2	Maximum number of physical/logical NSE devices supported during this session
maxSbsDevs	UINT2	Maximum number of physical/logical SBS devices supported during this session
maxGroups	UINT2	Maximum number of groups supported during this session
maxSbsInitProfs	UINT2	Maximum number of SBS initialization profiles
maxNseInitProfs	UINT2	Maximum number of NSE initialization profiles
sbsDrvPresent	UINT1	Indicates whether the SBS device driver is present locally. 0 = absent, 1 = present
nseDrvPresent	UINT1	Indicates whether the NSE device driver is present locally. 0 = absent, 1 = present

Field Name	Field Type	Field Description
nopaLibUse	UINT1	Indicates whether the OPA library usage is required. 0 = not required, 1 = required
sysBusType	eNBCS_BUSTYPE	System bus type NBCS_BUS_SBI: SBI bus-based NBCS_BUS_TCB: TeleCombus-based
swhMode	eNBCS_SWHMODE	Fabric switching mode: NBCS_SWH_BYTE for byte switching mode NBCS_SWH_COLUMN for column switching mode
casMuxMode	UINT1	This field has dual meaning depending upon the bus and switching mode. a) CAS processing mode when in SBI byte mode. Note that OPA is the only scheduler allowed when CAS traffic is present 0 = no CAS traffic present 1 = T1 CAS traffic present 2 or above = E1 CAS traffic present This field is ignored when in SBI column mode b) Multiplexer control signal selection when in TeleCombus/SBI mode with doubled SBS or doubled SBS/NSE fabric. In general, OTAIS can be used for both HPT and MST modes, while OPL must be used for LPT mode. 0 = OPL signal non-zero = OTAIS signal This field is ignored if standard fabric is selected.
nseCoreType	eNBCS_DEVTYPE	Type of NSE device(s) that make up the space-switching core. Valid entries are NBCS_NSE20G or NBCS_NSE8G.

Field Name	Field Type	Field Description
pageAutoSync	UINT1	Automatic active page to inactive connection page synchronization. When this field is a logic one, the settings are copied from the active to inactive page after a page switch in all SBS and NSE devices in the system.
wpLinkCntl	eNBCS_WPLINK_CNTL	Source of control for the working and protection LVDS link in all SBSs: NBCS_LINK_CNTL_SW: software controls whether working or protection link is active NBCS_LINK_CNTL_HW: a hardware pin controls whether working or protection link is active
pageAutoUpdate	UINT1	Automatic connection setting update for all local devices: CSD automatically updates the offline connection map of devices under its control after a call request when this field is a logic one.
pageSwapCntl	eNBCS_CONMAP_CNTL	Source of control for the connection page switching in all SBSs: NBCS_MAP_CNTL_SW: software controls the page switching NBCS_MAP_CNTL_HW: hardware pin controls the page switching NBCS_MAP_CNTL_ILC: the PAGE bits in ILC controls the page switching. This option is required for API nbcsStswTogglePage to work properly.
coreDepth	UINT2	The depth of the NSE switch core
coreNumStage	UINT2	The number of stages of the NSE switch core

Field Name	Field Type	Field Description
fabType	eNBCS_FABRIC_TYPE	Type of switching fabric NBCS_FABRIC_STD: standard NSE/SBS fabric NBCS_FABRIC_DOUBLE_SBS: double SBS fabric. SBS devices are doubled up but NSE device(s) are not. . NBCS_FABRIC_DOUBLE_NSESBS: double SBS and NSE fabric. Both SBS and NSE devices are doubled up in the fabric.
mCastScheduler	UINT1	Call scheduler type: 0 = unicast scheduler (OPA), non-zero = multicast scheduler (LOPA)
pageAutoUpdate	UINT1	Automatic connection setting update for all local devices: CSD automatically updates the offline connection map of devices under its control after a call request when this field is a logic one.
pageAutoSync	UINT1	Automatic active page to inactive connection page synchronization. When this field is a logic one, the settings are copied from the active to inactive page after a page switch in all SBS and NSE devices in the system.
pageSwapCntl	eNBCS_CONMAP_CNTL	Source of control for the connection page switching in all SBSs: NBCS_MAP_CNTL_SW: software controls the page switching NBCS_MAP_CNTL_HW: hardware pin controls the page switching NBCS_MAP_CNTL_ILC: the PAGE bits in ILC controls the page switching. This option is required for API `nbcsStswTogglePage` to work properly.

Field Name	Field Type	Field Description
wpLinkCntl	eNBCS_WPLINK_CNTL	Source of control for the working and protection LVDS link in all SBSs: NBCS_LINK_CNTL_SW: software controls whether working or protection link is active NBCS_LINK_CNTL_HW: a hardware pin controls whether working or protection link is active
pollMode	UINT1	Polling mode flag: 0 = disabled (interrupt mode), 1 = enabled
cbackC1FP	NBCS_CBACK_TC	Callback function for C1 frame pulse reception
cbackIlcRxData	NBCS_CBACK_TC	Callback function for in-band link Rx data
cbackIlcHead	NBCS_CBACK	Callback function for in-band link header bits change
cbackIntf	NBCS_CBACK	Callback function for Interface/clock block events
cbackLkc	NBCS_CBACK	Callback function for LVDS link controller block events
cbackStsw	NBCS_CBACK	Callback function for Space/time Configuration block events
cbackPrgm	NBCS_CBACK	Callback function for PRGM block events

Device Initialization Vector: DIV

The following structure contains chipset device initialization parameters. The DIV has two kinds: NSE and SBS. Depending on the device, the user passes either the SBS or the NSE DIV structure as an input parameter in the nbcsInit function call to initialize a Narrowband Chipset device. The following is a description of the fields in those two DIV structures:

- valid is the parameter indicating the validity of the structure and the type of DIV. It should be assigned to the part number of the device, NBCS_SBS_PARTNUM, NBCS_SBS_LITE_PARTNUM, NBCS_NSE20G_PARTNUM or NBCS_NSE8G_PARTNUM.

- `intfBusCfg` is the structure that contains the interface bus configuration. It consists of mainly bus mode configuration.
- `lkcCfg[]` is the LVDS link controller configuration block.
- `ilcCfg` is the In-band link controller configuration block.

Table 6: Narrowband Chipset SBS Device Initialization Vector: sNBCS_DIV_SBS

Field Name	Field Type	Field Description
valid	UINT2	Indicates that this structure is valid. This value should be assigned to constant NBCS_SBS_PARTNUM for SBS and NBCS_SBS_LITE_PARTNUM for SBSLITE
intfBusCfg	sNBCS_CFG_INTF_BUS	Interface bus configuration block structure
ilcCfg	sNBCS_CFG_ILC	In-band link controller configuration block structure
lkcCfg [NBCS_SBS_NUM_LINKS]	sNBCS_CFG_LKC	LVDS link controller configuration block structure

Table 7: Narrowband Chipset NSE Device Initialization Vector: sNBCS_DIV_NSE

Field Name	Field Type	Field Description
valid	UINT2	Indicates that this structure is valid. This value should be assigned to constant NBCS_NSE20G_PARTNUM for NSE-20G and NBCS_NSE8G_PARTNUM for NSE-8G
ilcCfg	sNBCS_CFG_ILC	In-band link controller configuration block structure
lkcCfg [NBCS_NSE_MAX_LINKS]	sNBCS_CFG_LKC	LVDS link controller configuration block structure

Group Initialization Vector: GIV

The following structure contains chipset group initialization parameters. The following is a description of the fields in the GIV structures:

- `perDevDiv` is the parameter indicating whether all devices of same type are initialized to the same DIV/Initialization Profile or an array of DIV/Initialization Profiles.

- `useInitProf` is the parameter indicating whether initialization profiles or DIVs are used in the structure.
- `pSbsDiv` is an array of SBS DIVs
- `pNseDiv` is an array of NSE DIVs
- `pSbsInitProf` is an array of SBS initialization profiles
- `pNseDiv` is an array of NSE initialization profiles

Table 8: Narrowband Chipset Group Initialization Vector: sNBCS_GIV

Field Name	Field Type	Field Description
perDevDiv	UINT1	If non-zero, each device is initialized with its own type of DIV. If FALSE, all devices of a given type are initialized with the same DIV.
useInitProf	UINT1	If non-zero, the initialization profile is used.
pSbsDiv	sNBCS_DIV_SBS*	(array of) SBS DIVs; if `perDevDiv` is a logic one, these DIVs are used to initialized each SBS device; otherwise, `pSbsDiv[0]` is used for all SBS devices. If this is NULL, initialization profiles, `pSbsInitProf,` are used instead.
pNseDiv	sNBCS_DIV_NSE*	(array of) NSE DIVs; if `perDevDiv` is a logic one, these DIVs are used to initialized each NSE device; otherwise, `pNseDiv[0]` is used for all NSE devices. If this is NULL, initialization profiles, `pNseInitProf,` are used instead.
pSbsInitProf	UINT2 *	(array of) SBS initialization profiles; if `perDevDiv` is a logic one and `pSbsDiv` is NULL, these profiles are used to initialize the SBS devices; otherwise, `pSbsInitProf[0]` is used for all SBS devices.

Field Name	Field Type	Field Description
pNseInitProf	UINT2 *	(array of) NSE initialization profiles; if perDevDiv is a logic one and pNseDiv is NULL, these profiles are used to initialize the NSE devices; otherwise, pNseInitProf[0] is used for all NSE devices.

Device Information Block: DEVINFO

The following structure contains chipset device information block. The following is a description of the fields in the structures:

- devType is the device type
- pBaseAddr is the base address of the device.
- devNum1 is the first device number assigned by the user. For SBS device, this is just the SBS user number. For NSE device, this is the device number and this should follow the following calculation: (stageNum * MAX_DEPTH) + depthNum where stageNum denotes the stage number, depthNum denotes the depth number of the device, and MAX_DEPTH denotes the depth of the space stage. Both stageNum and depthNum starts from zero. For instance, devNum1 should be zero if the fabric is 1-stage.
- devNum2 is the second device number assigned by the user. It is not used for SBS device. For NSE device, this indicates whether the NSE device is a secondary device in a doubled NSE/SBS fabric configuration.
- devNum3 is the third device number assigned by the user. It is reserved for future use.
- altMuxVal is the multiplexer control value. This is only applicable to doubled SBS or doubled SBS/NSE fabric configurations. If this field is zero, the default control value is selected by the CSD (the control value depending upon the field casMuxMode in MIV). However, user can override this value by supplying a non-zero value for this field. This value should not be arbitrary and should match the underlying hardware.
- isLocal indicates whether this device is present locally under the control of the same microprocessor that also runs the CSD. If a device is local, the CSD will invoke any underlying device driver API, if necessary, for a particular operation; otherwise, CSD treats the device as a logical one, or remote, and no API calls will be made to the underlying device driver. The sbsDrvPresent and nseDrvPresent fields in MIV take precedence over this field. In other words, setting the field isLocal to one if the underlying driver(s) is/are absent will be ignored by the CSD which will treat the device logical in that situation.

Table 9: Narrowband Chipset Device Information Block: sNBCS_DEVINFO

Field Name	Field Type	Field Description
devType	eNBCS_DEVTYPE	Device type: one of the following: NBCS_NSE20G, NBCS_NSE8G, NBCS_SBS, or NBCS_SBSLITE
pBaseAddr	void*	Base address of the device
devNum1	UINT1	First device number. For SBS devices, it is the SBS number. For NSE, it is the device number and should equal to (stageNum * MAX_DEPTH) + depthNum.
devNum2	UINT1	Second device number. Not used in SBS devices. For NSE, it indicates if the device is primary or secondary in a doubled NSE/SBS fabric configuration.
devNum3	UINT1	Third device number. Reserved for future use.
altMuxVal	UINT4	Alternate multiplexer control value
isLocal	UINT1	Flag indicating whether the device is under the control of the microprocessor that also runs the CSD. 0 = local, 1 = remote.

Event Servicing

SBS Event Processing Enable/Disable Mask (MASK_EVT_SBS)

This structure is used to pass/retrieve the event processing mask settings for the SBS device in a Narrowband Chipset. This structure is used in the nbcsEventSetMask, nbcsEventGetMask and nbcsEventClearMask function calls.

Table 10: Narrowband Chipset Event Mask for SBS Device: sNBCS_MASK_EVT_SBS

Field Name	Field Type	Field Description
intf	sNBCS_MASK_EVT_INTF	Event mask for Interface/Clock block

Field Name	Field Type	Field Description
tsw [NBCS_SBS_NUM_TSW]	sNBCS_MASK_EVT_STSW	Event mask for time switch configuration block
lkc [NBCS_SBS_NUM_LINKS]	sNBCS_MASK_EVT_LKC	Event mask for LVDS link control block
ilc [NBCS_SBS_NUM_LINKS]	sNBCS_MASK_EVT_ILC	Event mask for in-band link controller block
prgm [NBCS_SBS_NUM_LINKS]	sNBCS_MASK_EVT_PRGM	Event mask for PRGM block

NSE Event Processing Enable/Disable Mask (MASK_EVT_NSE)

This structure is used to pass/retrieve the event processing mask settings for the NSE device in a Narrowband Chipset. This structure is used in the `nbcsEventSetMask`, `nbcsEventGetMask` and `nbcsEventClearMask` function calls.

Table 11: Narrowband Chipset Event Mask for NSE Device: sNBCS_MASK_EVT_NSE

Field Name	Field Type	Field Description
intf	sNBCS_MASK_EVT_INTF	Event mask for Interface/Clock block. All events from this block are report via `cbackIntf` with the exception of C1 frame pulse events which are reported via `cbackC1FP`
ssw	sNBCS_MASK_EVT_STSW	Event mask for space switch configuration block. All events from this block are report via `cbackStsw`
lkc [NBCS_NSE_MAX_LINKS]	sNBCS_MASK_EVT_LKC	Event mask for LVDS link control block. All events from this block are report via `cbackLkc`
ilc [NBCS_NSE_MAX_LINKS]	sNBCS_MASK_EVT_ILC	Event mask for in-band link controller block. All events from this block are report via either `cbackIlcRxData` or `cbackIlcHead`

Interface/Clock Block Event Mask (MASK_EVT_INTF)

Table 12: Narrowband Chipset Event Mask for Interface/Clock Block: sNBCS_MASK_EVT_INTF

Field Name	Field Type	Field Description
refDllError	UINT1	reference DLL error event (SBS device only): 0 = disable, 1 = enable
sysDllError	UINT1	system DLL error event (SBS device only): 0 = disable, 1 = enable
csuLock	UINT1	CSU lock event: 0 = disable, 1 = enable
rxBusParityErr	UINT1	Receive bus parity error: 0 = disable, 1 = enable
outCollision [NBCS_QUAD_BUS]	UINT1	Collision on outgoing SBI bus (SBS device only): 0 = disable, 1 = enable
inBusParityErr [NBCS_QUAD_BUS]	UINT1	Incoming bus parity error (SBS device only): 0 = disable, 1 = enable

Space/Time Switch Configuration Block Event Mask (MASK_EVT_STSW)

Table 13: Narrowband Chipset Event Mask for Space/Time Configuration Block: sNBCS_MASK_EVT_STSW

Field Name	Field Type	Field Description
pageSwap	UINT1	Change in connection page swap status event: 0 = disable, 1 = enable
pageUpdate	UINT1	Change in the page update status event: 0 = disable, 1 = enable

LVDS Link Control Event Mask (MASK_EVT_LKC)

Table 14: Narrowband Chipset Event Mask for LVDS Link Control Block: sNBCS_MASK_EVT_LKC

Field Name	Field Type	Field Description
txFifoErr	UINT1	Tx FIFO error event: 0 = disable, 1 = enable
rxFifoErr	UINT1	Rx FIFO error event: 0 = disable, 1 = enable

Field Name	Field Type	Field Description
oca	UINT1	Out-of-character alignment event: 0 = disable, 1 = enable
ofa	UINT1	Out-of-frame alignment event: 0 = disable, 1 = enable
lcv	UINT1	Line code violation event: 0 = disable, 1 = enable

In-band Link Controller Block Event Mask (MASK_EVT_ILC)

Table 15: Narrowband Chipset Event Mask for In-band Link Controller Block: sNBCS_MASK_EVT_ILC

Field Name	Field Type	Field Description
fifoOverflow	UINT1	Rx FIFO overflow event: 0 = disable, 1 = enable. This event is reported via the cbackIlcRxData callback function. Note: Disabling the event adversely hampers the ability of the driver to detect data arrival and should normally be left enabled
fifoThresh	UINT1	Rx FIFO Threshold crossed event: 0 = disable, 1 = enable. This event is reported via the cbackIlcRxData callback function. Note: Disabling the event adversely hampers the ability of the driver to detect data arrival and should normally be left enabled
fifoTimeout	UINT1	Rx FIFO data timeout event (detection of stale data in FIFO): 0 = disable, 1 = enable. This event is reported via the cbackIlcRxData callback function. Note: Disabling the event adversely hampers the ability of the driver to detect data arrival and should normally be left enabled
user0bitChg	UINT1	USER[0] header bit change event: 0 = disable, 1 = enable. This event is reported via the cbackIlcHead callback function.
linkbitsChg	UINT1	LINK[1:0] bits change event: 0 = disable, 1 = enable. This event is reported via the cbackIlcHead callback function.

Field Name	Field Type	Field Description
pg0bitChg	UINT1	PAGE[0] bit change event: 0 = disable, 1 = enable. This event is reported via the cbackIlcHead callback function.
pg1bitChg	UINT1	PAGE[1] bit change event: 0 = disable, 1 = enable. This event is reported via the cbackIlcHead callback function.

PRGM Block Event Mask (MASK_EVT_PRGM)

Table 16: Narrowband Chipset Event Mask for PRGM Block: sNBCS_MASK_EVT_PRGM

Field Name	Field Type	Field Description
prbsByteErr [NBCS_NUM_STS1PATH]	UINT1	PRBS byte error event for each of the STS-1 slice: 0 = disable, 1 = enable
prbsSync [NBCS_NUM_STS1PATH]	UINT1	PRBS synchronization event for each of the STS-1 slice: 0 = disable, 1 = enable

Status and Counts Structures

Status (STATUS)

This structure is used to retrieve a snapshot of the status information not processed by interrupts such as clock monitoring. This structure is used in the nbcsStatsGetStatus function calls

Table 17: Narrowband Chipset Status Block: sNBCS_STATUS

Field Name	Field Type	Field Description
handle	sNBCS_HNDL	Device handle
intf	sNBCS_STATUS_INTF	Status for the Interface/Clock Configuration block
stsw [NBCS_SBS_NUM_TSW]	sNBCS_STATUS_STSW	Status for the Space/Time Configuration block: stsw[0] is status for incoming time switch for SBS and space switch for NSE. stsw[1] is status for outgoing time switch for SBS and not used for NSE devices.

Field Name	Field Type	Field Description
lkc [NBCS_NSE_MAX_LINKS]	sNBCS_STATUS_LKC	Status for the LVDS link control block: lkc[0..1] are the statuses for the working and protection link in SBS. For NSE, lkc[] are the status for all the links. 12 in NSE-8G case and 32 in NSE-32G
prgm [NBCS_SBS_NUM_LINKS]	sNBCS_STATUS_PRGM	Status for the PRGM blocks in SBS devices. prgm[0] and prgm[1] are the status of the working and protection PRGM block respectively.

Interface/Clock Configuration Block Status (STATUS_INTF)

Table 18: Narrowband Chipset Status for Interface/Clock Configuration Block: sNBCS_STATUS_INTF

Field Name	Field Type	Field Description
csu1Lockv	UINT1	CSU#1 lock status: 0 = unlocked, 1 = locked
csu2Lockv	UINT1	CSU#2 lock status: 0 = unlocked, 1 = locked (NSE device only)
sysDll	sNBCS_STATUS_DLL	System DLL status block (SBS only)
refDll	sNBCS_STATUS_DLL	Reference DLL status block (SBS only)
sRefClka	UINT1	Reference clock signal: 0 = inactive, 1 = active
sysClka	UINT1	System clock signal: 0 = inactive, 1 = active
rc1fpa	UINT1	receive bus C1 frame pulse signal: 0 = inactive, 1 = active (NSE only)
rxBus	sNBCS_STATUS_SIGBUS	receive bus signal status block (SBS only)
inBus [NBCS_QUAD_BUS]	sNBCS_STATUS_SIGBUS	incoming quad bus signal status block (SBS only)

DLL Sub-Block Status (STATUS_DLL)

Table 19: Narrowband Chipset Status for DLL Sub-Block: sNBCS_STATUS_DLL

Field Name	Field Type	Field Description
run	UINT1	DLL lock status: 0 = unlocked, 1 = locked
error	UINT1	DLL delay line error: 0 = OK, 1 = error

Bus Signal Status (STATUS_SIGBUS)

Table 20: Narrowband Chipset Status for Bus Signal: sNBCS_STATUS_SIGBUS

Field Name	Field Type	Field Description
dataa	UINT1	RDATAA or IDATAA bus signal: 0 = inactive, 1 = active
pla	UINT1	RPLA or IPLA bus signal: 0 = inactive, 1 = active
v5a	UINT1	RV5A or IV5A bus signal: 0 = inactive, 1 = active
tpla	UINT1	RTPLA or ITPLA bus signal: 0 = inactive, 1 = active
c1fpa	UINT1	C1 frame pulse signal: 0 = inactive, 1 = active

Space/Time Switch Configuration Block Status (STATUS_STSW)

Table 21: Narrowband Chipset Status for Space/Time Switch Configuration Block: sNBCS_STATUS_STSW

Field Name	Field Type	Field Description
pgSwap	UINT1	connection page swap status: 0 = not pending, 1 = pending
pgUpdate	UINT1	connection page update status: 0 = complete, 1 = in progress

LVDS Link Controller Block Status (STATUS_LKC)

Table 22: Narrowband Chipset Status for LVDS Link Controller Block: sNBCS_STATUS_LKC

Field Name	Field Type	Field Description
oca	UINT1	out-of-character alignment status: 0 = aligned, 1 = mis-aligned
ofa	UINT1	out-of-frame alignment status: 0 = aligned, 1 = mis-aligned

PRGM Block Status (STATUS_PRGM)

Table 23: Narrowband Chipset Status for PRGM Block: sNBCS_STATUS_PRGM

Field Name	Field Type	Field Description
sync [NBCS_NUM_STS1PATH]	UINT1	byte sync status array for all STS-1 paths: 0 = sync, 1 = not sync

Device Counts (CNTR)

This structure is used to retrieve a snapshot of the various counts accumulated by the Narrowband Chipset device. This structure is used in the `nbcsStatsGetCounts` function.

Table 24: Narrowband Chipset Device Counts Block: sNBCS_CNTR

Field Name	Field Type	Field Description
handle	sNBCS_HNDL	Device handle
lcvCtr [NBCS_NSE_MAX_LINKS]	UINT2	Line code violation counter for the links. `lcvCtr[0]` and `lcvCtr[1]` are the counters for working and protection link respectively in SBS. For NSE devices, this array stores count for 12 and 32 links in NSE-8G and NSE-32G respectively.
prbsErrCtr [NBCS_SBS_NUM_LINKS] [NBCS_NUM_STS1PATH]	UINT2	PRBS byte error counter for the working and protection links and all 12 STS-1 paths. (SBS device only)

In-band Link Controller

In-band Link Message Header (HEADER_ILC)

Table 25: Narrowband Chipset In-band Link Message Header: sNBCS_HEADER_ILC

Field Name	Field Type	Field Description
userBits	UINT1	USER[2:0] header bits
pageBits	UINT1	PAGE[1:0] header bits
linkBits	UINT1	LINK[1:0] header bits
auxBits	UINT1	AUX[7:0] header bits

In-band Link Message Descriptor (MSG_DESC_ILC)

Table 26: Narrowband Chipset In-band Link Message Descriptor: sNBCS_MSG_DESC_ILC

Field Name	Field Type	Field Description
pBuf	UINT1*	Pointer to the data buffer
crcErr	UINT1	CRC error flag. 0 = normal, 1 = error

In-band Link Rx Buffer Descriptor (RXBUF_DESC_ILC)

Table 27: Narrowband Chipset In-band Link Message Descriptor: sNBCS_RXBUF_DESC_ILC

Field Name	Field Type	Field Description
linkDesc	UINT1	Link Descriptor: For SBS, 0 = working Tx, 1 = working Rx, 2 = protect Tx, 3 = protect For NSE, it is the physical port number. 0-31 for NSE20G and 0-11 for NSE8G
numMsgs	UINT1	Number of messages received
pMsgDesc	sNBCS_MSG_DESC_ILC*	Pointer to the received data buffer

In-band Link Tx Buffer Descriptor (TXBUF_DESC_ILC)

Table 28: Narrowband Chipset In-band Link Tx Buffer Descriptor: sNBCS_TXBUF_ILC

Field Name	Field Type	Field Description
linkDesc	UINT1	Link descriptor: For SBS, 0 = working Tx, 1 = working Rx, 2 = protect Tx, 3 = protect For NSE, it is the physical port number. 0-31 for NSE20G and 0-11 for NSE8G
pBuf	UINT1*	Pointer to the transmit data buffer
bufSz	UINT2	Data buffer size

In-band Link Configuration Structure (CFG_ILC)

Table 29: Narrowband Chipset In-band Link Configuration: sNBCS_CFG_ILC

Field Name	Field Type	Field Description
fifoThresh	UINT1	Hardware Rx FIFO threshold level: 0 – 7
fifoTimeout	eNBCS_ILC_FIFO_TIMEOUT	Hardware Rx FIFO timeout: NBCS_ILC_FIFO_125US : 125us NBCS_ILC_FIFO_250US : 250us NBCS_ILC_FIFO_375US : 375 us NBCS_ILC_FIFO_500US : 500 us

LVDS Link Controller

LVDS Link Configuration Structure (CFG_LKC)

Table 30: Narrowband Chipset LVDS Link Configuration: sNBCS_CFG_LKC

Field Name	Field Type	Field Description
rxInv	UINT1	Active polarity control (NSE links only): 0 = normal, 1 = complemented
tmode [NBCS_NUM_STS1PATH]	eNBCS_TMODE	Termination mode: NBCS_TMODE_MST, NBCS_TMODE_HPT and NBCS_TMODE_LPT

Field Name	Field Type	Field Description
swMode	eNBCS_LKC_SWITCHMODE	(This field is reserved for CSD use only. User does not have to set this and any value will be ignored) Link switch mode. This is to keep track of what frame boundary the port should set to. For DS0 CAS traffic, it should be a 48-mulitframe boundary.

Space/Time Switch Configuration

Map Setting Structure (CONMAP_STSW)

Table 31: Narrowband Chipset Space/Time Switch Map Setting: sNBCS_CONMAP_STSW

Field Name	Field Type	Field Description
devHndl	sNBCS_HNDL	device handle
devId	UINT1	device Identification
devType	eNBCS_DEVTYPE	device type
devNum1	UINT2	device number #1 supplied by user at the time when the device is added
devNum2	UINT2	device number #2 supplied by user at the time when the device is added. This field is zero if the device is SBS or SBSLITE. For NSE device, this denotes whether the device is a primary or secondary device (only applicable when in doubled SBS or doubled SBS/NSE fabric)
devNum3	UINT2	device number #3 supplied by user at the time when the device is added
accMode	eNBCS_ACCESSMODE_STSW	access mode
numSetting	UINT4	number of settings
pBuf	void*	pointer to data buffer
pBuf2	void*	pointer to data buffer 2

Field Name	Field Type	Field Description
pBuf3	void*	pointer to data buffer3

Pseudo Random Bit Sequence Generator/Monitor Configuration

PRGM Configuration Structure (CFG_PRGM)

Table 32: Narrowband Chipset PRGM Configuration: sNBCS_CFG_PRGM

Field Name	Field Type	Field Description
seqPrbs	UINT1	Pattern control: 0 = prbs, 1 = sequential
invPrbs	UINT1	Inversion control: 0 = disable, 1 = enable
lfsr	UINT4	Linear feedback shift register seed value
amode	UINT1	Autonomous mode: 0 = disable, 1 = enable

PRGM Payload Configuration Structure (CFG_PRGM_PYLD)

Table 33: Narrowband Chipset PRGM Payload Configuration: sNBCS_CFG_PRGM_PYLD

Field Name	Field Type	Field Description
sts12c	UINT1	STS-12c mode: 0 = disable, 1 = enable
sts3c [NBCS_NUM_STS3]	UINT1	STS-3c mode: 0 = disable, 1 = enable

Interface/Clock Configuration

CSU/DLL Configuration Structure (CFG_INTF_CSU)

Table 34: Narrowband Chipset CSU/DLL Configuration: sNBCS_CFG_INTF_CSU

Field Name	Field Type	Field Description
csuReset	UINT1	CSU soft reset: 0 = reset, 1 = normal
csuMode	UINT1	CSU operating mode: 0 = normal, 1 = low power
csu2Reset	UINT1	CSU#2 soft reset: 0 = reset, 1 = normal (NSE device only)

Field Name	Field Type	Field Description
csu2Mode	UINT1	CSU#2 operating mode: 0 = normal, 1 = low power (NSE device only)
sysDllIgnorePhase	UINT1	System DLL phase track: 0 = track, 1 = ignore
refDllIgnorePhase	UINT1	Reference DLL phase track: 0 = track, 1 = ignore

Interface Bus Configuration Structure (CFG_INTF_BUS)

Table 35: Narrowband Chipset Interface Bus Configuration: sNBCS_CFG_INTF_BUS

Field Name	Field Type	Field Description
busMode	sNBCS_CFG_BUSMODE	Bus mode configuration structure
inBusCfgParam	sNBCS_CFG_BUSPARAM	Incoming bus configuration parameter structure
outBusCfgParam	sNBCS_CFG_BUSPARAM	Outgoing bus configuration parameter structure
txBusCfgParam	sNBCS_CFG_BUSPARAM	Transmit serial bus configuration parameter structure
rxBusCfgParam	sNBCS_CFG_BUSPARAM	Receive serial bus configuration parameter structure

Interface Bus Mode Configuration Structure (CFG_BUSMODE)

Table 36: Narrowband Chipset Interface Bus Mode Configuration: sNBCS_CFG_BUSMODE

Field Name	Field Type	Field Description
io	eNBCS_IO_BUSMODE	Bus mode: NBCS_IO_BUS_QUAD = 4 x 19.44 MHz bus, NBCS_IO_BUS_SINGLE = 1 x 77.76 MHz bus **NOTE**: NBCS_IO_BUS_QUAD is *not* supported in SBSLITE devices
bridge	UINT1	Bridge mode: 0 = serial LVDS in SBS enabled, 1 = serial LVDS disabled and parallel bus I/O is enabled. This field is ignored in SBSLITE devices.

Field Name	Field Type	Field Description
multiFrm	eNBCS_MULTIFRM_MODE	Multi-frame mode: NBCS_MF_4 = 4 frames in multi-frame, NBCS_MF_48 = 48 frames in multi-frame
phyDevice	UINT1	SBI physical/link layer device mode: 0 = link layer device, 1 = physical layer device

Interface SBI/TeleCombus Configuration Parameter Structure (CFG_BUSPARAM)

Table 37: Narrowband Chipset Interface SBI/TeleCombus Configuration Parameter: sNBCS_CFG_BUSPARAM

Field Name	Field Type	Field Description
oddParity [NBCS_QUAD_BUS]	UINT1	Bus parity selection: 0 = even, 1 = odd. Note: The second, third, and fourth elements in the array are only applicable in the case of quad incoming and outgoing bus configuration
incPl [NBCS_QUAD_BUS]	UINT1	PL signal parity inclusion: 0 = no, 1 = yes Note: The second, third, and fourth elements in the array are only applicable in the case of quad incoming and outgoing bus configuration (For TeleCombus only)
incC1fp [NBCS_QUAD_BUS]	UINT1	C1FP signal parity inclusion: 0 = no, 1 = yes Note: The second, third, and fourth elements in the array are only applicable in the case of quad incoming and outgoing bus configuration (For TeleCombus only)
j1LockPos	UINT1	J1 byte position lock: 0 = lock at offset 522 (byte after C1), 1 = lock at offset 0 (For TeleCombus only)
j1Cfg	UINT2	J1 byte identification inclusion. j1Cfg[12:0] is a 12-bit bitmask for the 12 STS-1 signals that controls whether the J1 byte position is pulsed high in the bus signal C1FP: 0 = does not pulse high, 1 = pulse high (For TeleCombus only)
v1Cfg	UINT2	V1 byte identification inclusion. v1Cfg[12:0] is a 12-bit mask for the 12 STS-1 signals that controls whether the V1 byte position is pulsed high in the bus signal C1FP: 0 = does not pulse high, 1 = pulse high (For TeleCombus only)

Field Name	Field Type	Field Description
h1h2Ena	UINT1	H1 and H2 values output enable: 0 = disable, 1 = enable (For TeleCombus only)
h1h2PtrSel	UINT2	Alternate H1-H2 pointer selection. h1h2PtrSel[12:0] is a 12-bit bitmask for the 12 STS-1 signals that controls whether the H1/2 or the alternate H1/2 value is the output: 0 = H1-H2 value, 1 = alternate H1-H2 value (For TeleCombus only)
h1PtrVal	UINT1	H1 value to be output when the field h1h2Ena is logic high. This field is applicable only for transmit and outgoing TeleCombus (For TeleCombus only)
h2PtrVal	UINT1	H2 value to be output when the field h1h2Ena is logic high. This field is applicable only for transmit and outgoing TeleCombus (For TeleCombus only)
altH1Val	UINT1	Alternate H1 value to be output when the field h1h2Ena is logic high. This field is applicable only for transmit and outgoing TeleCombus (For TeleCombus only)
altH2Val	UINT1	Alternate H2 value to be output when the field h1h2Ena is logic high. This field is applicable only for transmit and outgoing TeleCombus (For TeleCombus only)

TeleCombus Payload Configuration Structure (CFG_PYLD_TCB)

Table 38: Narrowband Chipset Fabric Management TeleCombus Payload Configuration: sNBCS_CFG_PYLD_TCB

Field Name	Field Type	Field Description
vtgpPyld [NBCS_NUM_STS3+1] [NBCS_NUM_STS1+1] [NBCS_NUM_VTGROUP+1]	eNBCS_TCBTRIB_TYPE	Payload type for the VT group: NBCS_TCBVT_VT15 = VT1.5, NBCS_TCBVT_VT2 = VT2, NBCS_TCBVT_VT3 = VT3, NBCS_TCBVT_VT6 = VT6
sdhAu4Frm [NBCS_NUM_STM1+1]	UINT1	SDH AU-4 frame indicator: A logic one indicates the STM-1 frame is AU-4 structured; otherwise set to logic zero for SDH AU-3 structured frame or SONET

SBI Bus Payload Configuration Structure (CFG_PYLD_SBI)

Table 39: Narrowband Chipset Fabric Management SBI Bus Payload Configuration: sNBCS_CFG_PYLD_SBI

Field Name	Field Type	Field Description
`spe [NBCS_MAX_SBI+1] [NBCS_MAX_SBI_SPE+1]`	`eSBS_SBITRIB_TYPE`	Payload type for the SPE: NBCS_T1_PYLD: T1 or TVT1.5 NBCS_E1_PYLD: E1 or TVT2 NBCS_DS3_E3_PYLD: DS3 or E3 NBCS_FRAC_RT_PYLD: fractional rate

SBI Bus Virtual Tributary Configuration Structure (CFG_TRIB_SBI)

Table 40: Narrowband Chipset Fabric Management SBI Virtual Tributaries Configuration Structure: sNBCS_CFG_TRIB_SBI

Field Name	Field Type	Field Description
`oe`	`UINT1`	Tributary output enable: 0 = disable,1 = enable
`casEna`	`UINT1`	CAS processing enable: 0 = disable, 1 = enable
`justReqEna`	`UINT1`	Justification request enable: 0 = disable, 1 = enable
`tvtEna`	`UINT1`	Transparent VT enable: 0 = disable, 1 = enable

Fabric Management Module

SBI/TeleCombus Time Slot Structure (SLOT)

Table 41: Narrowband Chipset Fabric Management Timeslot Structure: sNBCS_SLOT

Field Name	Field Type	Field Description
`handle`	`sNBCS_HNDL`	SBS device handle

Field Name	Field Type	Field Description
sbi	sNBCS_TRIB_SBI	SBI bus tributary structure. This field is a union member of the following field, tcb. The union name is bus. In other words, the following syntax in C is used to assess this member: "bus.sbi"
tcb	sNBCS_TRIB_TCB	TeleCombus tributary structure. This field is a union member of the above field, sbi. The union name is bus. In other words, the following syntax in C is used to assess this member: "bus.tcb"
ts	UINT2	Timeslot number

TeleCombus Virtual Tributary Structure (TRIB_TCB)

Table 42: Narrowband Chipset Fabric Management TeleCombus Virtual Tributaries Structure: sNBCS_TRIB_TCB

Field Name	Field Type	Field Description
sts3Num	UINT1	STS-3 number: 1-4
sts1Num	UINT1	STS-1 number: 1-3. A zero selects the entire STS-3/3c indicated by sts3Num.
vtgp	UINT1	Virtual tributary group number: 1-7; A zero selects the entire STS-1 indicated by (sts3Num,sts1Num)
trib	UINT1	Tributary number, VT1.5/VC11: 1-4, VT2/VC12: 1-3, VT3: 1-2, VT6/VC2: n/a.

SBI Bus Virtual Tributary Structure (TRIB_SBI)

Table 43: Narrowband Chipset Fabric Management SBI Bus Virtual Tributaries Structure: sNBCS_TRIB_SBI

Field Name	Field Type	Field Description
sbiNum	UINT1	SBI bus number: 1-4

Field Name	Field Type	Field Description
speNum	UINT1	SPE number: 1-3
trib	UINT1	Tributary number, T1/TVT1.5/TU11: 1-28, E1/TVT2/TU12: 1-21, DS3/E3: n/a

Fabric Edge Wiring Structure (EDGE_WIRING)

Table 44: Narrowband Chipset Fabric Management Edge Wiring: sNBCS_EDGE_WIRING

Field Name	Field Type	Field Description
nsePhyPortNum	UINT2	NSE device physical port number
sbsNum	UINT2	SBS user number

Device Diagnostics Structures (DIAG_TEST)

Register Test Structure

This structure contains the parameters required by the driver to perform a register test on a Narrowband Chipset device. The user passes this structure as an input parameter in the nbcsDiagTestReg function call.

Table 45: Narrowband Chipset RAM Test Structure: sNBCS_DIAG_TEST_REG

Field Name	Field Type	Field Description
type	UINT1	type of register test: 0x01 = test the full range of registers 0x02 = read/write test 0x04 = walking ones test
offset	UINT2	register offset
bitmask	UINT4	read/write bitmask
value	UINT4	value to write and read back

RAM Test Structure

This structure contains the parameters required by the driver to perform a RAM test on a Narrowband Chipset device. The user passes this structure as an input parameter in the nbcsDiagTestRam function call.

Table 46: Narrowband Chipset RAM Test Structure: sNBCS_DIAG_TEST_RAM

Field Name	Field Type	Field Description
type	UINT1	type of RAM test: 0x01 = full-range test 0x02 = read/write test 0x04 = walking ones test 0x08 = aliasing test
ramType	UINT1	type of RAM (n/a to NSE): 0x00 = time switch RAM test in incoming dir 0x01 = time switch RAM test in outgoing dir
startOffset	UINT2	starting RAM offset
endOffset	UINT2	ending RAM offset
pValue	void*	pointer to (array of) value to write and read back. For SBS, this should point to a UINT2 value. For NSE, this should point to an array of UINT1 with 32 or 12 elements for NSE-20G and NSE-8G respectively.

4.3 Structures in the Driver's Allocated Memory

These structures are defined and used by the driver and are part of the context memory allocated when the driver is opened. These structures are the Chipset Module Data Block (CSMDB) and the Chipset Device Data Block (CSDDB).

Chipset Module Data Block: CSMDB

The CSMDB is the top-level structure for the module. It contains configuration data about the Module level code and pointers to configuration data about the device level codes.

- `errModule` indicates specific error codes returned by API functions that are not passed directly to the application. Most of the module API functions return a specific error code directly. When the returned code is `NBCS_FAILURE`, this indicates that the top-level function was not able to carry the specified error code back to the application. Under those circumstances, the proper error code is recorded in this element. The element is the first in the structure so that the user can cast the CSMDB pointer into an INT4 pointer and retrieve the local error (this eliminates the need to include the CSMDB template into the application code).
- `valid` indicates that this structure has been properly initialized and can be read by the user.
- `stateDevice` contains the current state of the device and could be set to: `NBCS_START`, `NBCS_PRESENT`, `NBCS_ACTIVE` or `NBCS_INACTIVE`.

- stateModule contains the current state of the module and could be set to: NBCS_MOD_START, NBCS_MOD_IDLE or NBCS_MOD_READY.
- usrCtxt is a value that can be used by the application to identify the device during the execution of the callback functions. It is passed to the chipset driver when nbcsAdd is called and returned to the user in callback functions.

Table 47: Narrowband Chipset Module Data Block: sNBCS_CSMDB

Field Name	Field Type	Field Description
errModule	INT4	Global error Indicator for module calls
valid	UINT2	Indicates that this structure has been initialized
stateModule	eNBCS_MOD_STATE	Module state; can be one of the following NBCS_MOD_START, NBCS_MOD_IDLE or NBCS_MOD_READY
stateChipset	eNBCS_DEV_STATE	Chipset state; can be one of the following: NBCS_START, NBCS_PRESENT, NBCS_INACTIVE, or NBCS_ACTIVE
totalMemSz	UINT4	Total size of memory allocated by the chipset driver
chkPtType	eNBCS_CHKPT_TYPE	Checkpoint type
chkPtState	UINT2	The state of the checkpointing operation
pollMode	UINT1	Polling mode flag: 0 = disabled (interrupt mode), 1 = enabled
cbackC1FP	NBCS_CBACK_TC	Callback function for C1 frame pulse reception
cbackIlcRxData	NBCS_CBACK_TC	Callback function for in-band link Rx data
cbackIlcHead	NBCS_CBACK	Callback function for in-band link header bits change
cbackIntf	NBCS_CBACK	Callback function for Interface/clock block events
cbackLkc	NBCS_CBACK	Callback function for LVDS link controller block events

Field Name	Field Type	Field Description
cbackStsw	NBCS_CBACK	Callback function for Space/time Configuration block events
cbackPrgm	NBCS_CBACK	Callback function for PRGM block events
fabType	eNBCS_FABRIC_TYPE	System Fabric Type: NBCS_FABRIC_STD: standard NSE/SBS fabric NBCS_FABRIC_DOUBLE_SBS: double SBS fabric. SBS devices are doubled up but NSE device(s) are not. . NBCS_FABRIC_DOUBLE_NSESBS: double SBS and NSE fabric. Both SBS and NSE devices are doubled up in the fabric.
sysBusType	eNBCS_BUSTYPE	Bus Type: NBCS_BUS_SBI for SBI Bus or NBCS_BUS_TCB for TeleCombus applications
swhMode	eNBCS_SWHMODE	Switching mode (n/a in TeleCombus mode): NBCS_SWH_BYTE or NBCS_SWH_COLUMN.
casMuxMode	UINT1	CAS processing mode. This field is high when the CSD is in DS0 CAS switching mode. In TeleCombus mode, this indicates which bus signal, OPL, or OTAIS is used as external MUX control signal
pageSwapCntl	eNBCS_CONMAP_CNTL	Source of control for the connection page switching in all SBSs: NBCS_MAP_CNTL_SW: software controls the page switching NBCS_MAP_CNTL_HW: hardware pin controls the page switching NBCS_MAP_CNTL_ILC: the PAGE bits in ILC controls the page switching
wpLinkCntl	eNBCS_WPLINK_CNTL	Source of control for the working and protection LVDS link in all SBSs: NBCS_LINK_CNTL_SW: software controls whether working or protection link is active NBCS_LINK_CNTL_HW: a hardware pin controls whether working or protection link is active

Field Name	Field Type	Field Description
pageAutoSync	UINT1	Automatic active page to inactive connection page synchronization. When this field is logic one, the settings are copied from the active to inactive page after a page switch in all SBS and NSE devices in the system.
coreDepth	UINT2	The depth of the NSE switching core
coreNumStage	UINT2	The number of stages of the NSE switching core
numPortNse	UINT2	number of physical ports for the NSE that makes up the switching core. It should be 32 or 12 for NSE-20G and NSE-8G devices respectively
pageAutoUpdate	UINT1	NSE/SBS devices connection setting automatic update: 0 = CSD does not automatically update the map settings of the local devices; 1 = CSD automatically updates the map settings after a call request
nseDrv	sNBCS_DRV_NSE	NSE device driver database structure
sbsDrv	sNBCS_DRV_SBS	SBS device driver database structure
opaLib	sNBCS_LIB_OPA	OPA library database structure
maxGroups	UINT2	Maximum number of groups supported during this session
maxSbsDevs	UINT2	Maximum number of SBS devices in this session
maxNseDevs	UINT2	Maximum number of NSE devices in this session
numGroups	UINT2	Number of groups currently defined
pGdb	sNBCS_GDB *	(array of) Group Data Block (GDB) in context memory
maxSbsInitProfs	UINT2	Maximum number of SBS initialization profiles supported

Field Name	Field Type	Field Description
maxNseInitProfs	UINT2	Maximum number of NSE initialization profiles supported
pSbsInitProfs	sNBCS_DIV_SBS *	(array of) SBS initialization profiles
pNseInitProfs	sNBCS_DIV_NSE *	(array of) NSE initialization profiles
nseInitProfOffset	UINT2	Initialization profile offset for the NSE devices
inToggling	UINT1	Page Toggling. This flag is logic one if the toggle page operation is in progress

Group Data Block: GDB

The GDB is the top-level structure for each group. It contains configuration data describing the devices in the group.

- `type:` used to identify the group type. This has to be the first element in the structure because this field is used by the CSD to resolve the group type.
- `errGroup`: Most of the module API functions return a specific error code directly. When the returned code is `NBCS_FAILURE`, this indicates that the top-level function was not able to carry the specified error code back to the application. Under those circumstances, the proper error code is recorded in this element.
- `valid` indicates that this structure has been properly initialized and can be read by the user.
- `numSbs` indicates the total number of SBS devices in the group.
- `numNse` indicates the total number of NSE devices in the group.
- `ppSbs` is the array of SBS chipset device data block, which holds information pertaining the SBS device in the group.
- `ppNse` is the array of NSE chipset device data block, which holds information pertaining the NSE device in the group.

Table 48: Narrowband Chipset Group Data Block: sNBCS_GDB

Field Name	Field Type	Field Description
type	eNBCS_DEVTYPE	Group type
valid	UINT2	Indicates that this structure has been initialized
errGroup	INT4*	Global error Indicator for module calls

Field Name	Field Type	Field Description
numSbs	UINT2	Number of SBS devices in the group
numNse	UINT2	Number of NSE devices in the group
ppSbs	sNBCS_CSDDB_SBS **	(array of) SBS CSDDB
ppNse	sNBCS_CSDDB_NSE **	(array of) NSE CSDDB

Device Driver Database Block: DRV_SBS, DRV_NSE

Table 49: Narrowband Chipset Device Driver Database Block: sNBCS_DRV_SBS, sNBCS_DRV_NSE

Field Name	Field Type	Field Description
pModErr	INT4*	Pointer to the global error Indicator for module calls
maxDevs	UINT2	The maximum number of devices supported
numDevs	UINT2	The current number of devices registered
phyDevsPresent	UINT1	Indicates whether the physical device driver is present locally
dev	void*	For SBS devices, it is an array of sNBCS_CSDDB_SBS. For NSE devices, it is an array of sNBCS_CSDDB_NSE.

OPA Library Database Block: LIB_OPA

Table 50: Narrowband Chipset OPA Library Database Block: sNBCS_LIB_OPA

Field Name	Field Type	Field Description
edgeTblOffset	UINT2	Offset of the edge wiring table
coreTblOffset	UINT2	Offset of the core wiring table
opaUse	UINT1	Indicates if OPA routing is executed locally

Field Name	Field Type	Field Description
hFabric	void*	Handle of the fabric used in OPA library
nseLogic2CsddbLut	sNBCS_HNDL*	Lookup table for converting NSE logical number to CSDDB
igrsEdgeWireTbl	sNBCS_HNDL*	Lookup table for converting SBS logical number to CSDDB on ingress side
egrsEdgeWireTbl	sNBCS_HNDL*	Lookup table for converting SBS logical number to CSDDB on egress side
stdPhyWiring	UINT1	A logic one indicates the underlying wiring topology is “PMC-standard” compliant
devSettingHdr [NBCS_MAX_SETTING_HEADER]	sNBCS_DEV_SETTINGS	Internal buffer for storing setting headers retrieved from OPA
numDevSetting	UINT2	Number of device settings
devSettingIdx	UINT2	index for the device setting
isPrevSecDevSetting	UINT1	A logic one indicates that the setting just retrieved is for the secondary device in a doubled fabric configuration
prevSettingReq	eNBCS_FABRIC_SETTING	Previous setting request. This field keeps track of the setting type from previous operation.
defSpeType	UINT1	SPE payload type (This field is applicable only when using SBS rev A devices)

Device Settings Header: DEV_SETTINGS

Table 51: Narrowband Chipset Device Setting Header: sNBCS_DEV_SETTINGS

Field Name	Field Type	Field Description
settingType	INT1	setting type
devDirPort	INT1	device direction, ingress or egress
devId	UINT2	device ID
devSubId	UINT2	device sub ID
numSettings	UINT2	number of settings
pbuf1	void *	pointer to buffer 1
pbuf2	void *	pointer to buffer 2
pbuf3	void *	pointer to buffer 3

SBS Chipset Device Data Block: CSDDB_SBS

Table 52: Narrowband Chipset SBS Device Data Block: sNBCS_CSDDB_SBS

Field Name	Field Type	Field Description
type	eNBCS_DEVTYPE	Device type **Note**: This has to be the first element in the structure because the CSD uses this to resolve the actual type of the device.
valid	UINT2	Indicates that this structure has been initialized
pDevErr	INT4*	Pointer to the global device error module
hndl	sNBCS_HNDL	Handle of the device. Used when calling underlying device drivers API
usrContext	sNBCS_USR_CTXT	Stores the user's context for the device. It is passed as an input parameter when the driver invokes an application callback

Field Name	Field Type	Field Description
baseAddr	void *	Base address of the device
isLocal	UINT1	A logic one indicates the SBS device is local.
state	eNBCS_DEV_STATE	Device state
devIdParm	sNBCS_DEV_ID_PARM	Device ID parameters. A structure that holds information including device numbers and handle of the device
numGroups	UINT2	Total number of groups this device belongs to
userNum	UINT2	The user number of the SBS (specified by user when the device is added)
logicNum	INT4	This is the logical SBS number recognized by OPA
secNse	UINT2	This is logic one if this SBS is connected to the secondary NSE in a doubled SBS/NSE fabric configuration
igrsNseLogicNum	UINT2	Logical number of the NSE the ingress side (of this SBS) is connected to
igrsNsePhyPortNum	UINT1	Physical port number of the NSE (indicated by `nseIgrsLogicalNum`) the ingress side (of this SBS) is connected to
egrsNseLogicNum	UINT2	Logical number of the NSE the egress side (of this SBS) is connected to
egrsNsePhyPortNum	UINT1	Physical port number of the NSE (indicated by `nseIgrsLogicNum`) the egress side (of this SBS) is connected to

Field Name	Field Type	Field Description
portProtected	UINT2	Port protection indicator: This field is non-zero if the SBS is involved in some form of port protection (1+1/1:N). Furthermore, this field keeps track of number of working SBS this SBS is protecting if the SBS is a protect one in the 1:N protection
protectId	INT4	protection ID used in OPA library
sbsLink	sNBCS_CSDDB_SBS*	CSDDB pointer to the other SBS in a 1+1 protection
protectMode	eNBCS_PORTPROTECT	This field indicates what port protection mode the SBS is engaged in.
busCfg	sNBCS_CFG_INTF_BUS	Bus configuration information
sbiPyld	sNBCS_CFG_PYLD_SBI	SBI Bus payload type
tcbPyld	sNBCS_CFG_PYLD_TCB	TeleCombus payload type
sbiTribCfg [NBCS_NUM_STS3+1] [NBCS_NUM_STS1+1] [NBCS_MAX_T1_TRIB+1]	sNBCS_CFG_TRIB_SBI	SBI Tributaries configuration
egrsSpeIntegrity [NBCS_NUM_STS3+1] [NBCS_NUM_STS1+1]	UINT1	Egress SPE integrity. This is non-zero if the integrity is present
egrsTribIntegrity [NBCS_NUM_STS3+1] [NBCS_NUM_STS1+1] [NBCS_MAX_T1_TRIB+1]	UINT1	Egress tributary integrity. This is non-zero if the integrity is present
igrsPendingPage	UINT1	page number of the pending active page in the ingress direction
egrsPendingPage	UINT1	page number of the pending active page in the egress direction
egrsMuxCtlVal	UINT4	Multiplexer control value (in the case of doubled SBS or doubled SBS/NSE fabrics) used by the CSD for this SBS device.

NSE Chipset Device Data Block: CSDDB_NSE

Table 53: Narrowband Chipset NSE Device Data Block: sNBCS_CSDDB_NSE

Field Name	Field Type	Field Description
type	eNBCS_DEVTYPE	Device type **Note**: This has to be the first element in the structure because the CSD uses this to resolve the actual type of the device.
valid	UINT2	Indicates that this structure has been initialized
pDevErr	INT4*	Pointer to the global device error module
hndl	sNBCS_HNDL	Handle of the device. Used when calling underlying device drivers API
usrContext	sNBCS_USR_CTXT	Stores the user's context for the device. It is passed as an input parameter when the driver invokes an application callback
baseAddr	void *	Base address of the device
isLocal	UINT1	This field indicates whether the NSE device is local. 0 = remote, 1 = local
state	eNBCS_DEV_STATE	Device state
devIdParm	sNBCS_DEV_ID_PA RM	Device ID parameters. A structure that holds information including device numbers and handle of the device
numGroups	UINT2	Total number of groups this device belongs to
mapAutoUpdate	UINT1	Flag for connection map automatic update
secNse	UINT2	This field is logic high if the NSE device is the secondary NSE in a doubled SBS/NSE fabric configuration
logicalNum	UINT2	Logical number of the NSE
igrsPhyPortLut [NBCS_NSE_MAX_LINKS]	UINT1	Lookup table to convert logical port number to physical (ingress) port number of the NSE

Field Name	Field Type	Field Description
egrsPhyPortLut [NBCS_NSE_MAX_LINKS]	UINT1	Lookup table to convert logical port number to physical (egress) port number of the NSE
pendingPage	UINT1	Number of the pending active page
pendingIgrsSbsPg	UINT4	All 32 pending page number of the ingress SBS attached to this NSE
pendingEgrsSbsPg	UINT4	All 32 pending page number of the egress SBS attached to this NSE
devCfg	sNBCS_DIV_NSE	Copy of the current NSE device setting

Device Identification Parameter Block: DEV_ID_PARM

The following structure contains chipset device information block. The following is a description of the fields in the structures:

Table 54: Narrowband Chipset Device Information Block: sNBCS_DEV_ID_PARM

Field Name	Field Type	Field Description
devHandle	void*	Device handle assigned by the SBS/NSE device driver.
devNum1	UINT1	First device number assigned by the user. It is supplied by the user when the device is added via the structure sNBCS_DEVINFO.
devNum2	UINT1	Second device number assigned by the user. It is supplied by the user when the device is added via the structure sNBCS_DEVINFO.
devNum3	UINT1	Third device number assigned by the user. It is supplied by the user when the device is added via the structure sNBCS_DEVINFO.

Generic Device/Group Handle: HANDLE

Table 55: Narrowband Chipset Generic Device/Group Handle: uNBCS_HANDLE

Field Name	Field Type	Field Description
devCsddb	sNBCS_HNDL	device handle
nseCsddb	sNBCS_CSDDB_NSE*	NSE CSDDB
sbsCsddb	sNBCS_CSDDB_SBS*	SBS CSDDB
grpGdb	sNBCS_GDB*	Group GDB

4.4 Structures Passed through RTOS Buffers

Deferred Processing Vector: DPV

This structure is used in two ways. First, it is used to determine the size of buffer required by the RTOS for use in the driver. Second, it defines the format of the data that is assembled by the chipset driver and sent to the application code. It is the application's responsibility to create one pool of DPV buffers when the driver calls the user-supplied `sysNbcsBufferStart` function.

Note: the application code is responsible for returning this buffer to the RTOS buffer pool. `sysNbcsDPVBufferRtn` can be used to return a buffer to either pool.

The DPR reports events to the application using user-defined callbacks. The DPR uses each callback to report a functionally related group of events.

Table 56: Narrowband Chipset Deferred Processing Vector: sNBCS_DPV

Field Name	Field Type	Field Description
event	NBCS_EVENT	Bitmap indicating event(s) being reported.
info	UINT4	Event related information

4.5 Global Variables

Although most of the variables within the driver are not meant to be used by the application code, there is one global variable that can be of great use to the application code.

This variable is called `nbcsMdb` and acts as a global pointer to the Chipset Module Data Block (CSMDB). The content of this global variable should be considered *read-only* by the application.

- errModule: This structure element is used to store an error code that specifies the reason for an API function's failure. The field is only valid for functions that do not return an error code or when a value of NBCS_FAILURE is returned.
- stateModule: This structure element is used to store the module state (Figure 18).
- stateChipset: This structure element denotes the state of the chipset driver (Figure 18).

5 APPLICATION PROGRAMMING INTERFACE

This section of the manual provides a detailed description of each function that is a member of the Narrowband Chipset driver Application Programming Interface (API). API functions typically execute in the context of an application task.

It is important to note that these functions are not re-entrant. This means that two application tasks cannot invoke the same API at the same time. However the driver protects its data structures from concurrent accesses by the application.

5.1 Module, Device and Group Management

Module management can be accomplished through the use of a set of API functions that are used by the Application to open, start, stop and close the driver module. These functions take care of initializing the driver, allocating memory and requesting all RTOS resources needed by the driver. They are also used to change the module state. For more information on the module states see the state diagram on page 48. For a typical module management flow diagram see page 50.

Group management consists of a set of API functions that are used by the Application to define various groups of devices. The use of this grouping is optional; if future API calls are to be made at the "group" level of abstraction, however, the groups must be defined using the functions in this section. A group is considered to be in the same state as its constituent members, if all the members are in the same state. A group is considered to be in an indeterminate state if its constituent devices are not all in the same state.

Device management is performed by the use of a set of API functions to control the devices. These functions take care of initializing a device in a specific configuration, and enabling the device general activity. They are also used to change the software state for that device. For more information on the device states see the state diagram on page 48. For a typical device management flow diagram see page 51.

Some management function can act on either a single device or on a group. These functions distinguish whether the operation is intended for a device or group by examining the handle given by the user. For instance, `nbcsActivate` can operate on an individual device or a group.

Note that if group management functions are used, device management functions performed on devices within groups should be used carefully, as the use of these functions will often cause the group state to become `NBCS_INDETERMINATE`.

For more information on the module and device states see the state diagram on page 48. For typical module and device management flow diagrams see pages 50 and 51 respectively.

Opening the Driver Module: nbcsModuleOpen

This function performs module level initialization of the chipset device driver. This involves allocating all of the memory needed by the driver and initializing the internal structures.

Prototype `INT4 nbcsModuleOpen(sNBCS_MIV *pMiv)`

Inputs `pMiv` : (pointer to) Module Initialization Vector

Outputs Places the address of `errorModule` into the MIV passed by the application

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_MODULE_STATE`
`NBCS_ERR_INVALID_MIV`
`NBCS_ERR_MEM_ALLOC`

Valid States `NBCS_MOD_START`

Side Effects Changes the MODULE state `to NBCS_MOD_IDLE`

Closing the Driver Module: nbcsModuleClose

This function performs module level shutdown of the driver. This involves deleting all devices being controlled by the driver (by calling `nbcsDelete` for each device) and de-allocating all the memory allocated by the driver.

Prototype `INT4 nbcsModuleClose(void)`

Inputs None

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_MODULE_STATE`

Valid States `NBCS_MOD_IDLE, NBCS_MOD_READY`

Side Effects Changes the MODULE state to `NBCS_MOD_START`

Starting the Driver Module: nbcsModuleStart

This function connects the RTOS resources to the chipset driver. This involves allocating semaphores and timers, and initializing buffers. Upon successful return from this function, the driver is ready to add devices.

Prototype `INT4 nbcsModuleStart(void)`

Inputs None

Outputs None

Returns Success = `NBCS_SUCCESS`

Failure = `NBCS_ERR_INVALID_MODULE_STATE`
`NBCS_ERR_INT_INSTALL`
`NBCS_ERR_BUF_START`

Valid States `NBCS_MOD_IDLE`

Side Effects Changes the MODULE state to `NBCS_MOD_READY`

Stopping the Driver Module: nbcsModuleStop

This function disconnects the RTOS resources from the chipset driver. This involves de-allocating semaphores and timers, and freeing-up buffers. If there are any registered devices, `nbcsDelete` is called for each.

Prototype `INT4 nbcsModuleStop(void)`

Inputs None

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_MODULE_STATE`

Valid States `NBCS_MOD_READY`

Side Effects Changes the MODULE state to `NBCS_MOD_IDLE`

Adding a Device: nbcsAdd

This function verifies the presence of a new device in the hardware and then returns a handle back to the user. The device handle is passed as a parameter of most of the device API functions. It is used by the driver to identify the device on which the operation is to be performed.

Prototype `sNBCS_HNDL nbcsAdd(sNBCS_DEVINFO* pDevInfo, sNBCS_USR_CTXT usrCtxt, INT4 **pperrDevice)`

Inputs
`pDevInfo` : pointer to the information structure of the device to be added
`usrCtxt` : user context for this device
`pperrDevice` : (pointer to) an area of memory

Outputs ERROR code written to the CSMDB on failure
`NBCS_ERR_INVALID_MODULE_STATE`
`NBCS_ERR_INVALID_ARG`
`NBCS_ERR_DEVS_FULL`
`NBCS_ERR_DEV_ALREADY_ADDED`
`NBCS_ERR_INVALID_DEV`

`pperrDevice` : (pointer to) `errDevice` (inside the

CSDDB)

Returns Success = Device Handle (to be used as an argument to most of the Narrowband Chipset APIs)
Failure = NULL (pointer)

Valid States `NBCS_MOD_READY`

Side Effects Changes the DEVICE state to `NBCS_PRESENT`

Defining a Group or Adding Devices to a Group: nbcsGroupAdd

This function handles the following: (a) define a new group and add new device(s) to it; (b) define a new group and add existing device(s) to it; (c) add new devices to an existing group; or (d) add existing devices to an existing group. The group handle is passed as a parameter of most of the API functions operating in the group context. It is used by the driver to identify the group on which the operation is to be performed.

In case (a), content of `pGroupHndl` should be NULL, `pDevInfo` should be pointed to an array of element `numDev` with device information data structures. The handle of the new group will be stored in the location pointed to by `pGroupHndl`. The handles of all the new devices added will also be stored in the location pointed to by `pDevHandle`.

In case (b), content of `pGroupHndl` should be NULL, `pDevInfo` and `pperrDevice` should be NULL, and `pDevHandle` should be pointed to an array of device handles with `numDev` elements in it.

In case (c), `pGroupHndl` should be pointed to a valid group handle, and `pDevInfo` should be pointed to an array of element `numDev` with device information data structures. The handles of all the new devices added will also be stored in the location pointed to by `pDevHandle`.

In case (d), `pGroupHndl` should be pointed to a valid group handle, and `pDevHandle` should be pointed to an array of device handles with `numDev` elements in it.

In all cases, it is user's responsibility to make sure that the buffer is large enough to store returned values from the function.

Prototype

```
INT4 nbcsGroupAdd(sNBCS_HNDL* pGroupHandle,
sNBCS_DEVINFO* pDevInfo, sNBCS_USR_CTXT* pUsrCtxt,
INT4 **pperrDevice, sNBCS_HNDL *pDevHandle, UINT2
numDev)
```

Inputs	`pGroupHandle`	: pointer to the group handle
	`pDevInfo`	: pointer to `numDev`-element array of structures describing the devices, and how to locate them
	`pUsrCtxt`	: numDev-element array of user context structures; one for each device being added (optional; may be NULL pointer)
	`pperrDevice`	: (pointer to) an area of memory
	`pDevHandles`	: (pointer to) an area of memory
	`numDev`	: number of devices to be added to group

Outputs ERROR code written to the CSMDB on failure

	`pGroupHndl`	: pointer to the group handle
	`pperrDevice`	: (pointer to) errDevice (inside the GDB)
	`pDevHandles`	: (pointer to) array containing the device handles of the devices in the group

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_MODULE_STATE`
`NBCS_ERR_INVALID_DEV`
`NBCS_ERR_GROUPS_FULL`
`NBCS_ERR_INVALID_ARG`
`NBCS_ERR_DEV_ALREADY_ADDED`
`NBCS_ERR_ADDING_DEVICE_IN_GROUP`
`NBCS_ERR_INVALID_GROUP`

Valid States `NBCS_MOD_READY`

Side Effects Changes the GROUP state to `NBCS_PRESENT`, if the group devices haven't already been added.

Deleting a Group or Devices from a Group: nbcsGroupDelete

This function deletes an existing group or member devices from the group. When deleting an entire group with parameter `purge` equal to logic one and if the group member does not belong to other groups, that device will also be unregistered from the CSDDB. If the device belongs to some other group, the state of that device will be unchanged. If `purge` is FALSE, the device will not be deleted from the CSDDB. For group deletion, set `pDevHndl` to NULL. `numDev` is ignored if it is a group deletion.

Prototype

```
INT4 nbcsGroupDelete(sNBCS_HNDL groupHandle, UINT1
purge, sNBCS_HNDL* pDevHandle, UINT2 numDev)
```

Inputs
`groupHandle` : group handle (from `nbcsGroupAdd`)
`purge` : if logic one, the member device will also be deleted if it does not belong to any other groups
`pDevHandle` : pointer to array of device handles of devices to be deleted from the existing group; It should be NULL if this is a group deletion.
`numDev` : number of devices to be deleted from group; ignored if it is a group deletion.

Outputs None

Returns
Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_MODULE_STATE`
`NBCS_ERR_DELETING_DEVICE_IN_GROUP`
`NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_ARG`
`NBCS_ERR_INVALID_GROUP`

Valid States `NBCS_MOD_IDLE, NBCS_MOD_READY`

Side Effects See above for possible impact on the state of the device

Getting the state of a Group: nbcsGroupGetState

This function retrieves the state of a given group.

Prototype
```
INT4 nbcsGroupGetState(sNBCS_HNDL groupHandle,
eNBCS_DEV_STATE *pState)
```

Inputs
`groupHandle` : group handle (from `nbcsGroupAdd`)
`pState` : pointer to the group state

Outputs `pState` : pointer to the group state

Returns
Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_GROUP`

Valid States `NBCS_MOD_IDLE, NBCS_MOD_READY`

Side Effects None

Deleting a Device: nbcsDelete

This function removes the specified device from the list of devices being controlled by the Narrowband Chipset driver. Deleting a device involves clearing the Chipset Device Data Block (CSDDB) for that device and then releasing its associated device handle.

Prototype `INT4 nbcsDelete(sNBCS_HNDL handle)`

Inputs `handle` : device handle (from `nbcsAdd`)

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`

Valid States `NBCS_PRESENT, NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects Changes the DEVICE state to `NBCS_START`

Initializing a Device: nbcsInit

This function initializes the CSDDB associated with that device during `nbcsAdd`; it also applies a soft reset to the device and configures it according to the DIV passed by the Application. If the DIV is passed as a NULL, all the register bits are to be left in their default state (after a soft reset). `sNBCS_DIV` is a `void`* pointer and will accept DIV. The device handle (SBS or NSE) should be consistent with the actual type of device initialization vector (DIV) passed to this function. A profile number of zero indicates that all the register bits are to be left in their default state. Note that the profile number is ignored UNLESS the passed DIV is NULL.

In addition, this function also operates in the context of a group. It accepts a group initialization vector (GIV) if the handle passed is that of a valid group. The parameter, `profileNum`, is ignored in this case. If the GIV is passed as a NULL, all the register bits in the devices are left in their default state. Note: It is inadvisable to apply this function to a group which has some members already initialized (unless user wants to re-initialize those members). Instead, users should call this function for those devices (with proper DIV or profile number) on an individual basis.

Prototype `INT4 nbcsInit(sNBCS_HNDL handle, sNBCS_DIV *pDiv, UINT2 profileNum)`

Inputs
`handle` : device/group handle (from `nbcsAdd` or `nbcsGroupAdd`)
`pDiv` : (pointer to) Device or Group Initialization Vector
`profileNum` : device initialization profile number

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_INVALID_PROFILE_NUM`
`NBCS_ERR_INVALID_DIV`

Valid States `NBCS_PRESENT`

Side Effects Changes the DEVICE state to `NBCS_INACTIVE`

Resetting a Device: nbcsReset

This function applies a software reset to the Narrowband Chipset device or group. Also resets all the CSDDB contents (except for the user context). This function is typically called before re-initializing the device (via `nbcsInit`). The function acts on either a single device (NSE/SBS) or a group by examining the handle type.

Prototype `INT4 nbcsReset(sNBCS_HNDL handle)`

Inputs `handle` : device/group handle (from `nbcsAdd` or `nbcsGroupAdd`)

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`

Valid States `NBCS_PRESENT, NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects Changes the DEVICE state to `NBCS_PRESENT`

Activating a Device: nbcsActivate

This function restores the state of the specified device/group in the chipset after a de-activate. Hardware interrupts can be re-enabled. The function acts on either a single device (NSE/SBS) or a group by examining the handle type.

Prototype `INT4 nbcsActivate(sNBCS_HNDL handle)`

Inputs `handle` : device/group handle (from `nbcsAdd` or `nbcsGroupAdd`)

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_INVALID_GROUP_STATE`

Valid States `NBCS_INACTIVE`

Side Effects Changes the DEVICE state to `NBCS_ACTIVE`

De-Activating a Device: nbcsDeActivate

This function de-activates the specified device/group in the chipset from operation. Interrupts are masked and all the devices are put into a quiet state via enable bits. The function acts on either a single device (NSE/SBS) or a group by examining the handle type.

Prototype `INT4 nbcsDeActivate(sNBCS_HNDL handle)`

Inputs `handle` : device/group handle (from `nbcsAdd` or `nbcsGroupAdd`)

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_GROUP_STATE`
`NBCS_ERR_INVALID_DEVICE_STATE`

Valid States `NBCS_ACTIVE`

Side Effects Changes the DEVICE state to `NBCS_INACTIVE`

Adding an Initialization Profile: nbcsAddInitProfile

Creates an initialization profile that is stored by the chipset driver. A device can be initialized by passing the initialization profile number to `nbcsInit`. The device type and the initialization vector type have to be consistent.

Prototype `INT4 nbcsAddInitProfile(eNBCS_DEVTYPE type, sNBCS_DIV *pProfile, UINT2 *pProfileNum)`

Inputs
`type` : device type
`pProfile` : (pointer to) initialization profile being added
`pProfileNum` : (pointer to) profile number to be assigned by the driver

Outputs `pProfileNum` : profile number assigned by the driver

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_MODULE_STATE`
`NBCS_ERR_INVALID_ARG`
`NBCS_ERR_INVALID_PROFILE`
`NBCS_ERR_PROFILES_FULL`

Valid States `NBCS_MOD_IDLE, NBCS_MOD_READY`

Side Effects None

Getting an Initialization Profile: nbcsGetInitProfile

Gets the content of an initialization profile given its profile number. It is the user's responsibility to have a large enough buffer for the appropriate DIV.

Prototype `INT4 nbcsGetInitProfile(UINT2 profileNum, sNBCS_DIV *pProfile)`

Inputs
`profileNum` : initialization profile number
`pProfile` : (pointer to) initialization profile

Outputs `pProfile` : contents of the corresponding profile

Returns
Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_MODULE_STATE`
`NBCS_ERR_INVALID_ARG`
`NBCS_ERR_INVALID_PROFILE_NUM`

Valid States `NBCS_MOD_IDLE, NBCS_MOD_READY`

Side Effects None

Deleting an Initialization Profile: nbcsDeleteInitProfile

Deletes an initialization profile given its profile number.

Prototype `INT4 nbcsDeleteInitProfile(UINT2 profileNum)`

Inputs `profileNum` : initialization profile number

Outputs None

Returns
Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_MODULE_STATE`
`NBCS_ERR_INVALID_PROFILE_NUM`

Valid States `NBCS_MOD_IDLE, NBCS_MOD_READY`

Side Effects None

Reading from Device Registers: nbcsRead

This function is used to read a register of a specific Narrowband Chipset device by providing the register number. This function derives the actual address location based on the device handle and register number inputs. It then reads the contents of this address location from the device. Note that a failure to read returns a zero and that any error indication is written to the associated CSDDB.

Prototype `UINT4 nbcsRead(sNBCS_HNDL handle, UINT2 regNum)`

Inputs
`handle` : device handle (from `nbcsAdd`)
`regNum` : register number

Outputs ERROR code written to the CSMDB
`NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_REG`
`NBCS_ERR_DEV_ABSENT`

Returns Success = value read
Failure = 0

Valid States `NBCS_PRESENT, NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects Can affect registers that change after a read operation

Writing to Device Registers: nbcsWrite

This function is used to write to a register of a specific Narrowband Chipset device by providing the register number. This function derives the actual address location based on the device handle and register number inputs. It then writes the contents of this address location to the device. Note that a failure to write returns a zero and that any error indication is written to the CSDDB.

Prototype `UINT4 nbcsWrite(sNBCS_HNDL handle, UINT2 regNum, UINT4 value)`

Inputs
`handle` : device handle (from `nbcsAdd`)
`regNum` : register number
`value` : value to be written

Outputs ERROR code written to the CSMDB
`NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_REG`
`NBCS_ERR_DEV_ABSENT`

Returns Success = value written
Failure = 0

Valid States `NBCS_PRESENT, NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects Can change the configuration of the device

Reading from a block of Device Registers: nbcsReadBlock

This function is used to read a register block of a specific Narrowband Chipset device by providing the starting register number and the size to read. This function derives the actual start address location based on the device handle and starting register number inputs. It then reads the contents of this data block from the device. Note that a failure to read returns a zero and that any error indication is written to the CSDDB. It is the user's responsibility to allocate enough memory for the block read.

Prototype `UINT4 nbcsReadBlock(sNBCS_HNDL handle, UINT2 startRegNum, UINT2 size, UINT4 *pblock)`

Inputs

`handle`	: device handle (from `nbcsAdd`)
`startRegNum`	: starting register number
`size`	: size of the block to read
`pblock`	: (pointer to) the block to read

Outputs ERROR code written to the CSMDB

`NBCS_ERR_INVALID_DEV`
`NBCS_ERR_DEV_ABSENT`
`NBCS_ERR_INVALID_REG`

`pblock` : (pointer to) the block read

Returns Success = Last register value read
Failure = 0

Valid States `NBCS_PRESENT, NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects Can affect registers that change after a read operation

Writing to a Block of Device Registers: nbcsWriteBlock

This function is used to write to a register block of a specific Narrowband Chipset device by providing the starting register number and the block size. This function derives the actual starting address location based on the device handle and starting register number inputs. It then writes the contents of this data block to the actual device. A bit from the passed block is only modified in the device's registers if the corresponding bit is set in the passed mask. Note that any error indication is written to the CSDDB.

Prototype `UINT4 nbcsWriteBlock(sNBCS_HNDL handle, UINT2 startRegNum, UINT2 size, UINT4 *pblock, UINT4 *pmask)`

Inputs

`handle`	: device handle (from `nbcsAdd`)
`startRegNum`	: starting register number
`size`	: size of block to read
`pblock`	: (pointer to) block to write
`pmask`	: (pointer to) mask

Outputs ERROR code written to the CSMDB
`NBCS_ERR_INVALID_DEV`
`NBCS_ERR_DEV_ABSENT`
`NBCS_ERR_INVALID_REG`

Returns Success = Last register value written
Failure = 0

Valid States `NBCS_PRESENT, NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects Can change the configuration of the Device

5.2 Interface/Clock Configuration

This block provides functions to configure the bus interface of the chipset and controls the CSU/DLLs operation.

Configuring Bus Interface: nbcsIntfCfgBus

SBS devices in the chipset can either operate in SBI or TeleCombus mode. There are bus-related parameters to configure in either of the bus modes. This function configures the bus mode and various other aspects of the bus operating mode. In addition, the current state of the configuration can be read back using this function. (Note: Interface configuration parameters belong to the SBS devices only. NSE device operation does not change whether the chipset system is in SBI or TeleCombus mode).

This function accepts group handle and configures the group of SBS with the same bus configuration parameter given by `pBusCfg` in "set" mode. However, the function does not work with a group handle in "get" mode. It is user's responsibility to make sure the buffer is large enough to hold the parameters returned for all members.

Prototype
```
INT4 nbcsIntfCfgBus(sNBCS_HNDL handle,
sNBCS_CFG_INTF_BUS *pBusCfg, UINT1 accMode)
```

Inputs

`handle`	: device/group handle (from `nbcsAdd` or `nbcsGroupAdd`);
`pBusCfg`	: pointer to the bus interface configuration block
`accMode`	: 0 = get, 1 = set

Outputs

`pBusCfg`	: pointer to the bus interface configuration block when accMode equals 0

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_ARG`
`NBCS_ERR_INVALID_DEV`

```
NBCS_ERR_GROUPS_MIXED_DEV
NBCS_ERR_INVALID_GROUP_STATE
NBCS_ERR_INVALID_DEVICE_STATE
NBCS_ERR_INVALID_MODE
```

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Configuring Bus Payload Type: nbcsIntfCfgPyld

This function configures payload type once the bus type is defined. For SBI bus, each SPE can be configured to carry various kind of traffic such as T1, E1, fractional T1, and DS3. For TeleCombus, virtual tributaries are identified as VT1.5, VT2, VT3, and VT6 or the entire SPE is defined to carry DS3/E3 traffic. Payload symmetry is assumed in the incoming and outgoing direction of a SBS port.

This function accepts group handle and configures the group of SBS with the same payload type given by `pPyldCfg` in “set” mode. However, the function does not work with a group handle in “get” mode. It is user’s responsibility to make sure the buffer is large enough to hold the parameters returned for all members.

The `pPyldCfg` is a `void*` type to allow various types of payload configuration structures for different bus types, SBI or TeleCombus. In the case of SBI bus configuration, `pPyldCfg` should be a pointer of type `sNBCS_CFG_PYLD_SBI` typecasted as a `void*` pointer; for TeleCombus mode, it should be of type `sNBCS_CFG_PYLD_TCB` typecasted as a `void*` pointer.

Prototype

```
INT4 nbcsIntfCfgPyld(sNBCS_HNDL handle,
sNBCS_CFG_PYLD *pPyldCfg, UINT1 accMode)
```

Inputs

	`handle`	: device/group handle (from `nbcsAdd` or `nbcsGroupAdd`);
	`pPyldCfg`	: pointer to the bus payload configuration block
	`accMode`	: 0 = get, 1 = set

Outputs

	`pPyldCfg`	: pointer to the bus payload configuration block when accMode equals 0

Returns

Success = `NBCS_SUCCESS`

Failure =

```
NBCS_ERR_INVALID_DEV
NBCS_ERR_INVALID_DEVICE_STATE
NBCS_ERR_INVALID_MODE
NBCS_ERR_INVALID_ARG
NBCS_ERR_GROUPS_MIXED_DEV
NBCS_ERR_INVALID_GROUP_STATE
```

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Configuring SBI Bus Tributaries: nbcsIntfCfgTrib

This function configures the tributaries of the SBI bus. Attributes are bus output enable (applicable for outgoing bus only), CAS processing enable, justification request enable, and transparent virtual tributaries. This function cannot be used when the SBS device is configured for TeleCombus mode. Payload symmetry is assumed in the incoming and outgoing direction of a SBS port. If DS0 CAS processing is specified at initialization (by the field `casMuxMode` in MIV), then user relinquishes the control of all the CAS enable bits in the incoming direction of the SBS (ICASM) to the CSD and this function can no longer be used to access these CAS enable bit on a per tributary basis. Note that the outgoing direction is still under user's control even with CAS option selected at initialization. The CSD enables or disables the necessary number of CAS-enabled tributaries internally to carry CAS DS0 traffic. The number of dedicated CAS routes is specified by the API `nbcsFmgtRsvpCasRoute`. With the `casMuxMode` off, the CAS bits are enabled/disabled symmetrically in both the incoming and outgoing direction in the SBS.

This function accepts a group handle and configures the group of SBS with the same tributary configuration given by `pTribCfg`.

Prototype

```
INT4 nbcsIntfCfgTrib(sNBCS_HNDL handle,
sNBCS_TRIB_SBI *pTrib, sNBCS_CFG_TRIB_SBI
*pTribCfg)
```

Inputs

`handle`	: device/group handle (from `nbcsAdd` or `nbcsGroupAdd`);
`pTrib`	: pointer to the virtual tributary
`pTribCfg`	: pointer to the tributary configuration block

Outputs

`pTribCfg`	: pointer to the tributary payload configuration block when accMode equals 0

Returns

Success = `NBCS_SUCCESS`

Failure =
```
NBCS_ERR_INVALID_DEV
NBCS_ERR_INVALID_DEVICE_STATE
NBCS_ERR_INVALID_ARG
NBCS_ERR_INVALID_BUS_TYPE
NBCS_ERR_INVALID_TRIB
NBCS_ERR_GROUPS_MIXED_DEV
NBCS_ERR_INVALID_GROUP_STATE
NBCS_ERR_POLL_TIMEOUT
NBCS_FAILURE
```

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Configuring the CSU/DLL: nbcsIntfCfgCsu

There is one CSU, one system DLL, one reference DLL in SBS devices and two CSUs in NSE devices. This function controls the operation of the CSUs in the chipset. The CSU can either be in low-power or normal mode, or can be reset. The DLL can be set up to ignore phase difference.

This function accepts a group handle and acts on the group of SBS using the information given by `pCntl`.

Prototype `INT4 nbcsIntfCfgCsu(sNBCS_HNDL handle, sNBCS_CFG_INTF_CSU* pCntl)`

Inputs

`handle`	: device/group handle (from `nbcsAdd` or `nbcsGroupAdd`);
`pCntl`	: pointer to the CSU/DLL configuration block

Outputs None

Returns

Success = `NBCS_SUCCESS`

Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_INVALID_ARG`
`NBCS_ERR_DEV_ABSENT`
`NBCS_ERR_INVALID_GROUP_STATE`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Configuring the C1 Frame Pulse Delay: nbcsIntfCfgC1FrmDly

This function configures the C1 frame pulse delay of the given device/group. The delay is a 14 bit unsigned integer from 0 to 16383 in system clock cycles.

Prototype `INT4 nbcsIntfCfgC1FrmDly(sNBCS_HNDL handle, UINT2 dly)`

Inputs

`handle`	: device/group handle (from `nbcsAdd` or `nbcsGroupAdd`);
`dly`	: C1 frame pulse delay value

Outputs None

Returns

Success = `NBCS_SUCCESS`

Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_DEV_ABSENT`
`NBCS_ERR_INVALID_ARG`

NBCS_ERR_INVALID_GROUP_STATE

Valid States NBCS_ACTIVE, NBCS_INACTIVE

Side Effects None

5.3 LVDS Serial Link Control

The following functions control various aspects of the LVDS serial link operation in both SBS and NSE. In both NSE and SBS serial links, user can insert line code violation, force out-of-frame and out-of-character conditions, and center FIFOs. In addition, function is available to select the active link between the working and protection ones in SBS. The same function, when acting on NSE, controls the state of the link, which can be put in normal or standby mode or can be reset.

Inserting line code violation: nbcsLkcInsertLcv

This function is used to enable or disable insertion of line code violations in the LVDS links (32 for NSE-20G and 12 in the case of NSE-8G) and in the working or protection LVDS links in SBS devices. If parameter, `linkDesc`, is assigned to ffh, all links in the device will be operated on.

This function also accepts group handle and acts on all members in the group. If the group contains mixed devices, the only valid entry for `linkDesc` is ffh, which applies to all working and protect links in a SBS and to all ports in a NSE device.

Prototype `INT4 nbcsLkcInsertLcv(sNBCS_HNDL handle, UINT1 linkDesc, UINT1 ena)`

Inputs

handle	: device/group handle (from nbcsAdd or nbcsGroupAdd);
linkDesc	: For SBS devices: 0 = working link 1 = protection link; FFh = all links For NSE: it is the port number ranges from 0-11; for NSE-8G and from 0-31 for NSE- 20G.. FFh = all ports
ena	: 0 = disable, 1 = enable

Outputs None

Returns Success = NBCS_SUCCESS
Failure = NBCS_ERR_INVALID_DEV
NBCS_ERR_INVALID_DEVICE_STATE
NBCS_ERR_DEV_ABSENT
NBCS_ERR_INVALID_ARG

Valid States NBCS_ACTIVE, NBCS_INACTIVE

Side Effects None

Centering transmit FIFO: nbcsLkcCenterFifo

This function is used to center the transmit FIFO in the LVDS links (32 for NSE-20G and 12 in the case of NSE-8G) and in the working or protection LVDS links in SBS devices. If parameter, `linkDesc`, is assigned to FFh, all links in the device will be operated on.

This function also accepts group handle and acts on all members in the group. If the group contains mixed devices, the only valid entry for `linkDesc` is ffh, which applies to all working and protect links in a SBS and to all ports in a NSE device.

Prototype

```
INT4 nbcsLkcCenterFifo(sNBCS_HNDL handle, UINT1
linkDesc)
```

Inputs

`handle`	: device/group handle (from `nbcsAdd` or `nbcsGroupAdd`);
`linkDesc`	: For SBS: 0 = working link, 1 = protection link; FFh = all links; For NSE devices: this is the port number ranges from 0-11; for NSE-8G and from 0-31 for NSE-20G. FFh = all ports.

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_POLL_TIMEOUT`
`NBCS_ERR_DEV_ABSENT`
`NBCS_ERR_INVALID_ARG`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Forcing out-of-character alignment: nbcsLkcForceOca

This function is used to force out-of-character alignment in the LVDS links (32 for NSE-20G and 12 in the case of NSE-8G) and in the working or protection LVDS links in SBS devices. If parameter, `linkDesc`, is assigned to FFh, all links in the device will be operated on.

This function accepts group handle and acts on all members in the group. If the group contains mixed devices, the only valid entry for `linkDesc` is ffh, which applies to all working and protect links in a SBS and to all ports in a NSE device.

Prototype

```
INT4 nbcsLkcForceOca(sNBCS_HNDL handle, UINT1
linkDesc)
```

```
linkDesc)
```

Inputs

`handle`	: device/group handle (from `nbcsAdd` or `nbcsGroupAdd`);
`linkDesc`	: For SBS: 0 = working link, 1 = protection link; FFh = all links; For NSE devices: this is the port number ranges from 0-11; for NSE-8G and from 0-31 for NSE-20G. FFh indicates all ports

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_POLL_TIMEOUT`
`NBCS_ERR_DEV_ABSENT`
`NBCS_ERR_INVALID_ARG`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Forcing out-of-frame alignment: nbcsLkcForceOfa

This function is used to force out-of-frame alignment in the LVDS links (32 for NSE-20G and 12 in the case of NSE-8G) and in the working or protection LVDS links in SBS devices. If parameter, `linkDesc`, is assigned to FFh, all links in the device will be operated on.

This function also accepts group handle and acts on all members in the group. If the group contains mixed devices, the only valid entry for `linkDesc` is ffh, which applies to all working and protect links in a SBS and to all ports in a NSE device.

Prototype

```
INT4 nbcsLkcForceOfa(sNBCS_HNDL handle, UINT1
linkDesc)
```

Inputs

`handle`	: device/group handle (from `nbcsAdd` or `nbcsGroupAdd`);
`linkDesc`	: For SBS: 0 = working link, 1 = protection link; FFh = all links; For NSE devices: this is the port number ranges from 0-11; for NSE-8G and from 0-31 for NSE-20G. FFh indicates all ports.

Outputs None

Returns Success = `NBCS_SUCCESS`

Failure = NBCS_ERR_INVALID_DEV
NBCS_ERR_INVALID_DEVICE_STATE
NBCS_ERR_POLL_TIMEOUT
NBCS_ERR_DEV_ABSENT
NBCS_ERR_INVALID_ARG

Valid States NBCS_ACTIVE, NBCS_INACTIVE

Side Effects None

Controlling LVDS link operation mode: nbcsLkcCntl

This function allows user to control the current operation mode of a specified link in the SBS and NSE devices. In NSE devices, a link is by default in standby mode. The user can reset the link (which puts the link in normal mode after reset) or put it in a standby (low power) mode. Resetting or putting the link in normal mode brings the link out of standby mode. The valid range parameter linkDesc is 0-31 for NSE-20G and 0-11 in the case of NSE-8G. In SBS devices, this function enables/disables the selected LVDS links. If the receive direction is selected, it also controls whether the working or the protection LVDS link is the active link. This active link selection is, however, only functional when the software control option is enabled which is specified in the MIV during chipset initialization. In addition, the following parameters, path termination mode, ILC FIFO threshold level and FIFO timeout constant, supplied by the user during initialization via DIV will be restored when the link is selected to bring out from low-power state. This is due to the fact that these parameters cannot be updated at the hardware level when the link is in low-power state.

This function also accepts group handle and acts on all members in the group. If the group contains mixed devices, the only valid entry for linkDesc is ffh, which applies to all working and protect links in a SBS and to all ports in a NSE device.

When SBS device is involved in this function, the combination of dir equals 1 or 2, linkDesc equals 0xff and opMode equals 2 is disallowed because the working and the protect link of the SBS(s) cannot be active simultaneously. In addition, setting opMode to 2 for either the working or protect link is only effective when the link is controlled by software (as indicated by the field, wpLinkCntl, in MIV).

It is strongly encouraged to turn off any unused links in the system with this function.

Prototype `INT4 nbcsLkcCntl(sNBCS_HNDL handle, UINT1 linkDesc, UINT1 dir, UINT1 opMode)`

Inputs

handle	:	device/group handle (from nbcsAdd or nbcsGroupAdd);
linkDesc	:	For SBS: 0 = working link, 1 = protection link; For NSE devices, link number ranges from 0-11 for NSE-8G and from 0-31 for NSE-20G. A ffh indicates all ports.
dir	:	0 = transmit 1 = receive, 2 = both

		transmit and receive.
	opMode	: For SBS: 0 = disabled, 1 = enabled without selecting as active link on receive side, 2 = enabled and selects the link as active on receive side. For NSE: operating mode: 0 = standby, 1 = normal, 2 = reset

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_DEV_ABSENT`
`NBCS_ERR_INVALID_ARG`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Configuring LVDS link parameters: nbcsLkcCfg

This function allows user to configure the parameters for a specified link in the SBS and NSE devices. Parameters applicable to both the NSE and SBS are: J0 byte insertion, and path termination mode.

This function also accepts group handle and acts on all members in the group. If the group contains mixed devices, the only valid entry for `linkDesc` is ffh, which applies to all working and protect links in a SBS and to all ports in a NSE device.

Prototype `INT4 nbcsLkcCfg(sNBCS_HNDL handle, UINT1 linkDesc, sNBCS_CFG_LKC *pCfg)`

Inputs	handle	: device/group handle (from `nbcsAdd` or `nbcsGroupAdd`);
	linkDesc	: For SBS: 0 = working link, 1 = protection link; FFh = all links. For NSE devices, link number ranges from 0-11 for NSE-8G and from 0-31 for NSE-20G. A ffh indicates all ports.
	pCfg	: pointer to the configuration structure

Outputs `None`

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_DEV_ABSENT`
`NBCS_ERR_INVALID_ARG`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Inserting Test Pattern in LVDS link: nbcsLkcInsertTp

This function enables/disables the insertion of test patterns into the LVDS links. It also accepts a group handle and acts on all members in the group. If parameter, `linkDesc`, is assigned to FFh, all links in the device will be operated on. If the group contains mixed devices, the only valid entry for `linkDesc` is ffh, which applies to all working and protect links in a SBS and to all ports in a NSE device.

Prototype

```
INT4 nbcsLkcInsertTp(sNBCS_HNDL handle, UINT1
linkDesc, UINT2 tp, UINT1 ena)
```

Inputs

`handle`	: device/group handle (from `nbcsAdd` or `nbcsGroupAdd`);
`linkDesc`	: For SBS: 0 = working link, 1 = protection link; FFh = all link. For NSE devices, link number ranges from 0-11 for NSE-8G and from 0-31 for NSE-20G. A ffh indicates all ports.
`tp`	: test pattern tp[0..9], a 10-bit number
`ena`	: 0 = disable, 1 = enable

Outputs `None`

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_DEV_ABSENT`
`NBCS_ERR_INVALID_ARG`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

5.4 Space/Time Switch Configuration

This logical block provides functions to access the switch setting in the chipset fabric. There are Two types of switches present in the fabric, namely time and space switching provided by SBS and NSE devices respectively.

Mapping the time slot: nbcsStswMapSlot

Establish connections in the space or time switch by writing settings directly to the offline connection page into the hardware device. This mapping function can operate in multiple modes, namely unicast (NBCS_STSW_UNICAST), map (NBCS_STSW_MAP), inport (NBCS_STSW_INPORT), outport (NBCS_STSW_OUTPORT), time_inport (NBCS_STSW_TIME_INPORT) and time_outport (NBCS_STSW_TIME_OUTPORT) as controlled by the parameter `mode`. The first two modes apply to both the SBS and NSE devices while the latter four modes are applicable to NSE devices only.

For space switches (in NSE devices) operating in unicast mode, the connection between the first element pointed to by `pInport` is mapped to the first element indicated by `pOutport` for the time instance indicated by the first element in `pTimeSlot`. Such operation repeats `numSlots` times for all the pairs. It is designed to set up multiple unicast connections in the switch.

For space switches (in NSE devices) operating in map mode which is to update the entire connection map, `pInport` is expected to have 1080n or 9720n elements in TeleCombus/SBI column mode and SBI DS0/CAS modes, respectively, where n the number of ports in the device. The order in the array (pointed to by `pInport`) should be as follows: inport0[0]…inport0[M] inport1[0]…inport1[M]…inportN[0]…inportN[M] where M = frame size - 1 and is 1079 in TeleCombus/SBI column or 9719 in SBI DS0/CAS mode and N = total number of ports - 1, i.e., 11 for NSE-8G or 31 for NSE-20G. The parameters `pInSlot`, `pOutSlot, pOutport` and `numSlots` are all ignored in this mode.

Specific to the NSE devices, the time_inport (NBCS_STSW_TIME_INPORT) mode allows user to configure all the outports (`pOutport`[]) for all timeslots with a fixed inport. In other words, all the outports for all timeslots source data from the fixed inport (`pInport`[0]). `pInSlot`, `pOutSlot`, and `numSlots` are ignored in this mode. The total number of timeslots is expected to be either 1080 (in TeleCombus/SBI column modes) or 9720 (in SBI DS0/CAS modes). The inport (NBCS_STSW_INPORT) mode operates in a similar fashion. The difference is that the parameter `numSlots` is specified by the user to indicate the exact number of timeslots to be mapped.

For time_outport (NBCS_STSW_TIME_OUTPORT) mode, it is very similar to the time_inport mode except that a fixed outport (`pOutport`[0]) is given and the mapping is between the elements in the `pInport` array in all timeslots. `pInSlot`, `pOutSlot`, and `numSlots` are ignored in this mode. Similar to the time_outport mode, the outport mode (NBCS_STSW_OUTPORT) requires the user to provide `numSlots`, which specifies the total number of timeslots to be mapped.

For time switches (in SBS devices) operating in unicast mode, the user supplies an array of incoming bytes/columns and an array of the corresponding outgoing bytes/columns. One to one mapping is assumed for the `pInSlot` array and the `pOutSlot` array i.e. `pInSlot[0]` mapped to `pOutSlot[0]`, `pInSlot[1]` mapped to `pOutSlot[1]` and so on.

For time switches (in SBS devices) operating in map mode, the user supplies the `pInSlot` array with 1080 and 9720 elements in column and byte switching mode respectively. The `pOutSlot` array is ignored.

Prototype

```
INT4 nbcsStswMapSlot(sNBCS_HNDL handle, UINT1
swDesc, eNBCS_ACCESSMODE_STSW mode, UINT2
*pInSlot, UINT1 *pInport, UINT2 *pOutSlot,
UINT1 *pOutport, UINT4 numSlots)
```

Inputs

`handle`	: device handle (from `nbcsAdd`)
`swDesc`	: switch identifier for SBS: 0 = transmit, 1 = receive. Ignored for NSE type
`mode`	: access mode
`pInSlot`	: pointer to in time slot(s)
`pInport`	: pointer to in space port(s)
`pOutSlot`	: pointer to out time slot(s)
`pOutport`	: pointer to out space port(s)
`numSlots`	: number of slots presented

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_INVALID_ARG`
`NBCS_ERR_STSW_ACCESS`
`NBCS_ERR_DEV_ABSENT`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Getting the source slot: nbcsStswGetSrcSlot

This function returns the source space/time slot(s) map to the destination space/time slot(s) from the offline connection page directly from the hardware device. It operates in either of the two modes, unicast or map mode controlled by the parameter `mode`.

For space switch (found in NSE devices) operating in unicast mode, the inport mapped to the given outport in time instance (first element pointed to by `pInSlot`) will be returned in buffer pointed to by `pInport`. The total number retrieved is indicated by the parameter `numSlots`. In map mode, the entire connection map is returned to the buffer supplied by the user via `pInport`. The order in the array is as follows: inport0[0]…inport0[N-1] inport1[0]…inport1[N-1]…inportM[0]…inportM[N-1] where M = frame size - 1 and is 1079 in TeleCombus/SBI column or 9719 in SBI DS0/CAS mode and N = total number of ports, i.e., 12 for NSE-8G or 32 for NSE-20G. The only two valid access modes are NBCS_STSW_UNICAST and NBCS_STSW_MAP.

For time switch (found in SBS devices) operating in unicast mode, the source timeslots are retrieved by `pInSlot` for the destination timeslot(s) given by array `pOutSlot`. The total number is indicated by parameter `numSlots`. In map mode, all source timeslots will be retrieved by `pInSlot`, `pOutSlot`, and `numSlots` are ignored. User is responsible for supplying a large enough buffer for the data. The size should be 1080 or 9720 depending on what switching mode, byte or column it is set to. The only two valid access modes are NBCS_STSW_UNICAST and NBCS_STSW_MAP.

Note that this function requires dynamically allocated memory of size N * sizeof(UINT2) when retrieving source slot information in SBS map mode where N is 9720 and 1080 in byte and column mode respectively.

Prototype

```
INT4 nbcsStswGetSrcSlot(sNBCS_HNDL handle,
UINT1 swDesc, eNBCS_ACCESSMODE_STSW mode, UINT2
*pInSlot, UINT1 *pInport, UINT2 *pOutSlot,
UINT1 *pOutport, UINT4 numSlots)
```

Inputs

`handle`	: device handle (from `nbcsAdd`)
`swDesc`	: switch identifier for SBS: 0 = transmit, 1 = receive. Ignored for NSE type
`mode`	: access mode
`pInSlot`	: pointer to in time slot(s)
`pInport`	: pointer to in space port(s)
`pOutSlot`	: pointer to out time slot(s)
`pOutport`	: pointer to out space port(s)
`numSlots`	: number of slots presented

Outputs None

Returns

Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_INVALID_ARG`
`NBCS_ERR_STSW_ACCESS`
`NBCS_ERR_MEM_ALLOC`
`NBCS_ERR_DEV_ABSENT`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Copying connection page: nbcsStswCopyPage

This function copies connection page settings from one to another. The copying can be from active to inactive page within the same switch in the device, or can be from inactive to inactive page across different devices of the same type. Note that copying within the same switch can easily be achieved if auto page copy is enabled and this function will not be necessary.

Note that this function requires dynamically allocated memory of size N * numNsePorts * sizeof(UINT1) when copying page from one NSE to a different one. N is 9720 or 1080 in byte and column mode respectively. numNsePorts is 12 or 32 for NSE-8G or NSE-20G device respectively.

Prototype `INT4 nbcsStswCopyPage(sNBCS_HNDL srcHandle, UINT1 srcSwDesc, sNBCS_HNDL dstHandle, UINT1 dstSwDesc)`

Inputs

`srcHandle`	: device handle (from `nbcsAdd`);
`srcSwDesc`	: source switch descriptor. For SBS devices, 0 = transmit switch, 1 = receive switch. Ignored in NSE
`dstHandle`	: device handle (from `nbcsAdd`) of the destination. Ignored in group mode.
`dstSwDesc`	: destination switch descriptor. For SBS devices, 0 = transmit switch, 1 = receive switch. Ignored in NSE and in group mode

Outputs `None`

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_INVALID_ARG`
`NBCS_ERR_STSW_ACCESS`
`NBCS_ERR_MEM_ALLOC`
`NBCS_ERR_DEV_ABSENT`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Getting active connection page number: nbcsStswGetPage

This function retrieves the active connection page of the switch for the specified device. In the case of group, the buffer (`pPageNum`) has to be large enough to hold the active page number . In the case of a distributed system configuration, the page information of the remote SBS is obtained from the in-band link header byte PAGE[1:0]. The in-band link controller on the remote SBS has to be enabled for this API to function correctly.

Prototype `INT4 nbcsStswGetPage(sNBCS_HNDL handle, UINT1 swDesc, UINT1 *pPageNum)`

Inputs

`handle`	: device handle (from `nbcsAdd`);
`swDesc`	: switch descriptor. For SBS devices, 0 = transmit switch, 1 = receive switch; For NSE devices, this is ignored

	pPageNum	: pointer to (array of) the active page number
Outputs	pPageNum	: pointer to (array of) the active page number

Returns Success = NBCS_SUCCESS
Failure = NBCS_ERR_INVALID_DEV
NBCS_ERR_INVALID_DEVICE_STATE
NBCS_ERR_INVALID_ARG
NBCS_ERR_STSW_ACCESS
NBCS_ERR_DEV_ABSENT
NBCS_ERR_POLL_TIMEOUT

Valid States NBCS_ACTIVE, NBCS_INACTIVE

Side Effects None

Toggling the connection page: nbcsStswTogglePage

This function toggles the connection page(s) of the system. The handle should be that of a NSE device. It queries the current active page of all the (registered) devices in the system and promotes the inactive page(s) of each devices. The page toggling is synchronized with the C1 frame pulse (received by the specified NSE) in the system. In the case of a distributed system configuration, this function relies on the PAGE[1:0] bits in the in-band link to control the page switching in remote SBSs. As a result, all the remote SBSs have to be configured to listen to the PAGE bits in the in-band link header for a page switch; otherwise, this function will not operate correctly. A callback function, cbackC1FP, (if registered) is issued and can be treated as a notification of this function.

Prototype INT4 nbcsStswTogglePage(sNBCS_HNDL handle)

Inputs handle : NSE device handle (from nbcsAdd)

Outputs None

Returns Success = NBCS_SUCCESS
Failure = NBCS_ERR_INVALID_DEV
NBCS_ERR_INVALID_DEVICE_STATE
NBCS_ERR_INVALID_ARG
NBCS_ERR_STSW_ACCESS
NBCS_ERR_DEV_ABSENT

Valid States NBCS_ACTIVE, NBCS_INACTIVE

Side Effects None

Setting active connection page number: nbcsStswSetPage

This function sets the active connection page of the switch for the specified device/group. The operation is asynchronous. In conjunction with the API `nbcsEventDetectC1FP`, user can set the page synchronously with the C1 frame pulse. This function can be invoked in callback function such as `cbackC1FP`.

Prototype `INT4 nbcsStswSetPage(sNBCS_HNDL handle, UINT1 swDesc, UINT1 pageNum)`

Inputs

handle	: device/group handle (from `nbcsAdd` or `nbcsGroupAdd`);
swDesc	: switch descriptor. For SBS devices, 0 = transmit switch, 1 = receive, ffh = both transmit and receive switch, This field is ignored in NSE.
pageNum	: active page number

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_INVALID_ARG`
`NBCS_ERR_STSW_ACCESS`
`NBCS_ERR_INVALID_MODE`
`NBCS_ERR_DEV_ABSENT`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

5.5 In-band Communication Link

In-band link communication control. Services provided include receiving and sending data/header bytes across the link, controlling the operation mode of the links

Controlling in-band link controller: nbcsIlcCntl

This function enables/disables the in-band link controller of the specified link in the chipset. It also operates on groups.

Prototype `INT4 nbcsIlcCntl(sNBCS_HNDL handle, UINT1 linkDesc, UINT1 dir, UINT1 enable)`

Inputs

handle	: device/group handle (from `nbcsAdd` or `nbcsGroupAdd`)

	`linkDesc`	: link descriptor: For SBS, 0 = working link, 1 = protect link, ffh = both working and protect links. For NSE-20G, port number: 0 – 31 For NSE-8G, port number: 0 –11. ffh indicates all ports.
	`dir`	: direction: 0 = transmit, 1 = receive, ffh = both transmit and receive
	`enable`	: 0 = disable, 1 = enable

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_INVALID_ARG`
`NBCS_ERR_DEV_ABSENT`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Retrieving the received header bytes: nbcsIlcGetRxHdr

This function retrieves the received header bytes, LINK, PAGE, USER, and AUX from the in-band link controller. In the case of retrieving header information from an NSE ILC, user can either specify the transmitting SBS device using `rxHandle`, or by specifying the receiving port number by `linkDesc` with `txHandle` equals to NULL. When SBS device is receiving, `txHandle` is ignored and `linkDesc` is used to distinguish the working and protect link in the device.

Prototype

```
INT4 nbcsIlcGetRxHdr (sNBCS_HNDL rxHandle,
sNBCS_HNDL txHandle, UINT1 linkDesc,
sNBCS_HEADER_ILC *pHdr)
```

Inputs	`rxHandle`	: device handle (from `nbcsAdd`) of the receiving device
	`txHandle`	: device handle (from `nbcsAdd`) of the transmitting device
	`linkDesc`	: link descriptor: For SBS, 0 = working receive, 1 = protect receive; For NSE-20G, port number from 0 – 31 and 0-11 for NSE-8G.
	`pHdr`	: pointer to the header byte structure
Outputs	`pHdr`	: pointer to the header byte structure

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_INVALID_ARG`
`NBCS_ERR_ILC_INVALID_OP`
`NBCS_ERR_POLL_TIMEOUT`
`NBCS_ERR_DEV_ABSENT`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Retrieving the received messages: nbcsIlcGetRxMsg

This function retrieves one or more ILC messages from the Rx FIFO of one or more links for a device (SBS or NSE) in the chipset. (A maximum of 8 messages per port can be retrieved each time this function is called.)

`pRxBufDesc` points to an array of `numDesc` buffer descriptors, one for each link from which a message is to be retrieved. Using the field, `linkDesc`, each buffer descriptor indicates the link (`linkDesc` in SBS, indicates whether it is working receive, or protect receive; in NSE-20G, it is the port number from 0 to 31; and in NSE-8G, it is the port number from 0-11) from which to read, the maximum number of messages to read (`numMsgs`), and has a pointer to `numMsgs` message descriptors (`pmsgDesc`). (If `numMsgs` is set to 0, this port will be ignored.)

Each message descriptor contains the location in which the message is to be stored (`pmsg`), and the status of the CRC for that message (`crc`) (returned by the driver).

This function reads up to `numMsgs` messages from each link for which a buffer descriptor exists. The number of messages actually received is returned to the user in the `numMsgs` field of the buffer descriptor. (Setting `numMsgs` to 8 will always read all available messages in the Rx FIFO.)

Alternatively, user can supply the function an array of handles, via `pTxHandle`, of all the remote SBS devices (size of the array is indicated by `numDesc`) that are transmitting to the specified NSE device. This supersedes the `linkDesc` field inside the array of `pRxBufDesc`. All the transmitting SBS devices have to be physically attached to the NSE device or an error message will return if at least one of the SBS devices is not. If a SBS device is receiving rather than a NSE, `pTxHandle` is ignored in this case.

The parameter `pyldSz` controls the number of bytes to be read in one message. The maximum payload size in a message is 32 bytes. This function only attempts to read the number of bytes specified in `pyldSz`. This gives the user the ability to avoid reading extra bytes in a message if the payload is known to be fewer than 32 bytes.

This function does not operate in the context of a group.

Prototype
```
INT4 nbcsIlcGetRxMsg (sNBCS_HNDL rxHandle,
sNBCS_HNDL* pTxHandle, sNBCS_RXBUF_DESC_ILC
```

`*pRxBufDesc, UINT1 numDesc, UINT1 pyldSz)`

Inputs

	`rxHandle`	: device handle (from `nbcsAdd`) of the device receiving messages
	`txHandle`	: (pointer to) array of device handle(s) (from `nbcsAdd`) of the SBS device(s) transmitting messages to the NSE
	`pRxBufDesc`	: (pointer to) buffer descriptors (this must point to `numDesc` descriptors)
	`pyldSz`	: payload size (from 1 to 32 bytes)
	`numDesc`	: number of descriptors

Outputs `pRxBufDesc` : pointer to buffer descriptor structures which include the received messages and their corresponding CRC status

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_INVALID_ARG`
`NBCS_ERR_ILC_INVALID_OP`
`NBCS_ERR_POLL_TIMEOUT`
`NBCS_ERR_DEV_ABSENT`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Getting the number of received messages: nbcsIlcGetRxNumMsg

This function queries the total number of messages currently stored in the Rx FIFO for the specified link. For NSE devices, user can either specify the handle(s) of the transmitting SBS device via pT`xHandle` which is a pointer to an array of SBS devices (the number of SBS devices is indicated by `linkDesc`) or the port number via `linkDesc`. If `txHandle` is non-NULL, `linkDesc` will be used to indicate the total number of ports to retrieve. The SBS device has to be physically attached to this NSE device or an error message will be returned.

User can also retrieve the received message level for a specified port by supplying a NULL `pTxhandle` and a valid `linkDesc` (0-11/31 for NSE8/20G), or all ports in an NSE device if `linkDesc` equals to NBCS_ALL_LINKS. Number of messages in Rx FIFO from port 1 to 12/32 ports for NSE-8/20G will be returned. The same applies to SBS devices. If NBCS_ALL_LINKS is provided in `linkDesc`, RxFIFO level for both the working and protect link will be returned. User will have to ensure that `pTxHandle` is NULL and the buffer is large enough to hold the returned values in those cases when number of ports is greater than one.

Prototype

```
INT4 nbcsIlcGetRxNumMsg (sNBCS_HNDL rxHandle,
sNBCS_HNDL* pTxHandle, UINT1 linkDesc, UINT1
*pNumRxMsg)
```

Inputs	`rxHandle`	: device handle (from `nbcsAdd`) of the receiving device
	`pTxHandle`	: (pointer to)device handle(s) (from `nbcsAdd`) of the transmitting device(s)
	`linkDesc`	: link descriptor when `pTxHandle` is NULL: For SBS, 0 = working receive, 1 = protect receive; 0xff = both links. For NSE-20G, port number from 0 – 31 and 0-11 for NSE-8G. A 0xff indicates all ports. When `pTxHandle` is non-NULL, this indicates the total number of ports to retrieve.
	`pNumMsg`	: pointer to the buffer that holds the number of messages stored in FIFO
Outputs	`pNumMsg`	: pointer to the buffer that holds the number of messages stored in FIFO

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_INVALID_ARG`
`NBCS_ERR_ILC_INVALID_OP`
`NBCS_ERR_POLL_TIMEOUT`
`NBCS_ERR_DEV_ABSENT`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Sending in-band link messages: nbcsIlcTxMsg

This function is used to initiate the transmission of one or more in-band messages on one or more links for a given device (SBS or NSE) in the chipset. User can send arbitrary number of messages in one request as specified by `numDesc`. This function can also initiate transmission on multiple links.

`pTxBufDesc` points to an array of descriptors, one for each port on which messages are to be transmitted. The field, `linkDesc,` in this structure indicates the link (for SBS, 0 – working transmit, 1- protect transmit; for NSE, the link descriptor is the port number) on which to transmit, the size of this buffer (`bufSz`), and has a pointer to the buffer to be transmitted (`pBuf`). On return, the `bufSz` field contains the number of bytes transmitted on that link.

Alternatively, user can supply the function an array of handles, via `pRxHandle`, of all the remote SBS devices for the transmission if the transmitting device is a NSE. This supersedes the `linkDesc` field inside the array of `pTxBufDesc`. All the recipient SBS devices have to be physically attached to the NSE device or an error message will return if at least one of the SBS devices is not. If the transmission is originated from a SBS device, `pRxHandle` is ignored.

The length of each message is fixed at 32 bytes. The parameter `pyldSz` controls the number of user bytes that are written in each message. The maximum payload size a message can carry is 32 bytes. If `pyldSz` is less than 32 bytes, the hardware automatically pads the unfilled bytes in the message to 32. (Note that these remaining `(32 - pyldSz)` bytes are uninitialized. Also note that this function is more efficient if `pyldSz` and `pbuf` are multiples of 4.) This function is a blocking function and will not return until the buffer is emptied or an error condition is detected.

This function does not operate in the context of a group.

Prototype

```
INT4 nbcsIlcTxMsg (sNBCS_HNDL txHandle,
sNBCS_HNDL* pRxHandle, sNBCS_TXBUF_DESC_ILC*
pTxBufDesc, UINT1 numDesc, UINT1 pyldSz)
```

Inputs

`txHandle`	: device handle (from `nbcsAdd`) of the transmitting device
`pRxHandle`	: (pointer to) device handle(s) (from `nbcsAdd`) of the receiving device(s)
`pTxBufDesc`	: pointer to Tx buffer descriptor(s)
`numDesc`	: number of buffer descriptor(s)
`pyldSz`	: payload size (from 1 to 32 bytes)

Outputs

`pTxBufDesc`	: pointer to Tx buffer descriptor(s) which contains actual number of bytes sent in each link

Returns

Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_INVALID_ARG`
`NBCS_ERR_ILC_INVALID_OP`
`NBCS_ERR_POLL_TIMEOUT`
`NBCS_ERR_DEV_ABSENT`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Querying Free Space in ILC Tx FIFO: nbcsIlcGetTxFifoLvl

This function is to check the current capacity of the Tx FIFO for the given device and link. This allows the user to find out how many more messages can be written to FIFO for transmission. When the NSE is the transmitting device, the handle of the remote SBS can be given (via `rxHandle`) instead of parameter `linkDesc`. This allows user to easily check the Tx FIFO level to the intended SBS. The parameter `rxHandle` is ignored when SBS is the transmitting device and `linkDesc` is used to distinguish between the working or the protect link.

Prototype

```
INT4 nbcsIlcGetTxFifoLvl (sNBCS_HNDL txHandle,
sNBCS_HNDL rxHandle, UINT1 linkDesc, UINT1*
pNumMsg)
```

Inputs

`txHandle`	: device handle (from `nbcsAdd`) of the transmitting device
`rxHandle`	: device handle (from `nbcsAdd`) of the receiving device
`linkDesc`	: link descriptor: For SBS, 0 = working transmit, 1 = protect transmit; For NSE-20G, port number from 0 – 31 and 0-11 for NSE-8G. Ignored if `txHandle` is NSE device and `rxHandle` is non-NULL
`pNumMsg`	: pointer to free FIFO capacity

Outputs

`pNumMsg`	: pointer to free FIFO capacity

Returns

Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_INVALID_ARG`
`NBCS_ERR_ILC_INVALID_OP`
`NBCS_ERR_POLL_TIMEOUT`
`NBCS_ERR_DEV_ABSENT`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Setting Tx Message Header: nbcsIlcSetTxHdr

This function sets the ILC header bytes, USER[2:0], PAGE[1:0], LINK[1:0], and AUX[7:0] going out to the receiving ILC on the remote device. `txHandle` must be a valid handle for a NSE or SBS device. If a (transmitting) NSE device is given, the user can either specify the handle of the receiving device `rxHandle`, or the raw physical port number, as given by `linkDesc`, it is going to be transmitting on. In this case, the `rxHandle` should be NULL. This parameter should also be assigned NULL if the transmission from the NSE is to another NSE device (in the case of a multi-stage fabric). `pLinkBits, pPageBits, pUserBits`, and `pAuxBits` are pointer to buffers that hold the desirable values to be transmitted. A NULL for any of pointers indicates current value is to be retained. The valid range for `linkDesc` should be from 0-31 and 0-11 for NSE20G and NSE8G respectively. When transmitting the header bits from a NSE device, the PAGE[1:0] and USER[2:0] bits can be transmitted across all links to all the remote SBS devices. When such synchronization is required, `linkDesc` should be assigned to the constant `NBCS_ALL_LINKS` (0xff) and pointer `pPageBits` and/or `pUserBits` should be pointed to buffers containing 32 or 12 (depending on whether it is NSE20G or NSE8G) bytes of PAGE/USER bits.

When a SBS device is transmitting as indicated by txHandle, the `rxHandle` is ignored. Similar to the case of NSE, a NULL should be assigned to the header bit pointers if that header bit is to remain unchanged in the transmission. `linkDesc` is used to distinguish the working or protect link to be used for transmission. `NBCS_ALL_LINKS` (0xff) can also be used to send the same header bits through both the working and protect links.

Prototype

```
INT4 nbcsIlcSetTxHdr(sNBCS_HNDL txHandle,
sNBCS_HNDL rxHandle, UINT1 linkDesc, UINT1
*pLinkBits, UINT1 *pPageBits, UINT1 *pUserBits,
UINT1 *pAuxBits)
```

Inputs

`txHandle`	: device handle of the transmitting device (from `nbcsAdd`)
`rxHandle`	: device handle of the receiving device (from `nbcsAdd`)
`linkDesc`	: link descriptor: For SBS, 0 = working transmit, 1 = protect transmit; 0xff = both working and protect, For NSE-20G, port number from 0-31 and 0-11 for NSE-8G. 0xff = synchronized PAGE and/or USER bits change across all links
`pLinkBits`	: pointer to the LINK[1:0] header bits
`pPageBits`	: pointer to the PAGE[1:0] header bits
`pUserBits`	: pointer to the USER[2:0] header bits
`pAuxBits`	: pointer to the AUX[7:0] header bits

Outputs None

Returns Success = NBCS_SUCCESS
Failure = NBCS_ERR_INVALID_DEV

NBCS_ERR_INVALID_DEVICE_STATE
NBCS_ERR_INVALID_ARG
NBCS_ERR_ILC_INVALID_OP
NBCS_ERR_POLL_TIMEOUT
NBCS_ERR_DEV_ABSENT

Valid States NBCS_ACTIVE, NBCS_INACTIVE

Side Effects None

5.6 PRBS Generator and Monitor

This section describes the functions used to control/configure the PRBS (pseudo-random bit sequence) generator and monitor for the working and protection links of the SBS in the chipset. These functions include configuring the payload, traffic pattern for each STS-1s, controlling the error insertion for each STS-1s and the resynchronization of data received on a per STS-1 basis. All the functions are applicable only to SBS in the chipset. An error code will be returned if attempts are made to invoke these functions on NSE or on groups with members other than SBSs.

Configuring payload for the PRGM: nbcsPrgmCfgPyld

This function configures the payload type of the PRBS generator and monitor. The traffic payload can be one of the following: 12 STS-1s, 4 STS-3c, combination of STS-1s and STS-3cs or a single STS-12c stream. A group handle is not allowed in this function.

Prototype
```
INT4 nbcsPrgmCfgPyld(sNBCS_HNDL handle, UINT1
linkDesc, UINT1 genMon, sNBCS_CFG_PRGM_PYLD
*pPyldCfg, UINT1 accMode)
```

Inputs
- `handle` : device handle (from `nbcsAdd`)
- `linkDesc` : 0 = working link; 1 = protection link
- `genMon` : 0 = generator; 1 = monitor
- `pPyldCfg` : structure containing the payload configuration
- `accMode` : access control: 0 = get, 1 = set

Outputs
- `pPyldCfg` : structure containing the payload configuration when `accMode` = 0

Returns
Success = NBCS_SUCCESS
Failure = NBCS_ERR_INVALID_DEV
NBCS_ERR_INVALID_DEVICE_STATE
NBCS_ERR_DEV_ABSENT
NBCS_ERR_INVALID_ARG

Valid States NBCS_ACTIVE, NBCS_INACTIVE

Side Effects None

Configuring the PRGM: nbcsPrgmCfg

This function configures and controls the PRGM on each STS-1 on the working and protect links in the SBS. It enables/disables the PRGM and configures the linear feedback shift register(LFSR), and the invert PRBS sequence mode or sequential mode on a per STS-1 basis. A group handle is not allowed in this function.

Prototype

```
INT4 nbcsPrgmCfg(sNBCS_HNDL handle, UINT1
linkDesc, UINT1 genMon, UINT1 sts1Path,
sNBCS_CFG_PRGM *pCfg, UINT1 accMode)
```

Inputs

`handle`	: device handle (from `nbcsAdd`)
`linkDesc`	: 0 = working link; 1 = protection link
`genMon`	: 0 = generator; 1 = monitor
`sts1Path`	: STS-1 path, valid range: 0 – 11
`pCfg`	: structure containing the PRGM configuration
`accMode`	: access control: 0 = disable PRGM 1 = enable without configuring, 2 = configure first, then enable, 3 = retrieve configuration

Outputs

`pCfg`	: structure containing the PRGM configuration when `accMode = 3`

Returns

Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_DEV_ABSENT`
`NBCS_ERR_INVALID_ARG`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Forcing a bit error in the PRGM: nbcsPrgmForceErr

This function forces a bit error in the PRBS sequence on the specified STS-1 data stream on the working or protect link in the SBS. One bit error is inserted each time the function is invoked. A group handle is not allowed in this function.

Prototype

```
INT4 nbcsPrgmForceErr(sNBCS_HNDL handle, UINT1
linkDesc, UINT1 sts1Path)
```

Inputs

`handle`	: device handle (from `nbcsAdd`)
`linkDesc`	: 0 = working link; 1 = protection link

	`sts1Path`	: STS-1 path, valid range: 0 – 11

Outputs `None`

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_DEV_ABSENT`
`NBCS_ERR_INVALID_ARG`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Resynchronizing in the PRGM: nbcsPrgmResync

This function resynchronizes the PRBS monitor on a specified STS-1 on the working or protect link in the SBS to the incoming data stream. A group handle is not allowed in this function.

Prototype

```
INT4 nbcsPrgmResync(sNBCS_HNDL handle, UINT1
linkDesc, UINT1 sts1Path)
```

Inputs

`handle`	: device handle (from `nbcsAdd`)
`linkDesc`	: 0 = working link; 1 = protection link
`sts1Path`	: STS-1 path, valid range: 0 – 11

Outputs `None`

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_DEV_ABSENT`
`NBCS_ERR_INVALID_ARG`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

5.7 Narrowband Switching Service Module

Core driver functionality for routing calls, and setting up port protections. The following services are provided.

Mapping virtual tributaries: nbcsFmgtMapTrib

This function maps one or multiple virtual tributaries (largest payload type is STS-1 SPE or TU-3) from the source SBS to the destination SBS. It is designed to work in column mode of both the SBI bus and the TeleCombus mode. The actual hardware connection map setting will not be changed by this function call.

Unicast or multicast (`callType` = NBCS_CALL_MCAST) is supported from one single source tributary. In the case of multicast, multiple destination tributaries are given in the `pdstTrib` array. The number of destination tributaries is given by `numSlot`.

It also operates in the context of UPSR operation. To perform a *protected* drop from a UPSR, the user requires to supply two entries in `psrcSlot` that define the first and the second SBS and in the UPSR and the timeslots that the traffic is dropping from. The callType must be defined to be NBCS_CALL_UPSRDROP in this case. In the case of an *unprotected* drop, only one entry is required (in `psrcSlot`) to specify the UPSR SBS traffic is dropping from. The callType is NBCS_CALL_MCAST for this operation. In either case, multiple destination points may be specified by `pdstSlot` and the total is indicated by `numSlot`.

For multicast connections, it is extremely important not to have duplicate destination slots given in the list `pdstSlot`.

Prototype

```
INT4 nbcsFmgtMapTrib(sNBCS_SLOT* psrcSlot,
sNBCS_SLOT* pdstSlot, UINT2 numSlot,
eNBCS_CALLTYPE callType)
```

Inputs

psrcSlot	: pointer to (array of) the structure of the source tributary
pdstSlot	: pointer to (array of) the structure of the destination tributaries
numSlot	: number of destination tributaries
callType	: call type

Outputs None

Returns

Success = NBCS_SUCCESS
Failure = NBCS_ERR_INVALID_DEV
NBCS_ERR_INVALID_DEVICE_STATE
NBCS_ERR_INVALID_SYS_CONFIG
NBCS_ERR_INVALID_TRIB
NBCS_ERR_INVALID_PYLD
NBCS_ERR_OPA_CONNECT
NBCS_ERR_OPA_SCHEDULE

Valid States NBCS_ACTIVE, NBCS_INACTIVE

Side Effects None

Unmapping virtual tributary: nbcsFmgtUnMapTrib

This function unmaps one or more virtual tributaries (largest payload type is STS-1 SPE or TU-3) from the source SBS to the destination SBS. It is designed to work in both the TeleCombus and the SBI bus (column and byte) modes. The total number of tributaries to be unmapped is indicated by `numSlot`. The unmapping can also be achieved simply by furnishing the destination slot alone. In this case, user can set `psrcSlot` to NULL and the destination slot will be disconnected regardless of what the source slot is. Doing so will also set up the egress bus integrity of the tributaries (for rev B SBS devices only).

Prototype `INT4 nbcsFmgtUnMapTrib(sNBCS_SLOT* psrcSlot, sNBCS_SLOT* pdstSlot, UINT2 numSlot)`

Inputs

`psrcSlot`	: pointer to (array of) the structure of the source tributary
`pdstSlot`	: pointer to (array of) the structure of the destination tributaries
`numSlot`	: number of destination tributaries

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_INVALID_SYS_CONFIG`
`NBCS_ERR_INVALID_TRIB`
`NBCS_ERR_INVALID_PYLD`
`NBCS_ERR_OPA_CONNECT`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Setting chipset to loopback state: nbcsFmgtSetLpbkMode

This function sets the switching fabric to the loopback mode. Records of all existing connections are wiped out. In the centralized model, all SBS and NSE hardware setting are to be changed to support the system loopback mode. In the distributed model, device(s) present in the local microprocessor space are updated. It can be viewed as a reset to the entire switching fabric that puts the system into the known initial state. User should invoke this when a "clean slate" for connection is desirable. The function updates the offline connection page of all local devices and user will have to then perform asynchronous page switch upon a successful invocation.

Prototype `INT4 nbcsFmgtSetLpbkMode(void)`

Inputs None

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_STSW_ACCESS`
`NBCS_FAILURE`

Valid States `NBCS_INACTIVE, NBCS_ACTIVE, NBCS_PRESENT`

Side Effects None

Retrieving Current Connection Map: nbcsFmgtGetMap

This function retrieves the current device setting from the OPA. The structure `conMapHdr` serves both as the input and output between user and the CSD. In this structure, `devHndl` is an input field that allows the user to fill in a valid device handle (returned from `nbcsAdd/nbcsGroupAdd`) to specify the device to retrieve from. Another input field is `devId` that has a different definition for SBS and NSE devices. For SBS device, this field indicates the direction, 0 = ingress and 1 = egress; for NSE device, this denotes the port number, 0-11 or 31 for NSE-8G and NSE-20G respectively. If the port number is ffh, settings for all ports will be retrieved. Lastly, the field `pBuf` is a pointer to the buffer for holding the actual settings. The data format in the buffer varies depending on the device type.

Upon a successful retrieval, the CSD fills in the `accMode` indicating what access mode user should use to populate the settings to the device (via `nbcsStswMapSlot`), and `numSetting` indicating the total number of settings returned. For SBS device, this should be either 9720 or 1080 for byte or column mode operation. For NSE device, this should be either 9720 or 1080 for the two modes on a per port basis. If ffh is specified, this will be one of the four possibilities: 9720/1080 x 32/12 depending on the mode and the NSE type.

The actual settings are copied to the buffer pointed to by `pBuf`, as indicated by the user. For SBS, the order is as follows: inSlot[0],…,inSlot[M-1] where M = 9720 or 1080 for byte and column mode respectively. For NSE device and single port, the order is as follows: inport[0],…,inport[M-1] where M = 9720 or 1080 for byte and column mode respectively. For NSE device in all port mode, the order is inport0[0]…inport0[M] inport1[0]…inport1[M]…inportN[0]…inportN[M] where M = 9719 or 1079 in column or byte mode and N = total number of ports – 1, i.e., 11 or 31 for NSE-8G and NSE-20G respectively.

Table 57 summarizes the definition of all the fields in the header structure for different devices

Table 57: Narrowband Chipset Connection Map Header Definition – Entire Map

Fields	I/O	SBS	NSE
`devHndl`	input	device handle	device handle
`devId`	input	0 = ingress, or 1 = egress	port number: 0-11/31 for NSE-8/20G or ffh for all ports

`devType`	output	NBCS_SBS or NBCS_SBSLITE	NBCS_NSE20G or NBCS_NSE8G
`devNum1`	output	user number of the device	user number of the device
`devNum2`	output	reserved	denotes whether the device is a primary device or secondary one in the case of doubled SBS or doubled SBS/NSE fabric. It is always zero in standard fabric.
`devNum3`	output	user number 3 of the device	user number 3 of the device
`accMode`	output	NBCS_STSW_MAP	NBCS_STSW_TIME_OUTPORT or NBCS_STSW_MAP
`numSetting`	output	number of settings, one of the following: 9720 or 1080	number of settings, one of the following: 9720, 1080, 9720 x 12, 9720 x 32, 1080 x 12, or 1080 x 32
`pBuf`	input	starting location of buffer. Size of this buffer should be large enough to hold the returned data. It is the `numSetting` multiples by the size of a UINT2 integer.	starting location of buffer. Size of this buffer should be large enough to hold the returned data. It is the `numSetting` multiples by the size of a UINT1 integer.
`pBuf2`	n/a	n/a	n/a
`pBuf3`	n/a	n/a	n/a

Prototype `INT4 nbcsFmgtGetMap(sNBCS_CONMAP_STSW* conMapHdr)`

Inputs `conMapHdr` : pointer to the connection map header

Outputs `conMapHdr` : pointer to the connection map header

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_MODULE_STATE`
`NBCS_ERR_INVALID_ARG`
`NBCS_ERR_INVALID_DEV`

NBCS_ERR_INVALID_SYS_CONFIG
NBCS_FAILURE

Valid States NBCS_ACTIVE, NBCS_INACTIVE

Side Effects None

Retrieving Changed Setting of the Connection Map: nbcsFmgtGetChgMap

This function allows user to retrieve the changed (or delta) connection setting of the device(s) as a result of new call connection setup, and protection switchover activities. The user should call this function repeatedly until no further settings are returned (indicated by the numSetting field in the structure, a zero denotes there is no more settings). It is imperative to retrieve all changed settings of a particular type (e.g., call connection settings) once it has started until its completion (this condition is denoted by the field numSetting = 0) before other types (such as protection switchover settings) can be retrieved to preserve data integrity. The function returns an error code if an attempt is made to retrieve settings of different type prior to completely retrieving changed settings of another type. It is, however, not necessary to retrieve changed settings immediately after every single operation (such as a call setup request). User can delay the retrieval until after several operations (such as multiple call requests) as long as no retrieval has been started.

The structure conMapHdr serves both as the input and output between user and the CSD. The only input parameters in the structure are pBuf, pBuf2, and pBuf3, which indicate the beginning of the three buffers for the device settings. The rest of the fields in the structure are filled in by the CSD as output parameters. These include devHndl for the device handle, devId for further device identification (ingress or egress for SBS device, and port number for NSE device), devType for the type for device, accMode for used in API nbcsStswMapSlot indicating what access mode to use, pBuf, pBuf2, pBuf3 are pointers to buffers. Table 58 gives a summary of the definition of all fields in the structure.

If the corresponding fields, sbsAutoUpdate, and/or nseAutoUpdate (specified by MIV) are set, the changed settings are written to the standby (offline) pages of the devices and the settings will not be copied to the user-specified buffers (pBuf, pBuf2 and pBuf3). Otherwise, the settings are copied to their respective buffers without being written to the standby pages of the devices.

Table 58: Narrowband Chipset Connection Map Header Definition – Changed Map

Fields	I/O	SBS	NSE
devHndl	output	device handle	device handle
devId	output	0 = ingress, 1 = egress	port number: 0-11/31 for NSE-8/20G
devType	output	NBCS_SBS or NBCS_SBSLITE	NBCS_NSE20G or NBCS_NSE8G
devNum1	output	user number of the device	user number of the device

Fields	I/O	SBS	NSE
`devNum2`	output	reserved	denotes whether the device is a primary device or secondary one in the case of doubled SBS or doubled SBS/NSE fabric. It is always zero in standard fabric.
`devNum3`	output	user number 3 of the device	user number 3 of the device
`accMode`	output	NBCS_STSW_MAP or NBCS_STSW_UNICAST	NBCS_STSW_TIME_OUTPORT, NBCS_STSW_TIME_INPORT, NBCS_STSW_INPORT, NBCS_STSW_OUTPORT, or NBCS_STSW_UNICAST
`numSetting`	output	number of settings: NBCS_STSW_MAP: either 9720 or 1080 NBCS_STSW_UNICAST: ranges from 0-9720/1080	number of settings NBCS_STSW_MAP: NBCS_STSW_TIME_INPORT: NBCS_STSW_TIME_OUTPORT : either 9720 or 1080 NBCS_STSW_UNICAST NBCS_STSW_INPORT NBCS_STSW_OUTPORT: ranges from 0-9720/1080
`pBuf`	input output	**Input**: user defines the starting location of buffer **Output**: NBCS_STSW_MAP: NBCS_STSW_UNICAST: pointer to inSlot[] array	**Input**: user defines the starting location of buffer **Output**: NBCS_STSW_TIME_INPORT : NBCS_STSW_INPORT pointer to inPort[0] NBCS_STSW_TIME_OUTPORT: NBCS_STSW_OUTPORT NBCS_STSW_UNICAST: pointer to inPort[] array
`pBuf2`	input/ output	**Input**: user defines the starting location of buffer **Output**: NBCS_STSW_MAP: n/a NBCS_STSW_UNICAST: pointer to outSlot[] array	**Input**: user defines the starting location of buffer **Output**: NBCS_STSW_TIME_OUTPORT: NBCS_STSW_OUTPORT: pointer to outPort[0] NBCS_STSW_TIME_INPORT: NBCS_STSW_INPORT: NBCS_STSW_UNICAST: pointer to outPort[] array

Fields	I/O	SBS	NSE
pBuf3	input output	n/a	**Input**: user defines the starting location of buffer **Output**: NBCS_STSW_TIME_INPORT NBCS_STSW_TIME_OUTPORT :n/a NBCS_STSW_UNICAST NBCS_STSW_INPORT NBCS_STSW_OUTPORT: pointer to outSlot[] array

Prototype `INT4 nbcsFmgtGetChgMap(sNBCS_CONMAP_STSW* conMapHdr, eNBCS_FABRIC_SETTING settingType)`

Inputs

`conMapHdr` : pointer to number of device affected

`settingType` : one of the following types of setting to be retrieved: NBCS_SWITCHOVER_SETTING, or NBCS_CALL_SETTING

Outputs

`conMap` : pointer to number of device affected

Returns

Success = `NBCS_SUCCESS`

Failure = `NBCS_ERR_INVALID_MODULE_STATE`
`NBCS_ERR_INVALID_ARG`
`NBCS_ERR_STSW_ACCESS`
`NBCS_ERR_INVALID_SYS_CONFIG`
`NBCS_FAILURE`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Defining the Physical Wiring of the Fabric: nbcsFmgtDefWiring

This function defines how the SBS devices are connected to the NSE(s) core. The user supplies wiring tables that describes how the wiring connection between all the SBS devices and the NSE(s) (edge wiring). The wiring information for the ingress and egress ports of the primary SBSs and then for the secondary SBSs are specified in the four arrays, `pIgrsPriWireTbl, pEgrsPriWireTbl, pIgrsSecWireTbl, and pEgrsSecWireTbl` respectively. The secondary wiring information is only required if it is a doubled SBS or a double SBS/NSE fabric. In a standard fabric, the secondary SBS wiring tables are ignored.

The table is used to construct an internal look-up table for translating logical SBS numbers to SBS CSDDBs.

Prototype

```
INT4 nbcsFmgtDefWiring(
sNBCS_EDGE_WIRING *pIgrsPriWireTbl,
sNBCS_EDGE_WIRING *pEgrsPriWireTbl,
sNBCS_EDGE_WIRING *pIgrsSecWireTbl,
sNBCS_EDGE_WIRING *pEgrsSecWireTbl, UINT2
numEntries )
```

Inputs

pIgrsPriWireTbl	: pointer to the connection table between primary SBSs and NSE core in the ingress direction
pEgrsPriWireTbl	: pointer to the connection table between primary SBSs and NSE core in the egress direction
pIgrsSecWireTbl	: pointer to the connection table between secondary SBSs and the NSE core in the ingress direction
pEgrsSecWireTbl	: pointer to the connection table between secondary SBSs and the NSE core in the egress direction
numEntries	: number of entries in the table

Outputs None

Returns Success = NBCS_SUCCESS
Failure = NBCS_ERR_INVALID_MODULE_STATE
NBCS_ERR_INVALID_ARG
NBCS_ERR_INVALID_SYS_CONFIG
NBCS_ERR_INVALID_WIRING

Valid States NBCS_INACTIVE, NBCS_PRESENT, NBCS_ACTIVE

Side Effects None

Mapping DS0 in SBI bus mode: nbcsFmgtMapDS0

This function maps DS0(s) (with or without CAS) from the source tributary of the SBS to the destination tributaries of the SBS. It is only applicable when the chipset is in SBI bus mode and byte switching mode. The actual hardware connection map setting will not be changed by this function call.

For multicast connections, it is extremely important not to have duplicate destination slots given in the list pdstSlot.

The field, cas, specifies whether the CAS-enabled routes are used for the DS0 scheduling. Care should be taken when DS0 switching in a CAS-enabled E1 tributary is involved. The normal range for the timeslots is from 1 to 30. DS0#0 is also accepted. However, since it does not contain any CAS information, the cas bit should not be set if DS0#0 is to be switched. Instead, the non-CAS scheduler should be employed for the timeslot.

Prototype `INT4 nbcsFmgtMapDS0(sNBCS_SLOT* psrcSlot, sNBCS_SLOT* pdstSlot, UINT2 numSlot, UINT1 cas)`

Inputs

`psrcSlot`	: pointer to (array of) the structure of the source tributary/timeslots
`pdstSlot`	: pointer to (array of) the structure of the destination tributaries/timeslots
`numSlot`	: number of connections
`cas`	: 0 = do not use CAS scheduler, 1 = use CAS scheduler

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_INVALID_SYS_CONFIG`
`NBCS_ERR_INVALID_MODE`
`NBCS_ERR_INVALID_TRIB`
`NBCS_ERR_INVALID_ARG`
`NBCS_ERR_OPA_CONNECT`
`NBCS_ERR_OPA_SCHEDULE`
`NBCS_ERR_OPA_DISCONNECT`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Unmapping DS0 in SBI bus mode: nbcsFmgtUnMapDS0

This function unmaps DS0(s) from the source tributary of the SBS to the destination tributary of the SBS. It is only applicable when the chipset is in SBI bus mode and byte switching mode. The unmapping can also be achieved simply by furnishing the destination slot alone. In this case, user can set `psrcSlot` to NULL and the destination slot will be disconnected regardless of what the source slot is. Doing so will also overwrite the DS0 location with 0h (for rev B SBS devices only).

Prototype `INT4 nbcsFmgtUnMapDS0(sNBCS_SLOT* psrcSlot, sNBCS_SLOT* pdstSlot, UINT2 numSlot)`

Inputs

`psrcSlot`	: pointer to (array of) the structure of the source tributary/timeslot
`pdstSlot`	: pointer to (array of) the structure of the destination tributaries/timeslots
`numSlot`	: number of connections

Outputs None

Returns Success = `NBCS_SUCCESS`

Failure = NBCS_ERR_INVALID_DEV
NBCS_ERR_INVALID_DEVICE_STATE
NBCS_ERR_INVALID_SYS_CONFIG
NBCS_ERR_INVALID_MODE
NBCS_ERR_INVALID_TRIB
NBCS_ERR_INVALID_ARG
NBCS_ERR_OPA_CONNECT
NBCS_ERR_OPA_DISCONNECT

Valid States NBCS_ACTIVE, NBCS_INACTIVE

Side Effects None

Reserving total number of virtual tributaries for CAS routes: nbcsFmgtRsvpCasRoute

Applicable only when the fabric is initialized to handle CAS traffic, this function reserves the total number of virtual tributaries as CAS routes for routing CAS DS0 calls. It should be invoked in all CSD instances in the system in case of a distributed one. This function has to be called again after invocation of nbcsFmgtSetLpbkMode.

Prototype INT4 nbcsFmgtRsvpCasRoute(UINT2 numCasRoute)

Inputs numCasRoute : number of VTs reserved for CAS routing

Outputs None

Returns Success = NBCS_SUCCESS
Failure = NBCS_ERR_INVALID_DEVICE_STATE
NBCS_ERR_INVALID_ARG
NBCS_ERR_INVALID_MODE
NBCS_ERR_POLL_TIMEOUT

Valid States NBCS_ACTIVE, NBCS_INACTIVE

Side Effects None

Setting Port Protection: nbcsFmgtSetProtect

This function sets up 1+1, 1:N port protection or UPSR association. For 1+1 port protection, the handles of the working and protection port are specified by port1Handle and port2Handle respectively. For 1:N port protection application, user can set up the protection by either calling this function N times with N different working ports (specified by the handle in port1Handle) or calling this function once by supplying a group handle (specified again by port2Handle) with N working ports grouped together. In the case of UPSR, the handles of the two SBSs involved in a UPSR are specified by port1Handle and port2Handle.

For 1+1 port protection, both the working and protect SBSs must not be engaged in other protection; otherwise, an error code will be returned. Likewise for 1:N protection, all the SBSs should not be involved in other protection at the time of this call with the exception of the protect SBS which may currently be defined as a protect SBS in a 1:N protection. The same applies to the UPSR case.

Prototype
```
INT4 nbcsFmgtSetProtect(sNBCS_HNDL port1Handle,
sNBCS_HNDL port2Handle, eNBCS_PORTPROTECT
protectMode)
```

Inputs

`port1Handle`	: device/group handle (from `nbcsAdd`) of the working SBS(s) or the first SBS in a UPSR
`port2Handle`	: device handle of the protection SBS or the second SBS in a UPSR
`protectMode`	: protection mode: NBCS_PORTPROTECT_1PLUS1, NBCS_PORTPROTECT_1FORN NBCS_PORTPROTECT_UPSR

Outputs None

Returns Success = NBCS_SUCCESS
Failure = NBCS_ERR_INVALID_DEV
NBCS_ERR_INVALID_SYS_CONFIG
NBCS_ERR_OPA_PROTECT_EXIST
NBCS_ERR_INVALID_ARG

Valid States NBCS_INACTIVE, NBCS_ACTIVE

Side Effects None

Clearing Port Protection: nbcsFmgtClearProtect

This function clears a 1+1, 1:N port protection or an UPSR association. For 1+1 port protection, either the working or the protect SBS can be given to clear the protection. In the case of a UPSR, either one of the SBSs can be given for the clearing. For 1:N port protection, the working SBS or SBSs (if a group handle containing all working SBSs is supplied) must be given to clear the 1:N protection. The exception is for the trivial case when N=1 in the 1:N protection, the protect SBS can also be given to clear the protection.

Prototype
```
INT4 nbcsFmgtClearProtect(sNBCS_HNDL sbsHandle)
```

Inputs `sbsHandle` : device handle of the SBS

Outputs None

Returns Success = NBCS_SUCCESS
Failure = NBCS_ERR_INVALID_DEV

NBCS_ERR_INVALID_SYS_CONFIG
NBCS_ERR_OPA_PROTECT_NONEXISTENT
NBCS_ERR_OPA_PROTECT_1FORN

Valid States NBCS_INACTIVE, NBCS_ACTIVE, NBCS_PRESENT

Side Effects None

Switching Over a Port Protection: nbcsFmgtSwitchProtect

This function switches traffic over for a current 1+1 and 1:N port protection or switches virtual paths in a UPSR. In a 1+1 port protection, this function swaps traffic between active and inactive ports. If at the time of the request the working port is active, then this function swaps traffic to the protection port and makes it active. In the 1:N port protection case, this function swaps traffic from the working port (if currently active) to the protection port which then becomes active. If the protection port is currently active protecting other working ports, then this call fails. Calling this function to switch traffic over from the protection port back to the working port always succeeds. The handle of the working port should be given (via sbsSlot) when switching traffic back from protection to working port.

The pending active port after the switch over is returned via argument activeSbsSlot. If a NULL for this parameter is given, no active port information will be returned. The parameter, numSlots, is ignored when performing 1+1 or 1:N port protection switchover.

This function also performs paths switchover in the context of UPSR. When dropping traffic from a UPSR, it is by default from the working path. In the event of a signal degradation in the working path, traffic will instead be dropped from the protect path. One or more protected connections (the total is specified by numSlots) dropped from the UPSR can be specified by sbsSlot. All destinations currently drawing traffic from the specified path(s) will be switched over to the protect path from the active one. The function returns an error code if the first path does not originate from a SBS defined in UPSR protection. (Hence, the handles in the sbsSlot array for subsequent entries can be left unfilled.). If the parameter activeSbsSlot is non-NULL, the pending active path of the first tributary (i.e., the first entry in the sbsSlot array) after a UPSR path switchover will be returned in this parameter.

Prototype

```
INT4 nbcsFmgtSwitchProtect(sNBCS_SLOT* sbsSlot,
UINT2 numSlots, sNBCS_SLOT* activeSbsSlot)
```

Inputs

sbsSlot	: pointer to (array of) tributaries for switchover
numSlots	: number of paths to switchover
activeSbsSlot	: pointer to device handle for active port

Outputs

activeSbsSlot	: pointer to device handle for active port

Returns Success = NBCS_SUCCESS
Failure = NBCS_ERR_INVALID_DEV
NBCS_ERR_INVALID_SYS_CONFIG
NBCS_ERR_INVALID_SWITCHOVER

Valid States NBCS_INACTIVE, NBCS_ACTIVE, NBCS_PRESENT

Side Effects None

5.8 Event Processing Functions

This section of the document describes the functions that perform the following tasks that sets, gets and clears the event mask.

Polling the Chipset Driver Events: nbcsPoll

Commands the chipset driver to poll the underlying drivers for the NSE/SBS device(s). The call will fail unless the chipset driver was initialized (via the MIV when calling nbcsModuleOpen) to operate in polling mode. The function also works in the context of a group.

Prototype `INT4 nbcsPoll(sNBCS_HNDL handle)`

Inputs handle : device/group handle (from nbcsAdd or nbcsGroupAdd)

Outputs None

Returns Success = NBCS_SUCCESS
Failure = NBCS_ERR_INVALID_DEV
NBCS_ERR_INVALID_DEVICE_STATE
NBCS_ERR_INVALID_MODE
NBCS_ERR_DEV_ABSENT

Valid States NBCS_ACTIVE

Side Effects None

Getting the Event Enable Mask: nbcsEventGetMask

This function returns the current setting of the event mask of the Narrowband Chipset device. User has to ensure that the buffer pointed to by pMask is large enough to hold all the masks.

Prototype `INT4 nbcsEventGetMask(sNBCS_HNDL handle, sNBCS_MASK_EVT *pMask)`

Inputs handle : device handle (from nbcsAdd)
pMask : pointer to the event mask structure

Outputs pMask : pointer to updated event mask structure

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_DEV_ABSENT`
`NBCS_ERR_INVALID_ARG`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Setting the Event Mask: nbcsEventSetMask

This function sets the event mask of the Narrowband Chipset device. A field set in the mask enables the processing of the corresponding event for the specified device. Enabled events will be processed and the corresponding callback function (if properly registered) will be invoked when the event occurs. For the zero values in the mask, the processing state of the corresponding event remains unchanged.

The function also operates in the context of a group. All the members (devices) in the group will be set to the mask given by `pMask`. The members have to be the same type or an error code will be returned.

Prototype `INT4 nbcsEventSetMask(sNBCS_HNDL handle, sNBCS_MASK_EVT *pMask)`

Inputs

`handle`	: device/group handle (from `nbcsAdd` or `nbcsGroupAdd`)
`pMask`	: pointer to the event mask structure

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_GROUPS_MIXED_DEV`
`NBCS_ERR_DEV_ABSENT`
`NBCS_ERR_INVALID_ARG`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Clearing the Event Mask: nbcsEventClearMask

This function clears the event mask of the Narrowband Chipset device. A field set in the mask disables the processing of the corresponding event for the specified device. Application will not be notified of the occurrence of any disabled events via any callback functions. For the zero values in the mask, the processing state of the corresponding event remains unchanged.

The function also operates in the context of a group. All the members (devices) in the group will have their event mask cleared as indicated by `pMask`. The members have to be the same type or an error code will be returned.

Prototype `INT4 nbcsEventClearMask(sNBCS_HNDL handle, sNBCS_MASK_EVT *pMask)`

Inputs

handle	: device/group handle (from nbcsAdd or nbcsGroupAdd)
pMask	: pointer to the event mask structure

Outputs None

Returns
Success = NBCS_SUCCESS
Failure = NBCS_ERR_INVALID_DEV
NBCS_ERR_INVALID_DEVICE_STATE
NBCS_ERR_GROUPS_MIXED_DEV
NBCS_ERR_DEV_ABSENT
NBCS_ERR_INVALID_ARG

Valid States NBCS_ACTIVE, NBCS_INACTIVE

Side Effects None

Detecting C1 Frame Pulse: nbcsEventDetectC1FP

This function prepares the CSD to detect the arrival of the C1 frame pulse. A callback function will be issued to notify user the receipt of C1 frame pulse. This is mostly used when a synchronized page switch across the entire fabric is required. The main task for this function is to enable the C1 frame pulse interrupt in the specified underlying NSE or SBS device. For a NSE device, the callback function is called from the context of the ISR (interrupt service routine) of the NSE device driver. It is very important to keep this callback function to a bare minimum of processing. For a SBS device, the callback function is called from the context of a deferred processing routine (DPR) task.

The arrival of the C1 frame pulse depends on the system configuration mode. In a TeleCombus-based system, the C1 frame pulse arrives at every 125 microseconds. In a SBI bus system configured in column mode or DS0 mode without CAS, the pulse comes at every 4 frames, or 500 microseconds. With the presence of CAS DS0 in SBI bus, the pulse is detected at every 48 frames, or 6 milliseconds.

Prototype `INT4 nbcsEventDetectC1FP(sNBCS_HNDL handle, UINT1 dir)`

Inputs

handle	: device handle (from nbcsAdd) of NSE/SBS
dir	: direction for SBS devices: 0 = incoming, 1 = received. Ignored for NSE devices

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_DEV_ABSENT`

Valid States `NBCS_ACTIVE`

Side Effects None

5.9 Status and Counts Functions

Reading the Device Counters: nbcsStatsGetCounts

This function retrieves all the device counts. This routine should be called by the application code, in the context of a task. It is the user's responsibility to ensure that this function is called often enough to prevent the device counts from saturating or rolling over. This function also operates in the context of a group. The buffer pointed to by `pCntr` should be large enough to hold all the returned counts of the members in the group.

Prototype `INT4 nbcsStatsGetCounts(sNBCS_HNDL handle, sNBCS_CNTR *pCntr)`

Inputs
`handle` : device/group handle (from `nbcsAdd` or `nbcsGroupAdd`)
`pCntr` : allocated memory for counts

Outputs `pCntr` : current device counts

Returns Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_INVALID_ARG`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

Getting the Current Status: nbcsStatsGetStatus

This function retrieves a snapshot of the current status from the device registers. This involves retrieving current alarms, status, and clock activity. It also operates in the context of a group. It is the user's responsibility to ensure the buffer indicated by `pStatus` is large enough to hold all the returned status of the members in the group.

Prototype `INT4 nbcsStatsGetStatus(sNBCS_HNDL handle, sNBCS_STATUS *pStatus)`

Inputs
`handle` : device/group handle (from `nbcsAdd` or `nbcsGroupAdd`)
`pStatus` : pointer to allocated memory

Outputs
`pStatus` : current status

Returns
Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_INVALID_ARG`

Valid States `NBCS_ACTIVE, NBCS_INACTIVE`

Side Effects None

5.10 Device Diagnostics

Testing Register Accesses: nbcsDiagTestReg

This function verifies the specified device register access in the chipset by writing and reading back values. It also supports the group operation with the same device type. Mixed devices in the same group is disallowed.

Prototype `INT4 nbcsDiagTestReg(sNBCS_HNDL handle, sNBCS_DIAG_TEST_REG *ptestReg)`

Inputs
`handle` : device/group handle (from `nbcsAdd` or `nbcsGroupAdd`) ; if group, all members must be of same type
`ptestReg` : (pointer to) test structure

Outputs None

Returns
Success = `NBCS_SUCCESS`
Failure = `NBCS_ERR_INVALID_DEV`
`NBCS_ERR_INVALID_DEVICE_STATE`
`NBCS_ERR_INVALID_ARG`
`NBCS_ERR_INVALID_GROUP_STATE`
`NBCS_ERR_DEV_ABSENT`

Valid States `NBCS_PRESENT`

Side Effects None

Testing RAM Accesses: nbcsDiagTestRam

This function verifies the specified device RAM access by writing and reading back values. It also supports the group operation with the same device type. Mixed devices in the same group is disallowed.

Prototype `INT4 nbcsDiagTestRam(sNBCS_HNDL handle, sNBCS_DIAG_TEST_RAM *ptestRam)`

Inputs

handle	: device/group handle (from nbcsAdd or nbcsGroupAdd); if group, all members must be of same type
ptestRam	: (pointer to) test structure

Outputs None

Returns Success = NBCS_SUCCESS
Failure = NBCS_ERR_INVALID_DEV
NBCS_ERR_INVALID_DEVICE_STATE
NBCS_ERR_INVALID_ARG
NBCS_ERR_DEV_ABSENT
NBCS_ERR_INVALID_GROUP_STATE

Valid States NBCS_PRESENT

Side Effects None

Controlling diagnostic loopback: nbcsDiagLpbk

This function controls 3 diagnostic loopback available in the SBS devices.

Prototype `INT4 nbcsDiagLpbk(sNBCS_HNDL handle, eNBCS_LPBK lpbk, UINT1 enable)`

Inputs

handle	: device handle (from nbcsAdd)
lpbk	: specifies one of the three loopback options
enable	: 0 – disable, 1 – enable

Outputs None

Returns Success = NBCS_SUCCESS
Failure = NBCS_ERR_INVALID_DEV
NBCS_ERR_INVALID_DEVICE_STATE
NBCS_ERR_INVALID_ARG
NBCS_ERR_DEV_ABSENT

Valid States NBCS_ACTIVE, NBCS_INACTIVE

Side Effects None

5.11 Callback Functions

The Narrowband Chipset driver has the capability to callback functions within the user code when certain events occur. The names given to the callback functions are given as examples. The addresses of the callback functions are passed during the `nbcsModuleOpen` call (inside a MIV). To avoid using the callbacks, the user should set the address of the callback functions to NULL within the MIV.

Notifying the Application of ILC data received events: cbackIlcRxData

This callback function is provided by the user and is used by the CSD to report in-band link data received events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements. Note: the callback function's address is passed to the driver during the `nbcsModuleOpen` call. If the address of the callback function was passed as a NULL at initialization, then no callback will be made. Since DPV buffer is not employed, there is no need to call `sysNbcsDPVBufferRtn` to release the DPV buffer upon the return of this function.

Prototype
```
void cbackIlcRxData(sNBCS_USR_CTXT usrCtxt,
UINT4 event, UINT4 link)
```

Inputs

`usrCtxt`	: user context (from `nbcsAdd`)
`event`	: event bitmask
`link`	: link descriptor

Outputs None

Returns None

Valid States `NBCS_ACTIVE`

Side Effects None

Notifying the Application of ILC header bits changed events: cbackIlcHead

This callback function is provided by the user and is used by the CSD to report in-band link header bits changed events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements. Note: the callback function's address is passed to the driver during the `nbcsModuleOpen` call. If the address of the callback function was passed as a NULL at initialization, then no callback is made. The `event` field in the DPV is a bit mask that reports all the ILC event(s) encountered. The `info` field in the DPV is encoded as the link descriptor indicating which ILC link is causing the event(s).

Application is responsible for calling `sysNbcsDPVBufferRtn` to release the DPV buffer upon the return of this function.

Prototype
```
void cbackIlcHead(sNBCS_USR_CTXT usrCtxt,
sNBCS_DPV *pdpv)
```

Inputs

`usrCtxt`	: user context (from `nbcsAdd`)
`pdpv`	: (pointer to) DPV that describes this event

Outputs None

Returns None

Valid States `NBCS_ACTIVE`

Side Effects None

Notifying the Application of Interface events: cbackIntf

This callback function is provided by the user and is used by the CSD to report interface-related events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements. Note: the callback function's address is passed to the driver during the `nbcsModuleOpen` call. If the address of the callback function was passed as a NULL at initialization, then no callback is made. The `event` field in the DPV is a bit mask that reports all the INTF event(s) encountered. The `info` field in the DPV is encoded as the link descriptor indicating which link is causing the event(s).

Application is responsible for calling `sysNbcsDPVBufferRtn` to release the DPV buffer upon the return of this function.

Prototype
```
void cbackIntf(sNBCS_USR_CTXT usrCtxt,
sNBCS_DPV *pdpv)
```

Inputs	`usrCtxt`	: user context (from `nbcsAdd`)
	`pdpv`	: (pointer to) DPV that describes this event
Outputs	None	
Returns	None	
Valid States	`NBCS_ACTIVE`	
Side Effects	None	

Notifying the Application of LVDS Link events: cbackLkc

This callback function is provided by the user and is used by the CSD to report LVDS link-related events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements. Note: the callback function's address is passed to the driver during the `nbcsModuleOpen` call. If the address of the callback function was passed as a NULL at initialization, then no callback is made. The `event` field in the DPV is a bit mask that reports all the LKC event(s) encountered. The `info` field in the DPV is encoded as the link descriptor indicating which link is causing the event(s).

Application is responsible for calling `sysNbcsDPVBufferRtn` to release the DPV buffer upon the return of this function.

Prototype

```
void cbackLkc(sNBCS_USR_CTXT usrCtxt, sNBCS_DPV
*pdpv)
```

Inputs	`usrCtxt`	: user context (from `nbcsAdd`)
	`pdpv`	: (pointer to) DPV that describes this event
Outputs	None	
Returns	None	
Valid States	`NBCS_ACTIVE`	
Side Effects	None	

Notifying the Application of Space/time Switch events: cbackStsw

This callback function is provided by the user and is used by the CSD to report space/time switch-related events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements. Note: the callback function's address is passed to the driver during the `nbcsModuleOpen` call. If the address of the callback function was passed as a NULL at initialization, then no callback is made. The `event` field in the DPV is a bit mask that reports all the STSW event(s) encountered. The `info` field in the DPV is encoded as the switch descriptor indicating which space/time switch is causing the event(s).

Application is responsible for calling `sysNbcsDPVBufferRtn` to release the DPV buffer upon the return of this function.

Prototype `void cbackStsw(sNBCS_USR_CTXT usrCtxt, sNBCS_DPV *pdpv)`

Inputs

`usrCtxt`	: user context (from `nbcsAdd`)
`pdpv`	: (pointer to) DPV that describes this event

Outputs None

Returns None

Valid States `NBCS_ACTIVE`

Side Effects None

Notifying the Application of C1 Frame Pulse: cbackC1FP

This callback function is provided by the user and is used by the CSD to report arrival of C1FP events back to the application. This function should be non-blocking and short because it is invoked in the interrupt service routine (ISR) context. User should exercise caution when filling in this callback function. For instance, it should not wait for any semaphores which creates a blocking situation. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements. Note: the callback function's address is passed to the driver during the `nbcsModuleOpen` call. If the address of the callback function was passed as a NULL at initialization, then no callback is made. Since DPV buffer is not employed, there is no need to call `sysNbcsDPVBufferRtn` to release the DPV buffer upon the return of this function.

Prototype `void cbackC1FP(sNBCS_USR_CTXT usrCtxt, UINT4 rsv1, UINT4 rsv2)`

Inputs

`usrCtxt`	: user context (from `nbcsAdd`)
`rsv1`	: reserved field 1

	`rsv2`	: reserved field 2
Outputs	None	
Returns	None	
Valid States	`NBCS_ACTIVE`	
Side Effects	None	

Notifying the Application of PRGM events: cbackPrgm

This callback function is provided by the user and is used by the CSD to report PRGM-related events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements. Note: If the address of the callback function was passed as a NULL at initialization, then no callback is made. The `event` field in the DPV is a bit mask that reports all the PRGM event(s) encountered. The `info` field in the DPV is encoded as the timeslot indicating which STS-1 path is causing the event(s).

Application is responsible for calling `sysNbcsDPVBufferRtn` to release the DPV buffer upon the return of this function.

Prototype

```
void cbackPrgm(sNBCS_USR_CTXT usrCtxt,
sNBCS_DPV *pdpv)
```

Inputs	`usrCtxt`	: user context (from `nbcsAdd`)
	`pdpv`	: (pointer to) DPV that describes this event
Outputs	None	
Returns	None	
Valid States	`NBCS_ACTIVE`	
Side Effects	None	

6 HARDWARE INTERFACE

The Narrowband Chipset driver does not interface directly with the user's hardware. Instead, it goes through the underlying SBS/NSE device drivers. Please refer to the SBS and NSE drivers' manual for specific porting instructions. It is the responsibility of the user to connect these requirements into the hardware, either by defining a macro or by writing a function for each item listed. For correct operation, parameters and return values must match those prototypes.

7 RTOS INTERFACE

The Narrowband Chipset driver requires the use of some Real-Time Operating System (RTOS) resources. In this section of the manual, a listing of each required resource is shown, along with a declaration and any specific porting instructions. It is the responsibility of the user to connect these requirements into the RTOS, either by defining a macro or by writing a function for each item listed. Care should be taken when matching parameters and return values.

7.1 Memory Allocation / De-Allocation

Allocating Memory: sysNbcsMemAlloc

This function allocates specified number of bytes of memory.

Format `#define sysNbcsMemAlloc(numBytes)`

Prototype `UINT1 * sysNbcsMemAlloc(UINT4 numBytes)`

Inputs `numBytes` : number of bytes to be allocated

Outputs None

Returns Success = Pointer to first byte of allocated memory
Failure = NULL pointer (memory allocation failed)

Freeing Memory: sysNbcsMemFree

This function frees memory allocated using `sysNbcsMemAlloc`.

Format `#define sysNbcsMemFree(pfirstByte)`

Prototype `void sysNbcsMemFree(UINT1 *pfirstByte)`

Inputs `pfirstByte` : pointer to first byte of the memory region being de-allocated

Outputs None

Returns None

7.2 Buffer Management

All operating systems provide some sort of buffer system, particularly for use in sending and receiving messages. The following calls, provided by the user, allow the driver to get and return buffers from the RTOS. It is the user's responsibility to create any special resources or pools to handle buffers of these sizes during the `sysNbcsBufferStart` call. These functions must be non-blocking.

Starting Buffer Management: sysNbcsBufferStart

This function alerts the RTOS to make sure buffer is available and sized correctly. Depending on the RTOS, this can involve the creation of new buffer pools.

Format `#define sysNbcsBufferStart()`

Prototype `INT4 sysNbcsBufferStart(void)`

Inputs None

Outputs None

Returns Success = 0
Failure = <any other value>

Getting a DPV Buffer: sysNbcsDPVBufferGet

This function gets a buffer from the RTOS.

Format `#define sysNbcsDPVBufferGet()`

Prototype `void * sysNbcsDPVBufferGet(void)`

Inputs None

Outputs None

Returns Success = (pointer to) a buffer
Failure = NULL (pointer)

Returning a DPV Buffer: sysNbcsDPVBufferRtn

This function returns a buffer to the RTOS when the information in the block is no longer needed.

Format `#define sysNbcsDPVBufferRtn(pBuf)`

Prototype `void sysNbcsDPVBufferRtn(void *pBuf)`

Inputs `pBuf` : (pointer to) a buffer

Outputs None

Returns None

Stopping Buffer Management: sysNbcsBufferStop

This function alerts the RTOS that the driver no longer needs any buffers and that if any special resources were created to handle these buffers, they can now be deleted.

Format `#define sysNbcsBufferStop()`

Prototype `void sysNbcsBufferStop(void)`

Inputs None

Outputs None

Returns None

7.3 Timers

Creating a Timer: sysNbcsTimerCreate

This function creates a timer object for general use.

Format `#define sysNbcsTimerCreate()`

Prototype `void * sysNbcsTimerCreate(void)`

Inputs None

Outputs None

Returns Success = (pointer to) a timer object
Failure = NULL (pointer)

Starting a Timer: sysNbcsTimerStart

This function starts a timer.

Format `#define sysNbcsTimerStart(ptimer, period, pfunc)`

Prototype
```
INT4 sysNbcsTimerStart(void *ptimer, UINT4
period, void *pfunc)
```

Inputs	`ptimer`
	`period`
	`pfunc`
		: (pointer to) timer object
		: time (in milliseconds)
		: function to invoke when timer expires

Outputs	None	
Returns	Success = 0 Failure = <any other value>	

Aborting a Timer: sysNbcsTimerAbort

This function aborts a running timer.

Format	`#define sysNbcsTimerAbort(ptimer)`	
Prototype	`INT4 sysNbcsTimerAbort(void *ptimer)`	
Inputs	`ptimer`	: (pointer to) timer object
Outputs	None	
Returns	Success = 0 Failure = <any other value>	

Deleting a Timer: sysNbcsTimerDelete

This function deletes a timer.

Format	`#define sysNbcsTimerDelete(ptimer)`	
Prototype	`void sysNbcsTimerDelete(void *ptimer)`	
Inputs	`ptimer`	: (pointer to) timer object
Outputs	None	
Returns	None	

Suspending a Task: sysNbcsTimerSleep

This function suspends execution of a driver task for a specified number of milliseconds.

Format	`#define sysNbcsTimerSleep(msec)`	
Prototype	`void sysNbcsTimerSleep(UINT4 msec)`	
Inputs	`msec`	: sleep time in milliseconds

Outputs None

Returns None

7.4 Semaphores

Creating a Semaphore: sysNbcsSemCreate

This function creates a binary semaphore object.

Format `#define sysNbcsSemCreate()`

Prototype `void * sysNbcsSemCreate(void)`

Inputs None

Outputs None

Returns Success = (pointer to) a semaphore object
Failure = NULL (pointer)

Taking a Semaphore: sysNbcsSemTake

This function takes a binary semaphore.

Format `#define sysNbcsSemTake(psem)`

Prototype `INT4 sysNbcsSemTake(void *psem)`

Inputs `psem` : (pointer to) a semaphore object

Outputs None

Returns Success = 0
Failure = <any other value>

Giving a Semaphore: sysNbcsSemGive

This function gives a binary semaphore.

Format `#define sysNbcsSemGive(psem)`

Prototype `INT4 sysNbcsSemGive(void *psem)`

Inputs `psem` : (pointer to) a semaphore object

Outputs None

Returns Success = 0
Failure = <any other value>

Deleting a Semaphore: sysNbcsSemDelete

This function deletes a binary semaphore object.

Format `#define sysNbcsSemDelete(psem)`

Prototype `void sysNbcsSemDelete(void *psem)`

Inputs `psem` : (pointer to) a semaphore object

Outputs None

Returns None

7.5 Preemption

Disabling Preemption: sysNbcsPreemptDisable

This routine prevents the calling task from being preempted. If the driver is in interrupt mode, this routine locks out all interrupts as well as other tasks in the system. If the driver is in polling mode, this routine locks out other tasks only.

Format `#define sysNbcsPreemptDisable()`

Prototype `INT4 sysNbcsPreemptDisable(void)`

Inputs None

Outputs None

Returns Preemption key (passed back as an argument in `sysNbcsPreemptEnable`)

Re-Enabling Preemption: sysNbcsPreemptEnable

This routine allows the calling task to be preempted. If the driver is in interrupt mode, this routine unlocks all interrupts and other tasks in the system. If the driver is in polling mode, this routine unlocks other tasks only.

Format `#define sysNbcsPreemptEnable(key)`

Prototype	`void sysNbcsPreemptEnable(INT4 key)`	
Inputs	`key`	: preemption key (returned by `sysNbcsPreemptDisable`)
Outputs	None	
Returns	None	

8 PORTING THE NARROWBAND CHIPSET DRIVER

This section outlines how to port the Narrowband Chipset driver to your hardware and RTOS platform. However, this document can offer only guidelines for porting the Narrowband Chipset driver as each platform and application is unique.

8.1 Driver Source Files

The C source files listed in the following table contain the code for the Narrowband Chipset driver. You may need to either modify the existing code or develop additional code. The code is in the form of constants, macros, and functions. For the ease of porting, the code is grouped into source files (`src`) and header files (`inc`). The `src` files contain the functions and the `inc` files contain the constants and macros.

Directory	File	Description
src	nbcs_api.c	Device and module management
	nbcs_fmgt.c	Fabric management block functions
	nbcs_stsw.c	Space/time switch block functions
	nbcs_diag.c	Diagnostics functions
	nbcs_lkc.c	LVDS link controller block functions
	nbcs_ilc.c	In-band link controller block functions
	nbcs_prgm.c	PRGM block functions
	nbcs_intf.c	Interface/Clock configuration functions
	nbcs_evt.c	Event processing functions
	nbcs_rtos.c	RTOS specific functions
	nbcs_stats.c	Status and counts functions
	nbcs_util.c	Miscellaneous functions
	nbcs_dal_sbsnse.c	DAL implementation for SBS and NSE devices
inc	nbcs_api.h	API function prototypes

Directory	File	Description
	nbcs_defs.h	Constants, macros and enumerated types
	nbcs_err.h	Driver error codes
	nbcs_fns.h	Non-API function prototypes
	nbcs_rtos.h	RTOS specific constants, macros and function prototypes
	nbcs_strs.h	Driver structures
	nbcs_typs.h	Standard types
	nbcs_dal.h	DAL prototypes
example	nbcs_app.c	Example callback functions and example code
	nbcs_app.h	Prototype and definitions for the example code
	nbcs_debug.c	Example debug code for reporting register accesses to the device
	nbcs_debug.h	Prototype and definitions for the debug code
	nbcs_profile.c	Example profiles
	nbcs_dal_null.c	Empty DAL functions

8.2 Driver Porting Procedures

The following procedures summarize how to port the Narrowband Chipset driver to your platform.

To port the Narrowband Chipset driver to your platform:

Step 1: Port the driver's RTOS extensions (page 175)

Step 2: Port the driver's application-specific elements (page 176)

Step 3: Build the driver (page 177)

Step 1: Porting Driver RTOS Extensions

The RTOS extensions encapsulate all RTOS specific services and data types used by the driver. These RTOS extensions include:

- Memory Management
- Task management
- Message queues, semaphores and timers

The compiler-specific data type definitions are located in `nbcs_typs.h`. The files `nbcs_rtos.h` and `nbcs_rtos.c` contain macros and functions for RTOS specific services.

To port the driver's RTOS extensions:

1. Modify the data types in `nbcs_typs.h`. The number after the type identifies the data-type size. For example, UINT4 defines a 4-byte (32-bit) unsigned integer. Substitute the compiler types that yield the desired types as defined in this file.

2. Modify the RTOS specific macros in `nbcs_rtos.h`:

Service Type	Macro Name	Description
Memory	`sysNbcsMemAlloc`	Allocates the memory block
	`sysNbcsMemFree`	Frees the memory block
	`sysNbcsMemCpy`	Copies the memory block from src to dest
	`sysNbcsMemSet`	Sets each character in the memory buffer
Semaphores	`sysNbcsSemCreate`	Creates a semaphore
	`sysNbcsSemTake`	Takes a semaphore
	`sysNbcsSemGive`	Gives a semaphore
	`sysNbcsSemDelete`	Deletes a semaphore

3. Modify the RTOS specific functions in `nbcs_rtos.c`:

Service Type	Function Name	Description
Timer	`sysNbcsTimerSleep`	Sleeps a task
	`sysNbcsTimerCreate`	Creates a timer

Service Type	Function Name	Description
	sysNbcsTimerStart	Starts a timer
	sysNbcsTimerAbort	Aborts a timer
	sysNbcsTimerDelete	Deletes a timer
Buffer	sysNbcsBufferStart	Start buffer management
	sysNbcsDPVBufferGet	Gets a DPV buffer
	sysNbcsDPVBufferRtn	Returns a DPV buffer
	sysNbcsBufferStop	Stops buffer management
Preemption	sysNbcsPreemptDisable	Disables preemption
	sysNbcsPreemptEnable	Enables preemption

Step 2: Porting Driver Application-Specific Elements

Application specific elements are configuration constants used by the API for developing an application. This section describes how to modify the application specific elements in the Narrowband Chipset driver.

To port the driver's application-specific elements:

1. Modify the type definition for the user context in `nbcs_typs.h`. The user context is used to identify a device in your application callbacks.

2. Modify the value of the base error code (`NBCS_ERR_BASE`) in `nbcs_err.h`. This ensures that the driver error codes do not overlap with other error codes used in your application.

3. Define the application-specific constants for your hardware configuration in `nbcs_defs.h`:

Device Constant	Description	Default
NBCS_MAX_SBS	The maximum number of SBS devices that can be supported by this driver	32
NBCS_MAX_NSE	The maximum number of NSE devices that can be supported by this driver	5

NBCS_MAX_SBS_INIT_PROFS	The maximum number of SBS initialization profiles	5
NBCS_MAX_NSE_INIT_PROFS	The maximum number of NSE initialization profiles	5
NBCS_MAX_GROUP	The maximum number of groups allowed in the driver	7
NBCS_MAX_MCAST	The maximum number of destinations in a multicast connection attempt	32

4. Define the following application-specific constants for your RTOS-specific services in `nbcs_rtos.h`:

Task Constant	Description	Default
NBCS_MAX_DPV_BUF	The maximum number of DPV buffers	950

5. Code the callback functions according to your application. There are sample callback functions in `nbcs_app.c`. You can use these callback functions or you can customize them before using the driver. The driver will call these callback functions when an event occurs on the device. These functions must conform to the following prototype (`cback` should be replaced with your callback function name):

```
void cback(sNBCS_USR_CTXT usrCtxt, sNBCS_DPV *pdpv)
```

Step 3: Building the Driver

This section describes how to build the Narrowband Chipset driver.

To build the driver:

1. Modify the `Makefile` to reflect the absolute path of your code, your compiler and compiler options.
2. Choose from among the different compile options supported by the driver as per your requirements.
3. Compile the source files and build the Narrowband Chipset API driver library using your make utility.
4. Link the Narrowband Chipset API driver library to your application code.

APPENDIX A: CODING CONVENTIONS

This section of the manual describes the coding conventions used to implement PMC chipset driver software.

Variable Type Definitions

Table 59: Variable Type Definitions

Type	Description
UINT1	unsigned integer value of size 1 byte (0x0 – 0xFF)
UINT2	unsigned integer value of size 2 bytes (0x0 – 0xFFFF)
UINT4	unsigned integer value of size 4 bytes (0x0 – 0xFFFFFFFF)
INT1	signed integer value of size 1 byte (0x0 – 0xFF)
INT2	signed integer value of size 2 bytes (0x0 – 0xFFFF)
INT4	signed integer value of size 4 bytes (0x0 – 0xFFFFFFFF)

Naming Conventions

Table 60 summarizes the naming conventions followed by PMC-Sierra driver software. Detailed descriptions are then provided in the following sub-sections.

The names used in the drivers are detailed enough to make their purpose fairly clear. Please note that the device name appears in prefix.

Table 60: Naming Conventions

Type	Naming convention	Examples
Macros	Uppercase, prefix with "m" and device abbreviation	mNBCS_SLICE_OFFSET mNBCS_QE_VC_NUM mNBCS_[BLK]_<PURPOSE>

Type	Naming convention	Examples
Enumerated Types	Uppercase, prefix with "e" and device abbreviation	eNBCS_MOD_STATE eNBCS_DEV_STATE eNBCS_<OBJECT>
Constants	Uppercase, prefix with device abbreviation	NBCS_SUCCESS NBCS_BITMSK_RESET NBCS_BITOFF_BIP8 NBCS_[CATEGORY]_<OBJECT>
Structures	Uppercase, prefix with "s" and device abbreviation	sNBCS_CSDDB sNBCS_CNTR_LOH sNBCS_MASK_ISR sNBCS_STATUS_QE_VC sNBCS_<PURPOSE>_<BLK>_[OBJECT]
API Functions	Hungarian notation, prefix with device abbreviation	nbcsAdd() nbcsStatsGetStatus() nbcsStatsGetCountsXX() nbcsDiagTestReg() nbcs[Blk]<Action>[Object]()
Porting Functions and Macros	Hungarian notation, prefix with "sys" and device abbreviation	sysNbcsRead() sysNbcsBufferGet() sysNbcs[Object]<Action>()
Non-API Functions	Hungarian notation	utilNbcsReset() <blk>Nbcs<Action>[Object]()
Variables	Hungarian notation	maxDevs
Pointers to variables	Hungarian notation, prefix variable name with "p"	pmaxDevs

Type	Naming convention	Examples
Global variables	Hungarian notation, prefix with device abbreviation	`nbcsCsmdb`

File Organization

Table 61 presents a summary of the file naming conventions. All file names must start with the device abbreviation, followed by an underscore and the actual file name. File names convey their purpose with a minimum number of characters.

Table 61: File Naming Conventions

File Type	File Name	Description
API (Module and Device Management)	`nbcs_api.c`	Generic driver API block, contains Module & Device Management API such as installing/de-installing driver instances, read/writes, and initialization profiles. Contains functions such as `nbcsModuleOpen`, `nbcsModuleStart`, `nbcsAdd`.
API (Events)	`nbcs_evt.c`	Event processing is handled by this block. This includes interrupt callback function management Contains functions such as `nbcsEventSetMask`, and `nbcsEventClearMask`.
API (Diagnostics)	`nbcs_diag.c`	Contains device diagnostic functions such as `nbcsDiagTestReg`, `nbcsDiagTestRam`.
API (Interface/Clock Configuration)	`nbcs_intf.c`	Interface/Clock configuration functions for connecting the device to external interfaces (i.e.,PHY,PL3,UL2,SBI, TeleCombus, clk/data, LVDS, etc.)

File Type	File Name	Description
API (Status and counts)	nbcs_stats.c	Data collection block for all device results/counts. Contains nbcsStatsGetStatus, nbcsStatsGetCounts. Functions in this file perform basic state, error checks and retrieve block specific status and counts.
API (Device specific blocks)	nbcs_ilc.c, nbcs_prgm.c, nbcs_stsw.c, nbcs_lkc.c, nbcs_fmgt.c,	Device specific configuration functions defined in the driver architecture. Both API and functions used internally by driver are located in these files.
DAL (SBS/NSE implementation)	nbcs_dal_sbsnse.c	DAL implementation for use with SBS and NSE device drivers
RTOS Dependent	nbcs_rtos.c, nbcs_rtos.h	RTOS specific functions such as sysNbcsBufferGet, sysNbcsDPVBufferRtn, RTOS constants and macros
Other	nbcs_util.c	Utility functions used internally by the driver (i.e. utilNbcsResetDev)
Header file	nbcs_api.h	Prototypes for all the API functions of the driver
Header file	nbcs_err.h	Error return codes
Header file	nbcs_defs.h	Device constants and macros, register offset definitions, bit masks, enumerated types
Header file	nbcs_typs.h	Standard types definition (i.e., UINT1, UINT2, etc.)
Header file	nbcs_fns.h	Prototypes for all the non-API functions used in the driver
Header file	nbcs_strs.h	All structure definitions

APPENDIX B: NARROWBAND CHIPSET ERROR CODES

This appendix describes the error codes used in the Narrowband Chipset device driver.

Table 62: Narrowband Chipset Error Codes

Error Code	Description
NBCS_SUCCESS	Success
NBCS_FAILURE	Failure
NBCS_ERR_MEM_ALLOC	Memory allocation failure
NBCS_ERR_INVALID_ARG	Invalid argument
NBCS_ERR_INVALID_SYS_CONFIG	Invalid system configuration
NBCS_ERR_INVALID_GROUP	Invalid group ID
NBCS_ERR_DEV_ABSENT	Device is not present locally
NBCS_ERR_INVALID_MODULE_STATE	Invalid module state
NBCS_ERR_INVALID_MIV	Invalid Module Initialization Vector
NBCS_ERR_PROFILES_FULL	Maximum number of profiles already added
NBCS_ERR_INVALID_PROFILE	Invalid profile
NBCS_ERR_INVALID_PROFILE_NUM	Invalid profile number
NBCS_ERR_INT_INSTALL	Error while installing interrupts
NBCS_ERR_BUF_START	Error while starting buffer management
NBCS_ERR_INVALID_DEVICE_STATE	Invalid device state
NBCS_ERR_DEVS_FULL	Maximum number of devices already added
NBCS_ERR_DEV_ALREADY_ADDED	Device already added
NBCS_ERR_INVALID_DEV	Invalid device handle or device ID
NBCS_ERR_INVALID_DIV	Invalid Device Initialization Vector
NBCS_ERR_INVALID_MODE	Invalid ISR/polling mode

Error Code	Description
NBCS_ERR_INVALID_GROUP_STATE	Invalid group state
NBCS_ERR_GROUPS_FULL	No more groups are available
NBCS_ERR_ADDING_DEVICE_IN_GROUP	Error adding device to group
NBCS_ERR_DELETING_DEVICE_IN_GROUP	Error deleting device from group
NBCS_ERR_INVALID_REG	Invalid register number
NBCS_ERR_POLL_TIMEOUT	Time-out while polling
NBCS_ERR_INVALID_BUS_TYPE	Invalid bus type
NBCS_ERR_CSU_LOCK	CSU lock failure in devices is detected
NBCS_ERR_GROUPS_MIXED_DEV	Mixed devices are found in group
NBCS_ERR_STSW_ACCESS	Error accessing STSW blocks
NBCS_ERR_ILC_TX_TIMEOUT	Tx ILC timeout
NBCS_ERR_ILC_INVALID_OP	Invalid operation in ILC is received
NBCS_ERR_OPA_PROTECT_EXIST	Protection scheme exists already
NBCS_ERR_OPA_PROTECT_NONEXISTENT	Protection scheme does not exist
NBCS_ERR_OPA_PROTECT_1FORN	Error in 1:N Port Protection scheme
NBCS_ERR_OPA_CONNECT	Error in setting up tributary/byte connection
NBCS_ERR_OPA_DISCONNECT	Error in disconnecting tributary/byte connection
NBCS_ERR_INVALID_TRIB	Invalid tributary
NBCS_ERR_INVALID_PYLD	Invalid payload type
NBCS_ERR_INVALID_WIRING	Invalid physical wiring
NBCS_ERR_INVALID_SWITCHOVER	Invalid port or path level switchover

Error Code	Description
NBCS_ERR_PROTECT_BUSY	Protection Port in 1:N port protection scheme is currently used and not available
NBCS_ERR_OPA_SCHEDULE	Cannot schedule a call due to lack of resources

APPENDIX C: NARROWBAND CHIPSET EVENTS

This appendix describes the events used in the Narrowband Chipset device driver.

Table 63: Narrowband Chipset Events for PRGM Callbacks

Event Code	Description	Relevant Information
NBCS_EVENT_PRGM_BYTEERR1	PRGM byte error in timeslice #1	This event may combine with other PRGM events to form an event bitmask
NBCS_EVENT_PRGM_BYTEERR2	PRGM byte error in timeslice #2	This event may combine with other PRGM events to form an event bitmask
NBCS_EVENT_PRGM_BYTEERR3	PRGM byte error in timeslice #3	This event may combine with other PRGM events to form an event bitmask
NBCS_EVENT_PRGM_BYTEERR4	PRGM byte error in timeslice #4	This event may combine with other PRGM events to form an event bitmask
NBCS_EVENT_PRGM_BYTEERR5	PRGM byte error in timeslice #5	This event may combine with other PRGM events to form an event bitmask
NBCS_EVENT_PRGM_BYTEERR6	PRGM byte error in timeslice #6	This event may combine with other PRGM events to form an event bitmask
NBCS_EVENT_PRGM_BYTEERR7	PRGM byte error in timeslice #7	This event may combine with other PRGM events to form an event bitmask
NBCS_EVENT_PRGM_BYTEERR8	PRGM byte error in timeslice #8	This event may combine with other PRGM events to form an event bitmask
NBCS_EVENT_PRGM_BYTEERR9	PRGM byte error in timeslice #9	This event may combine with other PRGM events to form an event bitmask
NBCS_EVENT_PRGM_BYTEERR10	PRGM byte error in timeslice #10	This event may combine with other PRGM events to form an event bitmask

Event Code	Description	Relevant Information
NBCS_EVENT_PRGM_BYTEERR11	PRGM byte error in timeslice #11	This event may combine with other PRGM events to form an event bitmask
NBCS_EVENT_PRGM_BYTEERR12	PRGM byte error in timeslice #12	This event may combine with other PRGM events to form an event bitmask
NBCS_EVENT_PRGM_SYNC1	PRGM synchronization error in timeslice #1	This event may combine with other PRGM events to form an event bitmask
NBCS_EVENT_PRGM_SYNC2	PRGM synchronization error in timeslice #1	This event may combine with other PRGM events to form an event bitmask
NBCS_EVENT_PRGM_SYNC3	PRGM synchronization error in timeslice #3	This event may combine with other PRGM events to form an event bitmask
NBCS_EVENT_PRGM_SYNC4	PRGM synchronization error in timeslice #4	This event may combine with other PRGM events to form an event bitmask
NBCS_EVENT_PRGM_SYNC5	PRGM synchronization error in timeslice #5	This event may combine with other PRGM events to form an event bitmask
NBCS_EVENT_PRGM_SYNC6	PRGM synchronization error in timeslice #6	This event may combine with other PRGM events to form an event bitmask
NBCS_EVENT_PRGM_SYNC7	PRGM synchronization error in timeslice #7	This event may combine with other PRGM events to form an event bitmask
NBCS_EVENT_PRGM_SYNC8	PRGM synchronization error in timeslice #8	This event may combine with other PRGM events to form an event bitmask
NBCS_EVENT_PRGM_SYNC9	PRGM synchronization error in timeslice #9	This event may combine with other PRGM events to form an event bitmask

Event Code	Description	Relevant Information
NBCS_EVENT_PRGM_SYNC10	PRGM synchronization error in timeslice #10	This event may combine with other PRGM events to form an event bitmask
NBCS_EVENT_PRGM_SYNC11	PRGM synchronization error in timeslice #11	This event may combine with other PRGM events to form an event bitmask
NBCS_EVENT_PRGM_SYNC12	PRGM synchronization error in timeslice #12	This event may combine with other PRGM events to form an event bitmask

Table 64: Narrowband Chipset Events for STSW Callbacks

Event Code	Description	Relevant Information
NBCS_EVENT_STSW_SWAP	connection page swap event	This event may combine with other STSW events to form an event bitmask
NBCS_EVENT_STSW_UPDATE	connection page update event	This event may combine with other STSW events to form an event bitmask

Table 65: Narrowband Chipset Events for LKC Callbacks

Event Code	Description	Relevant Information
NBCS_EVENT_LKC_TXFIFO_ERR	Transmit FIFO error event	This event may combine with other LKC events to form an event bitmask
NBCS_EVENT_LKC_RXFIFO_ERR	Receive FIFO error event	This event may combine with other LKC events to form an event bitmask
NBCS_EVENT_LKC_OCA	Out of character alignment event	This event may combine with other LKC events to form an event bitmask
NBCS_EVENT_LKC_OFA	Out of frame alignment event	This event may combine with other LKC events to form an event bitmask

Event Code	Description	Relevant Information
NBCS_EVENT_LKC_LCV	Link code violation event	This event may combine with other LKC events to form an event bitmask

Table 66: Narrowband Chipset Events for ILC Callbacks

Event Code	Description	Relevant Information
NBCS_EVENT_ILC_LINKCHG	LINK bit in ILC header changed	This event may combine with other ILC events to form an event bitmask
NBCS_EVENT_ILC_USER0CHG	USER[0] bit in ILC header changed	This event may combine with other ILC events to form an event bitmask
NBCS_EVENT_ILC_FIFO_OVERFLOW	Rx FIFO overflow	This event may combine with other ILC events to form an event bitmask
NBCS_EVENT_ILC_FIFO_THRES	Rx FIFO threshold is reached	This event may combine with other ILC events to form an event bitmask
NBCS_EVENT_ILC_FIFO_TIMEOUT	Rx FIFO timeout	This event may combine with other ILC events to form an event bitmask
NBCS_EVENT_ILC_PG0CHG	PG[0] ILC header bit changed	This event may combine with other ILC events to form an event bitmask
NBCS_EVENT_ILC_PG1CHG	PG[1] ILC header bit changed	This event may combine with other ILC events to form an event bitmask

Table 67: Narrowband Chipset Events for INTF Callbacks

Event Code	Description	Relevant Information
NBCS_EVENT_INTF_WORKING_FCA	False character alignment detected in working link (in SBS device)	This event may combine with other INTF events to form an event bitmask

Event Code	Description	Relevant Information
NBCS_EVENT_INTF_PROTECT_FCA	False character alignment detected in protect link (in SBS device)	This event may combine with other INTF events to form an event bitmask
NBCS_EVENT_INTF_CSU1LOCK	CSU#1 Lock is detected	This event may combine with other INTF events to form an event bitmask
NBCS_EVENT_INTF_CSU2LOCK	CSU#2 Lock is detected (in NSE devices only)	This event may combine with other INTF events to form an event bitmask
NBCS_EVENT_INTF_REFDLL_ERR	Reference DLL error detected	This event may combine with other INTF events to form an event bitmask
NBCS_EVENT_INTF_SYSDLL_ERR	System DLL error detected	This event may combine with other INTF events to form an event bitmask
NBCS_EVENT_INTF_RXBUS_PARITY_ERR	Receive bus parity error	This event may combine with other INTF events to form an event bitmask
NBCS_EVENT_INTF_INC_C1FP	Incoming C1FP detected	This event may combine with other INTF events to form an event bitmask
NBCS_EVENT_INTF_RX_C1FP	Receive C1FP detected	This event may combine with other INTF events to form an event bitmask
NBCS_EVENT_INTF_OUTBUS1_COLLISION	Outgoing bus#1 collision detected	This event may combine with other INTF events to form an event bitmask
NBCS_EVENT_INTF_OUTBUS2_COLLISION	Outgoing bus#2 collision detected	This event may combine with other INTF events to form an event bitmask
NBCS_EVENT_INTF_OUTBUS3_COLLISION	Outgoing bus#3 collision detected	This event may combine with other INTF events to form an event bitmask

Event Code	Description	Relevant Information
NBCS_EVENT_INTF_OUTBUS4_COLLISION	Outgoing bus#4 collision detected	This event may combine with other INTF events to form an event bitmask
NBCS_EVENT_INTF_INCBUS1_PARITY_ERR	Incoming bus#1 parity error detected	This event may combine with other INTF events to form an event bitmask
NBCS_EVENT_INTF_INCBUS2_PARITY_ERR	Incoming bus#2 parity error detected	This event may combine with other INTF events to form an event bitmask
NBCS_EVENT_INTF_INCBUS3_PARITY_ERR	Incoming bus#3 parity error detected	This event may combine with other INTF events to form an event bitmask
NBCS_EVENT_INTF_INCBUS4_PARITY_ERR	Incoming bus#4 parity error detected	This event may combine with other INTF events to form an event bitmask

APPENDIX D: NARROWBAND CHIPSET INITIALIZATION PROFILES

The chipset module initialization profiles provide the user with a convenient way of setting up common setups in PMC device drivers. This appendix covers the CSD initialization in the context of centralized TeleCombus and SBI336 bus operation. Example MIVs, DIVs and GIVs will be presented for those system configurations and the code can be found in `example/nbcs_profile.c`. These profiles are built into subroutines that can be compiled as-is and used in the target application code. Additional profiles may be created by the user for other required applications.

For a detailed description of the module, device, and group initialization structures, please refer to page 63 where the MIV, DIV, and GIV are defined. Also please refer to section 5.1 for a description of the profile usage. Initialization profile management are described on page 104.

Centralized TeleCombus Application

All SBS and NSE devices are under the control of a single microprocessor. The OPA library is also activated. SBS devices are passing TeleCombus traffic. Path termination mode is HPT.

Module Initialization Vector: nbcsInitMivCentralTelecombus

This profile can be used to set the system in centralized TeleCombus mode with a 1-stage time-space-time fabric:

- Bus type is TeleCombus
- switching mode is column
- Both SBS and NSE device drivers are present
- standard fabric is assumed
- standard OPA scheduling is selected
- all connection map settings are automatically populated to the offline pages of the devices
- all connection map page switching is controlled by software
- all offline pages are automatically synchronized with the online connection page
- all working/protect link selection in SBS devices are controlled by hardware pin

SBS Device Initialization Vector: nbcsInitSbsDivHPT77

This SBS DIV sets the SBS to the following:

- single 77MHz incoming bus

- all timeslices are configured for HPT termination mode
- the multiframe is 4
- ILC threshold for all ports are 250 microseconds
- ILC FIFO threshold is 1 data unit

NSE Device Initialization Vector: nbcsNseDivHPT

This NSE DIV sets the NSE to the following:

- all timeslices are configured for HPT termination mode
- the ILC FIFO timeout is 250us
- the ILC FIFO threshold is 1 data unit

Centralized SBI Bus Application

All SBS and NSE devices are under the control of a single microprocessor. The OPA library is also activated. SBS devices are passing SBI bus traffic. Path termination mode is LPT.

Module Initialization Vector: nbcsInitMivCentralSbiByte

This profile can be used to set the system in centralized SBI bus byte mode with a 1-stage time-space-time fabric:

- Bus type is SBI
- switching mode is byte
- Both SBS and NSE device drivers are present
- standard fabric is assumed
- standard OPA scheduling is selected
- all connection map settings are automatically populated to the offline pages of the devices
- all connection map page switching is controlled by software
- all offline pages are automatically synchronized with the online connection page
- all working/protect link selection in SBS devices are controlled by hardware pin

SBS Device Initialization Vector: nbcs InitSbsDivLPT19

This SBS DIV sets the SBS to the following:

- quad 19.44MHz incoming bus
- all timeslices are configured for LPT termination mode

- the multiframe is 48
- ILC threshold for all ports are 250 microseconds
- ILC FIFO threshold is 1 data unit

NSE Device Initialization Vector: nbcsInitNseDivLPT

- all 12 timeslices are configured in LPT termination mode
- ILC threshold for all ports are 250 microseconds
- ILC FIFO threshold is 1 data unit

Distributed TeleCombus Core Card Application

The system assumes a distributed system model with only NSE device present locally. There is no local SBS device and the SBS driver is not required. The OPA module is hosted by the core card. The system assumes TeleCombus mode of operation. Path termination mode is HPT. ILC is assumed to be used as the primary mean of system page swapping.

Module Initialization Vector: nbcsInitMivDistCoreTelecombus

This profile can be used to set the system in a distributed TeleCombus mode with a 1-stage time-space-time fabric in a NSE core card:

- Bus type is TeleCombus
- switching mode is column
- Only NSE device driver is present and SBS driver is absent
- standard fabric is assumed
- standard OPA scheduling is selected
- all connection map settings are automatically populated to the offline pages of the devices
- all connection map page switching is controlled by ILC
- all offline pages are automatically synchronized with the online connection page
- all working/protect link selection in SBS devices are controlled by hardware pin

Distributed TeleCombus Line Card Application

The system assumes a distributed system model with SBS devices present locally. There is no local NSE device and the NSE driver is not required. The OPA module is also disabled in the line card. The system assumes TeleCombus mode of operation. Path termination mode is HPT. ILC is assumed to be used as the primary mean of system page swapping.

Module Initialization Vector: nbcsInitMivDistLineTelecombus

This profile can be used to set the system in a distributed TeleCombus mode with a 1-stage time-space-time fabric in a line card:

- Bus type is TeleCombus
- switching mode is column
- Only SBS device driver is present and NSE driver is absent
- standard fabric is assumed
- standard OPA scheduling is selected
- all connection map settings are automatically populated to the offline pages of the devices
- all connection map page switching is controlled by ILC
- all offline pages are automatically synchronized with the online connection page
- all working/protect link selection in SBS devices are controlled by hardware pin

APPENDIX F: NARROWBAND CHIPSET DRIVER SYNCHRONIZATION

Overview

In a regular system, there should only be one CSD configured to run the OPA which keeps track of all the connections in the fabric. If a more fault tolerant system is to be designed, the hardware that runs the CSD/OPA may become a single point of failure. One possible approach to achieve a fault tolerant system is to maintain two independent copies of the CSD/OPA running in a working and protect hardware mechanism (Commands may be broadcast to both for concurrent processing). In the event of a card failure, the system can be switched over to the protect hardware (if the working hardware fails). The failed hardware can then be replaced without any service interruption. Once boot up and initialized, the new hardware can then be *synchronized* with the currently active hardware. The fault tolerant system is then fully restored.

The CSD/OPA keeps track of all the system-wide connections by maintaining internal states. This state information is updated whenever there are call connection/disconnection or switchover requests. A fault tolerant system will not be completely restored unless this state information can be fully duplicated in the new hardware. Such process is being defined as the CSD synchronization. The CSD provides API function to retrieve and restore the internal state of the software. The saved state of the software is sometimes referred to as a *checkpoint*.

The CSD/OPA includes an example of a *log-based with checkpointing* recovery scheme. Prior to restoring the checkpoint of the CSD/OPA, the CSD should first be initialized. The system relies on an external repository to keep a log of all device and fabric initialization commands to the CSD (therefore it is called a log-based with checkpointing recovery scheme). During synchronization, the initialization command sequence is first “played back” to and then the checkpoint is restored in the new hardware. These information will then be the exact same state as the protect card, thus achieving synchronization.

The following outlines a typical event sequence before and after a failure recovery (assuming a failure in working fabric). The protection switchover procedure is described in (c) and the recovery procedure starts from step (d).

(a) Both working and protect card are brought up with the same device and fabric initialization command sequence initially. Subsequent call commands (such as setting up or tearing down tributaries, and protection switchovers) are always broadcast to both cards simultaneously. Assuming a reliable communication channel, the states of both cards are in synchronization.

(b) The working CSD is normally in control and incremental change in SBS settings are sent to remote line cards. Any SBS or NSE settings local to the switch card will be updated by the CSD.

(c) When the working fabric fails, a protection switchover occurs and all SBSs send and receive traffic via the protect LVDS links. The protect fabric becomes the master (and the lone card) in the system.

(d) The working fabric card is replaced. - Restart the replacement working fabric card by playing back the exact device and fabric initialization command sequence. This step ensures all parameters are properly written to the device registers and puts the system to a known initialized state. This essentially establishes the basic fabric mode of operation and allocates memory. User then initiates the state retrieval operation from the protect fabric card. The state information is either stored in a file system or some non-volatile memory. (The state retrieval may also be done periodically. The frequency is to be determined by the system designer.)

(e) The protect card is still in operation and may process new call request while the working card is restoring the state information. All new call requests subsequent to the checkpoint should also be queued up (by the user application) and played back to the working card after the state is restored though the queuing is optional to the system designer. The system will appear to be temporarily out of service to new call requests during this period of time if no queuing is implemented. In either case, all existing calls continue to be in service without any disruption.

(f) When the state is restored in the working card and there are no more pending calls, the state of the working and the protect fabric is in synchronization and user may switchover to the working card now. The redundant fabric system is fully restored and we are back at (a) again.

The following two functions outline the retrieval and restoration of checkpoint in a system. They are served solely as an example and the implementation can be found in the example code directory in the `nbcs_app.c` file.

Getting Checkpoint Information from the CSD: nbcsGetCheckPoint

This function retrieves the checkpoint information for the CSD, including the underlying OPA library. The information can then be used to restore (using API `nbcsSetCheckPoint`) the state of another CSD, thus achieving synchronization. It should be repeatedly called until no more data is returned and this condition is indicated by `*pbufSz` equals zero. The number and size of the buffer returned may vary and the exact information, including the order of those buffers being returned, should be presented to the other copy of the CSD unaltered.

Prototype	`INT4 nbcsGetCheckPoint(void* pbuf, UINT4* pbufSz)`	
Inputs	`pbuf`	: pointer to the buffer for holding checkpoint information
	`pbufSz`	: pointer to the buffer size
Outputs	`pbufSz`	: actual number of bytes written to the buffer `pbuf`.
Returns	Success =	`NBCS_SUCCESS`

	Failure =	`NBCS_FAILURE`
		`NBCS_ERR_INVALID_ARG`
		`NBCS_ERR_INVALID_MODULE_STATE`

Valid States	`NBCS_MOD_READY`	

Side Effects None

Setting Checkpoint Information in the CSD: nbcsSetCheckPoint

This function restores the checkpoint information in the CSD from another copy of the CSD (running in a different microprocessor space). The checkpoint information should be obtained from calling API `nbcsSetCheckPoint` (in another CSD). All the offline page settings for local devices will also be restored. It is user's responsibility to promote the offline to online page subsequently.

Prototype `INT4 nbcsSetCheckPoint(void* pbuf, UINT4* pbufSz)`

Inputs

Inputs	`pbuf`	: pointer to the buffer for holding checkpoint information
	`pbufSz`	: pointer to the buffer size

Outputs	`pbufSz`	: actual number of bytes written to the buffer `pbuf`.

Returns	Success =	`NBCS_SUCCESS`

	Failure =	`NBCS_FAILURE`
		`NBCS_ERR_INVALID_MODULE_STATE`
		`NBCS_ERR_STSW_ACCESS`
		`NBCS_ERR_INVALID_MODE`
		`NBCS_ERR_INVALID_ARG`
		`NBCS_ERR_INVALID_DEVICE_STATE`

Valid States `NBCS_MOD_READY`

Side Effects None

APPENDIX G: DRIVER ABSTRACTION LAYER (DAL)

This appendix describes the driver abstraction layer (DAL) between the CSD and the underlying device drivers. Acting as a “shim” layer between the CSD and the underlying device driver(s), the DAL can be viewed as a translation layer bridging the interface difference between the CSD and the low level drivers. When a CSD call is made by the upper layer application, the DAL dispatches the call to the appropriate underlying device driver(s) for the operation. In addition, the DAL deciphers messages from the device driver(s) to the CSD, e.g., ISR callback messages and error codes.

The purpose of the DAL is to decouple the CSD from the underlying device drivers. The reason is two-fold: (1) the CSD can more easily adapt to various system configurations. (2) allows porting of the CSD to any future device(s) that may provide similar time:space:time switching capabilities, as the SBS and NSE devices.

In a centralized configuration with SBS and NSE devices, the DAL is implemented to interact with both the SBS and NSE device drivers. In a distributed configuration where the SBS or NSE devices may be absent, the DAL can be implemented to exclude any of the calls to a device driver that is absent. For instance, in a core NSE card without any SBS devices, calls to the SBS driver can be implemented as “empty” functions that returns immediately upon invocation. No actual reference to any SBS driver calls is made.

The DAL lends itself to the porting of the CSD to any future devices that may provide similar time:space:time fabric capabilities. The CSD will be shielded from a change of underlying devices and changes are local to the DAL only. The CSD functionality can then be leveraged and reused in systems built with future switching devices.

The DAL is modeled after a generic time switch device driver and a space switch one. It comprises the space switch DAL and the time switch DAL. Arranged in logical blocks, the following sections describe the DAL interface.

DAL DATA STRUCTURES

This section describes the elements of the driver that configure and control its behavior. The constants, and structures that the DAL uses are listed.

Constants

The following enumerated constants are used in the DAL:

- `eNBCS_BUSTYPE_DAL: NBCS_INPUT_BUS, NBCS_OUTPUT_BUS, NBCS_TX_BUS and NBCS_RX_BUS:` define the input, output, serial LVDS transmit, and serial LVDS receive bus respectively for a time switch device.
- `eNBCS_SWH_ACCESSMODE_DAL: NBCS_SSWXFER_UNICAST, NBCS_SSWXFER_MULTICAST, NBCS_SSWXFER_TIMESLOT, NBCS_SSWXFER_MAP, NBCS_SSWXFER_STRTTHRU, NBCS_SSWXFER_INPORT and NBCS_SSWXFER_OUTPORT:` define all the access modes of both the time and space switch devices.

Data Structures

DAL Module Initialization Vector: MIV_DAL

Table 68: DAL Module Initialization Vector: sNBCS_MIV_DAL

Field Name	Field Type	Field Description
perrModule	INT4*	(pointer to) errModule (see description in the MDB)
maxDevs	UINT2	Maximum number of devices supported during this session
maxInitProfs	UINT2	Maximum number of initialization profiles

DAL Time/Space Switch Configuration: CFG_SWH_DAL

Table 69: DAL Time/Space Switch Configuration: sNBCS_CFG_SWH_DAL

Field Name	Field Type	Field Description
rc1Dly	UINT2	C1 frame pulse delay

Field Name	Field Type	Field Description
swMode.detailed	eNBCS_LKC_SWITCHMODE	switching mode for the space switch. This is a union member.
swMode.simplified	eNBCS_SWHMODE	switching mode for the time switch. This is a union member.
swapMode	eNBCS_CONMAP_CNTL	connection map swap mode
autoUpdate	UINT1	connection map automatic offline update from online page

DAL Space Switch Device Initialization Vector: DIV_SSW_DAL

Table 70: DAL Space Switch Device Initialization Vector: sNBCS_DIV_SSW_DAL

Field Name	Field Type	Field Description
valid	UINT2	Indicates that this structure is valid
pollISR	UINT1	Indicates the type of ISR / polling to do
cbackISRPageSwap	NBCS_CBACK_DAL	Address of the function to be called in the ISR when the C1 frame pulse interrupt is received.
cbackIntf	NBCS_CBACK_DAL	Address for the callback function for Interface/Clock events
cbackSsw	NBCS_CBACK_DAL	Address for the callback function for space switch events
cbackPort	NBCS_CBACK_DAL	Address for the callback function for I/O port events
cbackIlc	NBCS_CBACK_DAL	Address for the callback function for ILC events
swhCfg	sNBCS_CFG_SWH_DAL	switch configuration data structure
portCfg [NBCS_NSE_MAX_LINKS]	sNBCS_CFG_LKC	port configuration data structure
ilcCfg [NBCS_NSE_MAX_LINKS]	sNBCS_CFG_ILC	In-band link controller data structure

DAL Time Switch Device Initialization Vector: DIV_TSW_DAL

Table 71: DAL Time Switch Device Initialization Vector: sNBCS_DIV_TSW_DAL

Field Name	Field Type	Field Description
valid	UINT2	Indicates that this structure is valid
pollISR	UINT1	Indicates the type of ISR / polling to do
cbackIntf	NBCS_CBACK_DAL	Address for the callback function for Interface/Clock configuration events
cbackTsw	NBCS_CBACK_DAL	Address for the callback function for time switch events
cbackPgmc	NBCS_CBACK_DAL	Address for the callback function for PRGM events
cbackWplc	NBCS_CBACK_DAL	Address for the callback function for Working/Protect LVDS link events
cbackIlcRx	NBCS_CBACK_DAL	Address for the callback function for ILC events
pageSwapControlMode	eNBCS_CONMAP_CNTL	Source of control for the connection page switching in all SBSs:
linkControlMode	eNBCS_WPLINK_CNTL	Source of control for the working and protection LVDS link in all SBSs:
intfBusMode	sNBCS_CFG_BUSMODE	Bus mode configuration structure
telecomBusCfgFlag	UINT1	Set to logic High if TeleCombus is selected
outBusCfgParam	sNBCS_CFG_BUSPARAM	Outgoing TeleCombus parameters
txBusCfgParam	sNBCS_CFG_BUSPARAM	LVDS transmit TeleCombus parameters

DAL Space Switch Interface Control Structure: CTL_INTF_SSW_DAL

Table 72: DAL Space Switch Interface Control Structure: sNBCS_CTL_INTF_SSW_DAL

Field Name	Field Type	Field Description
csu1	sNBCS_CTL_CSU_DAL	CSU#1 control structure
csu2	sNBCS_CTL_CSU_DAL	CSU#2 control structure

DAL CSU Control Structure: CTL_CSU_DAL

Table 73: DAL CSU Control Structure: sNBCS_CTL_CSU_DAL

Field Name	Field Type	Field Description
reset	UINT1	1 – CSU is reset
lowPowerMode	UINT1	0 – normal mode, 1 – low power mode

DAL TeleCombus Configuration Structure: CFG_INTF_TCB _DAL

Table 74: DAL TeleCombus Configuration Structure: sNBCS_CFG_INTF_TCB _DAL

Field Name	Field Type	Field Description
j1Config	UINT2	Controls whether the C1FP signal is pulsed high during J1 byte for each of the 12 STS-1's. Bit 0 controls to STS-1 #1 and Bit 11 controls STS-1 #12 0 – No C1FP pulse on J1 byte 1 – C1FP pulse on J1 byte
v1Config	UINT2	Controls whether the C1FP signal is pulsed high during V1 byte for each of the 12 STS-1's. Bit 0 controls to STS-1 #1 and Bit 11 controls STS-1 #12 0 – No C1FP pulse on V1 byte 1 – C1FP pulse on V1 byte
h1PtrValue	UINT1	sets the value of the H1 pointer
h2PtrValue	UINT1	sets the value of the H2 pointer

Field Name	Field Type	Field Description
altH1PtrValue	UINT1	sets alternate value of H1 pointer
altH2PtrValue	UINT1	sets alternate value of the H2 pointers
h1h2PtrSel	UINT2	selects whether H1, H2 pointer value or the alternate H1,H2 pointer values is inserted on each of the 12 STS-1's. Bit 0 controls to STS-1 #1 and Bit 11 controls STS-1 #12 0 – H1, H2 pointer values used 1 – Alternate H1, H2 values used
h1h2EnableFlag	UINT1	0 - H1, H2 values are not inserted 1 - H1, H2 values are inserted

DAL Interface Bus Configuration Structure: CFG_INTF_BUSPARM_DAL

Table 75: DAL Interface Bus Configuration Structure: sNBCS_CFG_INTF_BUSPARM_DAL

Field Name	Field Type	Field Description
oddParityFlag	UINT1	0 = even parity, 1 = odd parity
includePl	UINT1	Controls whether the PL signal is included in calculating the parity. 0 – not included, 1 – included. (For telecom bus only)
includeC1fp	UINT1	Controls whether the C1FP signal is included in calculating the parity. 0 – not included, 1 – included. (For telecom bus only)
j1ByteLock	UINT1	controls the position of the J1 byte in telecom bus mode. 0 – J1 byte locked to offset 0 1 – J1 byte locked to offset 522.

DAL Interface Bus Mode Structure: CFG_BUSMODE_DAL

Table 76: DAL Interface Bus Mode Structure: sNBCS_CFG_BUSMODE_DAL

Field Name	Field Type	Field Description
busType	eNBCS_BUSTYPE	System bus type: SBI or TeleCombus

Field Name	Field Type	Field Description
io	eNBCS_IO_BUSMODE	single bus or quad bus mode
bridge	UINT1	Bridge mode: 0 = serial LVDS in SBS enabled, 1 = serial LVDS disabled and parallel bus I/O is enabled.
multiFrm	eNBCS_MULTIFRM_MODE	Multi-frame mode: NBCS_MF_4 = 4 frames in multi-frame, NBCS_MF_48 = 48 frames in multi-frame
phyDevice	UINT1	SBI physical/link layer device mode: 0 = link layer device, 1 = physical layer device

SPACE SWITCH DEVICE DRIVER INTERFACE

This section describes the DAL interface for a generic space switch device driver such as a NSE-20G or NSE-8G device. The module and device management block, Interface/Clock, Status/Counts and Diagnostics blocks are standard blocks that encapsulate that of a typical PMC device driver. The LVDS controller, In-band link Controller, and Space Switch configuration blocks are logical blocks that are specific to a typical space switch device.

Module and Device Management

The module and device management block connects with that of the underlying driver.

Opening the Space Switch Driver Module: dalNbcsSswModuleOpen

Performs module level initialization of the space switch device driver by calling the underlying module open function provided by the space switch driver. This usually involves allocating all of the memory required by the driver and initializing the internal structures.

Prototype `INT4 dalNbcsSswModuleOpen(sNBCS_MIV_DAL *pmiv)`

Inputs `pmiv` : (pointer to) Module Initialization Vector

Outputs Places the address of `errModule` into the MIV passed by the Application.

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Closing the Space Switch Driver Module: dalNbcsSswModuleClose

Performs module level shutdown of the space switch driver. This involves deleting all devices being controlled by the driver and freeing all the memory allocated by the driver.

Prototype `INT4 dalNbcsSswModuleClose(void)`

Inputs None

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Starting the Space Switch Driver Module: dalNbcsSswModuleStart

Starts the module of the underlying space switch driver. Upon successful return from this function, the driver is ready to add devices.

Prototype `INT4 dalNbcsSswModuleStart(void)`

Inputs None

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Stopping the Space Switch Driver Module: dalNbcsSswModuleStop

Stops the module in the underlying time switch driver.

Prototype `INT4 dalNbcsSswModuleStop(void)`

Inputs None

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Adding a Device: dalNbcsSswAdd

Invokes the native device add function supplied by the space switch driver. The error device pointer is returned along with the handle returned by the space switch driver.

Prototype

```
INT4 dalNbcsSswAdd(void* usrCtxt, void
baseAddr, void** pHndl, INT4 **pperrDevice)
```

Inputs

`usrCtxt` : user context for this device
`baseAddr` : base address of the device
`pHndl` : (pointer to) device handle
`pperrDevice` : (pointer to) an area of memory

Outputs

`pperrDevice` : (pointer to) errDevice (inside the DDB of the space switch driver)
`pHndl` : (pointer to) device handle

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Deleting a Device: dalNbcsSswDelete

This function is used to remove the specified device from the list of devices being controlled by the space switch driver. Deleting a device involves clearing the DDB for that device and releasing its associated device handle.

Prototype `INT4 dalNbcsSswDelete(void* deviceInfo)`

Inputs `deviceInfo` : device information handle

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Initializing a Device: dalNbcsSswInit

Invokes the device initialization function provided by the space switch driver using the DIV or the profile number. If the DIV is passed as a NULL the profile number is used. A profile number of zero indicates that all the register bits are to be left in their default state. Note that the profile number is ignored UNLESS the passed DIV is NULL.

Prototype

```
INT4 dalNbcsSswInit(void* deviceInfo,
sNBCS_DIV_SSW_DAL *pdiv, UINT2 profileNum)
```

Inputs
`deviceInfo` : device information handle
`pdiv` : (pointer to) Device Initialization Vector
`profileNum` : profile number (only used if `pdiv` is NULL)

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Updating the Configuration of a Device: dalNbcsSswUpdate

Updates the configuration of the device according to the DIV passed by the Application. The only difference between `dalNbcsSswUpdate` and `dalNbcsSswInit` is that no soft reset will be applied to the device. In addition, a profile number of zero is not allowed.

Prototype

```
INT4 dalNbcsSswUpdate(void* deviceInfo,
sNBCS_DIV_SSW_DAL *pdiv, UINT2 profileNum)
```

Inputs
`deviceInfo` : device information handle
`pdiv` : (pointer to) Device Initialization Vector
`profileNum` : profile number (only used if `pdiv` is

NULL)

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Resetting a Device: dalNbcsSswReset

Applies a software reset to the space switch device. This function is typically called before re-initializing the device (via `dalNbcsSswInit`).

Prototype `INT4 dalNbcsSswReset(void* deviceInfo)`

Inputs `deviceInfo` : device information handle

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Activating a Device: dalNbcsSswActivate

Restores the state of a device after a de-activate.

Prototype `INT4 dalNbcsSswActivate(void* deviceInfo)`

Inputs `deviceInfo` : device information handle

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

De-Activating a Device: dalNbcsSswDeActivate

De-activates the device from operation.

Prototype `INT4 dalNbcsSswDeActivate(void* deviceInfo)`

Inputs `deviceInfo` : device information handle

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Reading from Device Registers: dalNbcsSswRead

This function can be used to read a register of a specific space switch device by providing the register number. This function derives the actual address location based on the device handle and register number inputs. It then reads the contents of this address location

Prototype `INT4 dalNbcsSswRead(void* deviceInfo, UINT2 regNum, UINT4* pval)`

Inputs
`deviceInfo` : device information handle
`regNum` : register number
`pval` : pointer to the value read

Outputs `pval` : pointer to the value read

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Writing to Device Registers: dalNbcsSswWrite

This function can be used to write to a register of a specific space switch device by providing the register number. This function derives the actual address location based on the device handle and register number inputs. It then writes the data to the specified address location.

Prototype `INT4 dalNbcsSswWrite(void* deviceInfo, UINT2 regNum, UINT4 value)`

Inputs
`deviceInfo` : device information handle
`regNum` : register number
`value` : value to be written

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Reading from a block of Device Registers: dalNbcsSswReadBlock

This function can be used to read a register block of a specific space switch device by providing the starting register number and the size to read. This function derives the actual start address location based on the device handle and starting register number inputs. It then reads the contents of this data block.

Prototype `INT4 dalNbcsSswReadBlock(void* deviceInfo, UINT2 startRegNum, UINT2 size, UINT4 *pblock)`

Inputs
`deviceInfo` : device information handle
`startRegNum` : starting register number
`size` : size of the block to read

	pblock	: (pointer to) the block to read
Outputs	pblock	: (pointer to) the block read
Returns	Success = NBCS_SUCCESS Failure = <NBCS error codes>	

Writing to a Block of Device Registers: dalNbcsSswWriteBlock

This function can be used to write to a register block of a specific space switch device by providing the starting register number and the block size. This function derives the actual starting address location based on the device handle and starting register number inputs. It then writes the contents of this data block to the starting address location.

Prototype

```
INT4 dalNbcsSswWriteBlock(void* deviceInfo,
UINT2 startRegNum, UINT2 size, UINT4 *pblock,
UINT4 *pmask)
```

Inputs	deviceInfo	: device information handle
	startRegNum	: starting register number
	size	: size of block to read
	pblock	: (pointer to) block to write
	pmask	: (pointer to) mask
Outputs	None	
Returns	Success = NBCS_SUCCESS Failure = <NBCS error codes>	

Adding an Initialization Profile: dalNbcsSswAddInitProfile

Creates an initialization profile that is stored by the driver. A device can be initialized by passing the initialization profile number to dalNbcsSswInit.

Prototype

```
INT4 dalNbcsSswAddInitProfile(sNBCS_DIV_SSW_DAL
*pProfile, UINT2 *pProfileNum)
```

Inputs	pProfile	: (pointer to) initialization profile being added
	pProfileNum	: (pointer to) profile number to be assigned by the driver
Outputs	pProfileNum	: profile number assigned by the driver
Returns	Success = NBCS_SUCCESS Failure = <NBCS error codes>	

Getting an Initialization Profile: dalNbcsSswGetInitProfile

Gets the content of an initialization profile given its profile number.

Prototype `INT4 dalNbcsSswGetInitProfile(UINT2 profileNum, sNBCS_DIV_SSW_DAL *pProfile)`

Inputs
`profileNum` : initialization profile number
`pProfile` : (pointer to) initialization profile

Outputs `pProfile` : contents of the corresponding profile

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Deleting an Initialization Profile: dalNbcsSswDeleteInitProfile

Deletes an initialization profile given its profile number.

Prototype `INT4 dalNbcsSswDeleteInitProfile(UINT2 profileNum)`

Inputs `profileNum` : initialization profile number

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Interface/Clock Configuration

Getting/Setting Control: dalNbcsSswCntlIntf

Get/Set the control parameters for the interface/clock control block. This function can be used to reset one or both CSUs. This function can also be used to enable or disable one or both of the CSUs.

Prototype `INT4 dalNbcsSswCntlIntf(void* deviceInfo, UINT1 accMode, sNBCS_CTL_INTF_SSW_DAL *pcntl)`

Inputs
`deviceInfo` : device information handle
`accMode` : access control: 0 = get, 1 = set
`pcntl` : (pointer to) the control structure

Outputs `pcntl` : the control structure when `accMode` is 0

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Connection Switch Configuration

Configuring the Space Switch: dalNbcsSswCfgSwhParm

Get/Set configuration of the space switch. Parameters to configure include C1 delay, switching mode, swap mode, and page copy auto update.

Prototype `INT4 dalNbcsSswCfgSwhParm(void* deviceInfo, UINT1 accMode, sNBCS_CFG_SWH_DAL *pconfig)`

Inputs
`deviceInfo` : device information handle
`accMode` : access control: 0 = get, 1 = set
`pconfig` : (pointer to) configuration structure

Outputs
`pconfig` : configuration structure when `accMode` is 0

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Setting Up Connections: dalNbcsSswMapSlot

Establish connections in the space switch. This mapping function can operate in seven different modes, namely unicast, multicast, timeslot, inport, outport, map, and straight through.

In unicast mode, connection between the first element pointed to by `pinport` is mapped to the first element indicated by `poutport` for the time instance indicated by the first element in `pslot`. Such operation repeats `numSlot` times for all the pairs. It is designed to set up multiple unicast connections in the switch.

In multicast mode, the first data pointed to by `pinport` is mapped to all the ports (total indicated by `numSlot`) indicated by `poutport` for the time instance indicated by the first element pointed to by `pslot`. It is geared towards setting up one-to-many connections in the switch.

In timeslot mode, this operation will take place for the timeslot indicated by the first element pointed to by `pslot` across all ports. Argument `pinport` is expected to be a pointer to an array of 32 inports. The first inport in the array will be mapped to outport #1, the second inport in the array mapped to outport #2, so on and so forth, until all 32 outports are mapped. `poutport` and numSlot are ignored in this mode. At first sight, this mode can be achieved using unicast mode but timeslot mode is designed to take advantage of the efficient access in the hardware and is the preferred mode over unicast mode if all accesses are restricted to one time instance across all ports.

Inport mode allows unicast connections to be established for the port indicated by the first element pointed to by `pinport` across multiple timeslots. This mode (and outport mode) are designed for application connection maps which are organized per-port (rather than per-timeslot as in the device). In this mode, argument `poutport` is expected to be a pointer to an array of outports. This mode can be used to establish connections on all timeslots or on a user specified set of timeslots. To set up connections on all timeslots, `pslot` should be set to NULL. (In this case, the number of elements expected in the `poutport` array is either 1080 (in TeleCombus/SBI column modes) or 9720 (in SBI DS0/CAS modes). The inport will be mapped to the first element in `poutport` in timeslot #0, to the second element in `poutport` for timeslot #1, etc. `numSlot` will be ignored in this mode.) To set up connections on a user specified set of timeslots, set `numSlot` to the number of timeslots, and pass the timeslot values in an array pointed to by the `pslot` parameter. The inport will be mapped to the first element in `poutport` in the timeslot indicated by the first element in `pslot`, to the second element in `poutport` for the timeslot indicated by the second element in `pslot`, etc.

Outport mode is similar to inport mode. In this case, unicast connections can be established for an outport across multiple timeslots. `pinport` is expected to be a pointer to an array of inports (one element for each timeslot). The outport is indicated by the first element pointed to by `poutport`. The values of `pslot` and `numSlot` are to be set as described in the preceding paragraph.

Map mode is to update the entire connection map. `pinport` is expected to have 1080n or 9720n elements in TeleCombus/SBI column mode and SBI DS0/CAS modes, respectively, where n the number of ports in the device. The order in the array (pointed to by `pinport`) should be as follows: inport0[0]…inport0[N-1] inport1[0]…inport1[N-1]…inportM[0]…inportM[N-1] where M = frame size - 1 and is 1079 in TeleCombus/SBI column or 9719 in SBI DS0/CAS mode and N = total number of ports which equals to 32. `pslot`, `poutport` and `numSlot` are all ignored in this mode.

Straight through mode provides a one-to-one direct mapping from input to output ports for each timeslot. In this mode, input port n is mapped to output port n for every port and timeslot. `pslot`, `poutport`, `pinport`, and `numSlot` are all ignored in this mode.

Whenever applicable, the range of timeslots is expected to be from 0-1079 and 0-9719 in the case of TeleCombus/SBI column mode and SBI DS0/CAS mode respectively.

Prototype

```
INT4 dalNbcsSswMapSlot(void* deviceInfo,
eNBCS_SWH_ACCESSMODE_DAL mode, UINT2 *pslot,
UINT1 *poutport, UINT1 *pinport, UINT4 numSlot)
```

Inputs

`deviceInfo`	: device information handle
`mode`	: access mode
`pslot`	: pointer to (array of) timeslot(s)
`poutport`	: pointer to (array of) out port(s)
`pinport`	: pointer to (array of) in port(s)
`numSlot`	: number of elements

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Getting Source Connections: dalNbcsSswGetSrcSlot

This function returns the inport(s) which map to the given outport(s).

In unicast or multicast mode, the inport mapped to the given outport in time instance `slot` will be returned in buffer pointed to by `pinport`.

In timeslot mode, all 32 inports, for the given timeslot `slot`, will be returned to a user-supplied buffer pointed to by `pinport`, large enough to hold all 32 ports. `outport` is ignored in this mode.

In map mode, the entire connection map is returned to the buffer supplied by the user via `pinport`. The order in the array is as follows: inport0[0]…inport0[N-1] inport1[0]…inport1[N-1]…inportM[0]…inportM[N-1] where M = frame size - 1 and is 1079 in TeleCombus/SBI column or 9719 in SBI DS0/CAS mode and N = total number of ports which equals to 32. `outport` and `slot` are ignored in this mode.

Inport, outport, and straight-through modes are invalid for this function.

Prototype

```
INT4 dalNbcsSswGetSrcSlot(sNBCS_HNDL
deviceHandle, eNBCS_SWH_ACCESSMODE mode, UINT2
slot, UINT1 outport, UINT1 *pinport)
```

Inputs

`deviceHandle`	: device handle
`mode`	: access mode
`slot`	: timeslot (0-1079 in TeleCombus/SBI column modes or 0-9719 in SBI DS0/CAS modes)
`outport`	: outport number, ignored in timeslot mode
`pinport`	: (pointer to) inport(s)

Outputs `pinport` : (pointer to) inport(s)

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Getting Active Page: dalNbcsSswGetActivePage

Get the active page in the space switch.

Prototype

```
INT4 dalNbcsSswGetActivePage(void* deviceInfo,
UINT1* pPage)
```

Inputs `deviceInfo` : device information handle

	`pPage`	: (pointer to) the active page number
Outputs	`pPage`	: the active page number
Returns	Success = NBCS_SUCCESS Failure = <NBCS error codes>	

Setting Active Page: dalNbcsSswSetActivePage

Set the active page in the space switch.

Prototype
```
INT4 dalNbcsSswSetActivePage(void* deviceInfo,
UINT1 pageNum)
```

Inputs `deviceInfo` : device information handle
`pageNum` : the active page number

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Updating Inactive Page: dalNbcsSswUpdateInactivePage

Copy the connection settings from active to inactive page. This function is designed for manual copy operation when automatic page copy is not activated.

Prototype
```
INT4 dalNbcsSswUpdateInactivePage(void*
deviceInfo)
```

Inputs `deviceInfo` : device information handle

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

LVDS Link Controller

Inserting line code violation: dalNbcsSswInsertLkcLcv

This function enables or disables the insertion of line code violations in the LVDS links

Prototype
```
INT4 dalNbcsSswInsertLkcLcv(void* deviceInfo,
UINT1 port, UINT1 enable)
```

Inputs `deviceInfo` : device information handle

`port` :the port number ranges 0-31
`enable` : 0 = disable, 1 = enable

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = <NBCS error codes>

Centering transmit FIFO: dalNbcsSswCenterLkcFifo

This function is used to center the transmit FIFO in the LVDS links.

Prototype
```
INT4 dalNbcsSswCenterLkcFifo(void* deviceInfo,
UINT1 port)
```

Inputs `deviceInfo` : device information handle
`port` : the port number ranges from 0-31.

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = <NBCS error codes>

Forcing out-of-character alignment: dalNbcsSswForceLkcOca

This function is used to force out-of-character alignment in the LVDS links.

Prototype
```
INT4 dalNbcsSswForceLkcOca(void* deviceInfo,
UINT1 port)
```

Inputs `deviceInfo` : device information handle
`port` : the port number ranges from 0 –31

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = <NBCS error codes>

Forcing out-of-frame alignment: dalNbcsSswForceLkcOfa

This function is used to force out-of-frame alignment in the LVDS links.

Prototype
```
INT4 dalNbcsSswForceLkcOfa(void* deviceInfo,
UINT1 port)
```

Inputs `deviceInfo` : device information handle
`port` : the port number ranges from 0-31

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = <NBCS error codes>

Enabling/Disabling the LVDS Link: dalNbcsSswCntlLkc

This function enables/disables the specified link.

Prototype
```
INT4 dalNbcsSswCntlLkc(void* deviceInfo, UINT1
dir, UINT1 port, UINT1 enable)
```

Inputs
`deviceInfo` : device information handle
`dir` : 0 = transmit 1 = receive
`port` : the port number ranges from 0-31
`enable` : 0 = disable, 1 = enable

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = <NBCS error codes>

Accessing Link Operation Mode: dalNbcsSswCntlLkcOpMode

This function allows the user to get or set the current operating mode of the specified link.

A link is by default in standby (low-power) mode. The user can reset the port (which will be in normal mode again after the reset) or put it in a standby (low power) mode. Resetting or putting the port in normal mode will bring the port out of standby mode.

Prototype
```
INT4 dalNbcsSswCntlLkcOpMode(void* deviceInfo,
UINT1 port, UINT1 mode)
```

Inputs
`deviceInfo` : device information handle
port : port number (from 0-31 for NSE-20G and 0-11 for NSE-8G)
mode : operating mode: 0 = standby, 1 = normal, 2 = reset

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Configuring LVDS link parameters: dalNbcsSswCfgLkc

This function allows user to configure the parameters for a specified link. Parameters are: J0 byte insertion, and path termination mode.

Prototype `INT4 dalNbcsSswCfgLkc(void* deviceInfo, UINT1 port, UINT1 accMode, sNBCS_CFG_LKC *pconfig)`

Inputs
`deviceInfo` : device information handle
`port` : link number ranges from 0-31
`mode` : 0 = get ,1 = set
`pconfig` : pointer to the configuration structure

Outputs `pconfig` : pointer to the configuration structure if accMode = 0

Returns
Success = `NBCS_SUCCESS`
Failure = <NBCS error codes>

Inserting Test Pattern in LVDS link: dalNbcsSswInsertLkcTp

This function enables/disables the insertion of test patterns into the LVDS links.

Prototype `INT4 dalNbcsSswInsertLkcTp(void* deviceInfo, UINT1 port, UINT2 tp, UINT1 enable)`

Inputs
`deviceInfo` : device information handle
`port` : port number ranges from 0-31.
`tp` : test pattern tp[0..9], a 10-bit number
`enable` : 0 = disable, 1 = enable

Outputs `None`

Returns
Success = `NBCS_SUCCESS`
Failure = <NBCS error codes>

In-band Link Controller

The in-band link controller is provided to facilitate inter-device communication. It is particularly useful to centralize control when the space switch is located in fabric cards and the time switches are located in multiple line cards.

Configuring the In-band Link Controller: dalNbcsSswCfgIlc

Set/Get ILC configuration parameters which include Rx FIFO timeout, and Rx FIFO interrupt threshold.

Prototype `INT4 dalNbcsSswCfgIlc(void* deviceInfo, UINT1 inport, UINT1 accMode, sNBCS_CFG_ILC *pconfig)`

Inputs
`deviceInfo` : device information handle

	`inport`	: port number (from 0-31 max)
	`accMode`	: access control: 0 = get, 1 = set
	`pconfig`	: (pointer to) configuration structure
Outputs	`pconfig`	: configuration structure when `accMode` is 0
Returns	Success = NBCS_SUCCESS Failure = <NBCS error codes>	

Enabling/Disabling Tx/Rx ILC: dalNbcsSswEnableIlc

When disabled, the Tx/Rx ILC will be in "bypass" mode. No messages will be written or inserted.

Prototype

```
INT4 nbcsIlcTxEnable(void* deviceInfo, UINT1
dir, UINT1 port, UINT1 enable)
```

Inputs	`deviceInfo`	: device information handle
	`dir`	: 0 = Tx, 1 = Rx
	`port`	: port number (from 0-31 max)
	`enable`	: enable flag: 0 = disable, 1 = enable
Outputs	None	
Returns	Success = NBCS_SUCCESS Failure = <NBCS error codes>	

Sending Messages in ILC: dalNbcsSswTxIlcMsg

This function is used to initiate the transmission of one or more in-band messages on one or more ports. There is no limitation on the number of messages to send in one request. A single call to this function can initiate transmission on multiple ports (the exact number is indicated by `numPorts`). `ptxBufDesc` points to an array of descriptors, one for each port on which messages are to be transmitted. This structure indicates the port number on which to transmit (`outport`), the size of this buffer (`bufSz`), and has a pointer to the buffer to be transmitted (`pbuf`). On return, the `bufSz` field contains the number of bytes transmitted on that port.

The length of each message is fixed at 32 bytes. The parameter `pyldSz` controls the number of user bytes that will be written in each message. The maximum payload size a message can carry is 32 bytes. If `pyldSz` is less than 32 bytes, the hardware will automatically pad the unfilled bytes in the message to 32. (Note that these remaining `(32 - pyldSz)` bytes are uninitialized.)

Prototype

```
INT4 dalNbcsSswTxIlcMsg(void* deviceInfo,
sNBCS_TXBUF_DESC_ILC* ptxBufDesc, UINT1 pyldSz,
UINT1 numPorts)
```

Inputs	`deviceInfo`	: device information handle
	`ptxBufDesc`	: (pointer to) buffer descriptor(s)

	`ptxBufDesc`	: (pointer to) buffer descriptor(s)
	`pyldSz`	: payload size (from 1 to 32 bytes)
	`numPorts`	: number of ports (from 1-32 max)
Outputs	`ptxBufDesc`	: buffer descriptor(s) that include the number of bytes sent for each port.
Returns	Success = NBCS_SUCCESS Failure = <NBCS error codes>	

Querying Free Space in ILC Tx FIFO: dalNbcsSswGetIlcTxFifoLvl

This function is to check the current capacity of the Tx FIFO. This allows the user to find out how many more messages can be written to FIFO for transmission.

Prototype `INT4 dalNbcsSswGetIlcTxFifoLvl(void* deviceInfo, UINT1 outport, UINT1* pnumMsg)`

Inputs

`deviceInfo`	: device information handle
`outport`	: port number (from 0-31 max)
`pnumMsg`	: (pointer to) free FIFO capacity

Outputs `pnumMsg` : free FIFO capacity

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Setting Tx Message Header: dalNbcsSswSetIlcTxHdr

Sets LINK, AUX and optionally the PAGE and/or USER bits in the transmit header for a given port. If the PAGE and/or USER bits have to be changed in a coordinated fashion across all ports, set the arguments `pageUpdate` and/or `userUpdate` to false and use `dalNbcsSswSetIlcTxHdrPage` and/or `dalNbcsSetIlcTxHdrUser` instead.

Prototype `INT4 dalNbcsSswSetIlcTxHdr(void* deviceInfo, UINT1 outport, sNBCS_HEADER_ILC *phead, UINT1 pageUpdate, UINT1 userUpdate)`

Inputs

`deviceInfo`	: device information handle
`outport`	: port number (from 0-31 max)
`phead`	: pointer to header structure
`pageUpdate`	: flag: 0 = don't include page bits, 1 = include
`userUpdate`	: flag: 0 = don't include user bits, 1 = include

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Setting PAGE bits in Tx Message Header: dalNbcsSswSetIlcTxHdrPage

Sets PAGE[1:0] bits in header for all links simultaneously. This is used to coordinate the changes across all links. Argument pPage is a pointer to a buffer containing the value of all page bits to be sent out. The buffer is expected to contain 32 bytes. Each byte contains the value (0-3) of the PAGE bits to be transmitted in the ILC header on the corresponding port. `c1fpSync` is a flag that indicates whether the bits should be updated immediately, or synchronized with the arrival of the next c1 frame pulse interrupt.

Prototype

```
INT4 dalNbcsSswSetIlcTxHdrPage(void*
deviceInfo, UINT1* pPage, UINT1 c1fpSync)
```

Inputs

`deviceInfo` : device information handle
`pPage` : (pointer to) PAGE buffer
`c1fpSync` : flag: indicates when update takes place. (0 = update page bits now, 1 = update page bits when next c1fp interrupt occurs.)

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Setting USER bits in Tx Message Header: dalNbcsSswSetIlcTxHdrUser

Sets USER[2:0] bits in header for all links simultaneously. This is used to coordinate the changes across all links. Argument puser is a pointer to a buffer containing the value of all page bits to be sent out. The buffer is expected to contain 32 bytes. Each byte contains the value (0-7) of the USER bits to be transmitted in the ILC header on the corresponding port. (I.e. the first byte in the buffer contains the value of the user bits for port 0, the second byte contains the value for port 1, etc.)

Prototype

```
INT4 dalNbcsSswSetIlcTxHdrUser(void*
deviceInfo, UINT1* puser)
```

Inputs

`deviceInfo` : device information handle
`puser` : (pointer to) USER buffer

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Getting Tx Message Header: dalNbcsSswGetIlcTxHdr

Retrieves all header bits to be transmitted for a given port.

Prototype `INT4 dalNbcsSswGetIlcTxHdr(void* deviceInfo, UINT1 outport, sNBCS_HEADER_ILC *phead)`

Inputs
- `deviceInfo` : device information handle
- `outport` : port number (from 0-31 max)
- `phead` : (pointer to) header structure

Outputs
- `phead` : header structure

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Getting Number of Messages in Rx FIFO: dalNbcsSswGetIlcRxNumMsg

Query the total number of messages currently stored in the Rx FIFO.

Prototype `INT4 dalNbcsSswGetIlcRxNumMsg(void* deviceInfo, UINT1 inport, UINT1 *pnumMsg)`

Inputs
- `deviceInfo` : device information handle
- `inport` : port number (from 0-31 max)
- `pnumMsg` : (pointer to) the buffer that holds the number of messages stored in FIFO

Outputs
- `pnumMsg` : the number of messages in the FIFO

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Getting Messages in Rx FIFO: dalNbcsSswGetIlcRxMsg

This function retrieves one or more ILC messages from the Rx FIFO of one or more ports (the exact number is indicated by `numPorts`). (A maximum of 8 messages per port can be retrieved each time this function is called.)

`prxBufDesc` points to an array of `numPorts` buffer descriptors, one for each port from which a message is to be retrieved. Each buffer descriptor indicates the port number from which to read (`inport`), the maximum number of messages to read (`numMsgs`), and has a pointer to `numMsgs` message descriptors (`pmsgDesc`). (If `numMsgs` is set to 0, this port will be ignored.)

Each message descriptor contains the location in which the message is to be stored (`pmsg`), and the status of the CRC for that message (`crc`) (returned by the driver).

This function will read up to `numMsgs` messages from each port for which a buffer descriptor exists. The number of messages actually received is returned to the user in the `numMsgs` field of the buffer descriptor. (Setting `numMsgs` to 8 will always read all available messages in the Rx FIFO.)

The parameter `pyldSz` controls the number of bytes to be read in one message. The maximum payload size in a message is 32 bytes. This function will only attempt to read the number of bytes specified in `pyldSz`. This gives the user the ability to avoid reading extra bytes in a message if the payload is known to be fewer than 32 bytes.

Prototype

```
INT4 dalNbcsSswGetIlcRxMsg(void* deviceInfo,
sNBCS_RXBUF_DESC_ILC* prxBufDesc, UINT1 pyldSz,
UINT1 numPorts)
```

Inputs

`deviceInfo` : device information handle
`prxBufDesc` : (pointer to) buffer descriptors (this must point to `numPorts` descriptors)
`pyldSz` : payload size (from 1 to 32 bytes)
`numPorts` : number of ports (from 1-32)

Outputs

`prxBufDesc` : buffer descriptor structures which include the received messages and their corresponding CRC status

Returns

Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Getting Rx Header Bytes: nbcsIlcGetRxHdr

Gets the header bytes received for a given port.

Prototype

```
INT4 dalNbcsGetIlcRxHdr(void* deviceInfo, UINT1
inport, sNBCS_HEADER_ILC *phead)
```

Inputs

`deviceInfo` : device information handle
`inport` : port number (from 0-31 max)
`phead` : (pointer to) header structure

Outputs

`phead` : header structure

Returns

Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Status and Counts

Reading the Device Counters: dalNbcsSswGetCounts

This function retrieves all the device counts. This routine should be called by the application code, in the context of a task. It is the user's responsibility to ensure that this function is called often enough to prevent the device counts from saturating or rolling over.

Prototype `INT4 dalNbcsSswGetCounts(void* deviceInfo, sNBCS_CNTR *pCntr)`

Inputs `deviceInfo` : device information handle
`pCntr` : allocated memory for counts

Outputs `pCntr` : current device counts

Returns Success = `NBCS_SUCCESS`
Failure = <NBCS error codes>

Getting the Current Status: dalNbcsSswGetStatus

This function retrieves a snapshot of the current status from the device registers. This involves retrieving current alarms, status, and clock activity. It is the user's responsibility to ensure the buffer indicated by `pStatus` is large enough to hold all the returned status of the members in the group.

Prototype `INT4 dalNbcsSswGetStatus(void* deviceInfo, sNBCS_STATUS *pStatus)`

Inputs `deviceInfo` : device information handle
`pStatus` : pointer to allocated memory

Outputs `pStatus` : current status

Returns Success = `NBCS_SUCCESS`
Failure = <NBCS error codes>

Interrupt Service Functions

Configuring ISR Processing: dalNbcsSswCfgISRMode

This function allows the user to configure how the interrupts are handled: either in polling (NBCS_POLL_MODE) or interrupt driven (NBCS_ISR_MODE) modes. If polling is selected, the user is responsible for calling periodically `dalNbcsSswPoll` to collect exception data from the device.

Prototype `INT4 dalNbcsSswCfgISRMode (void* deviceInfo, eNBCS_ISR_MODE mode)`

Inputs `deviceInfo` : device information handle
`mode` : mode of operation

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Getting the Interrupt Enable Mask: dalNbcsSswGetISRMask

Returns the contents of the interrupt mask from the space switch device.

Prototype `INT4 dalNbcsSswGetISRMask(void* deviceInfo, void *pmask)`

Inputs `deviceInfo` : device information handle
`pmask` : (pointer to) mask structure

Outputs `pmask` : updated mask structure

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Setting the Interrupt Enable Mask: dalNbcsSswSetISRMask

Sets the contents of the interrupt mask of the space switch device. A field set in the passed mask will set the corresponding device interrupt enable. For those zero values in the passed mask, the corresponding interrupt enables are left unaltered.

Prototype `INT4 dalNbcsSswSetISRMask(void* deviceInfo, void *pmask)`

Inputs `deviceInfo` : device information handle
`pmask` : (pointer to) mask structure

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Clearing the Interrupt Enable Mask: dalNbcsSswClearISRMask

Clears the content of the interrupt mask of the space switch device. A field set in the passed mask will clear the corresponding device interrupt enable. For those zero values in the passed mask, the corresponding interrupt enable are left untouched.

Prototype `INT4 dalNbcsSswClearISRMask(void* deviceInfo, void *pmask)`

Inputs `deviceInfo` : device information handle
`pmask` : (pointer to) mask structure

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Polling the Interrupt Status Registers: dalNbcsSswPoll

Commands the driver to poll the interrupt registers in the device. The call will fail unless the device was initialized (via `dalNbcsSswInit`) or configured (via `dalNbcsSswCfgISRMode`) into polling mode.

Prototype `INT4 dalNbcsSswPoll(void* deviceInfo)`

Inputs `deviceInfo` : device information handle

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Enabling/Disabling the C1 Frame Pulse Interrupt: dalNbcsSswEnaIsrC1fp

Enables or disables the C1 frame pulse interrupt in the space switch.

Prototype `INT4 dalNbcsSswEnaIsrC1fp(void* deviceInfo, UINT1 ena)`

Inputs `deviceInfo` : device information handle
`ena` : 0 = disable, 1 = enable

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Diagnostics

Testing Register Accesses: dalNbcsSswDiagTestReg

Verifies the hardware access to the device registers by writing and reading back values. The following types of register tests can be performed --- single value write/read and walking ones. In addition, each of these tests can be run on the full range of registers.

The write/read test writes the specified value to the specified register and verifies that the same value is read back. The walking ones test performs a series of writes to the specified register.

Prototype `INT4 dalNbcsSswDiagTestReg(void* deviceInfo, sNBCS_DIAG_TEST_REG *ptestReg)`

Inputs `deviceInfo` : device information handle
`ptestReg` : (pointer to) test structure

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Testing RAM Accesses: dalNbcsSswDiagTestRam

Verifies the hardware access to the device internal RAM by writing and reading back values. The following types of RAM tests can be performed: single write/read, walking ones, migrating ones, and aliasing. Note that both connection maps are tested for all types of tests listed above. The first three types can be performed on either a user-specified range or the entire RAM. Aliasing is always performed on the entire RAM.

Prototype `INT4 dalNbcsSswDiagTestRam(void* deviceInfo, sNBCS_DIAG_TEST_RAM *ptestRam)`

Inputs `deviceInfo` : device information handle
`ptestRam` : (pointer to) test structure

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

TIME SWITCH DEVICE DRIVER INTERFACE

This section describes the DAL interface for a generic time switch device driver such as a SBS or a SBSLITE device. The module and device management block, Interface/Clock, Status/Counts and Diagnostics blocks are standard blocks that encapsulate that of a typical PMC device driver. The LVDS controller, In-band link Controller, and the Switch configuration blocks are logical blocks that are specific to a typical time switch device.

Module and Device Management

The module and device management block connects with that of the underlying driver.

Opening the Space Switch Driver Module: dalNbcsTswModuleOpen

Performs module level initialization of the time switch device driver by calling the underlying module open function provided by the time switch driver. This usually involves allocating all of the memory required by the driver and initializing the internal structures.

Prototype `INT4 dalNbcsTswModuleOpen(sNBCS_MIV_DAL *pmiv)`

Inputs `pmiv` : (pointer to) Module Initialization Vector

Outputs Places the address of `errModule` into the MIV passed by the Application.

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Closing the Space Switch Driver Module: dalNbcsTswModuleClose

Performs module level shutdown of the time switch driver. This involves deleting all devices being controlled by the driver and freeing all the memory allocated by the driver.

Prototype `INT4 dalNbcsTswModuleClose(void)`

Inputs None

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Starting the Space Switch Driver Module: dalNbcsTswModuleStart

Starts the module in the underlying time switch driver. Upon successful return from this function, the driver is ready to add devices.

Prototype `INT4 dalNbcsTswModuleStart(void)`

Inputs None

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Stopping the Space Switch Driver Module: dalNbcsTswModuleStop

Stops the module in the underlying time switch driver.

Prototype `INT4 dalNbcsTswModuleStop(void)`

Inputs None

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Adding a Device: dalNbcsTswAdd

Invokes the native device add function supplied by the time switch driver. The error device pointer is returned along with the handle returned by the time switch driver.

Prototype

```
INT4 dalNbcsTswAdd(void* usrCtxt, void
baseAddr, void** pHndl, INT4 **pperrDevice)
```

Inputs

`usrCtxt`	: user context for this device
`baseAddr`	: base address of the device
`pHndl`	: (pointer to) device handle
`pperrDevice`	: (pointer to) an area of memory

Outputs

`pperrDevice`	: (pointer to) errDevice (inside the DDB of the time switch driver)
`pHndl`	: (pointer to) device handle

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Deleting a Device: dalNbcsTswDelete

This function is used to remove the specified device from the list of devices being controlled by the time switch driver. Deleting a device involves clearing the DDB for that device and releasing its associated device handle.

Prototype `INT4 dalNbcsTswDelete(void* deviceInfo)`

Inputs `deviceInfo` : device information handle

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Initializing a Device: dalNbcsTswInit

Invokes the device initialization function provided by the time switch driver using the DIV or the profile number. If the DIV is passed as a NULL the profile number is used. A profile number of zero indicates that all the register bits are to be left in their default state. Note that the profile number is ignored UNLESS the passed DIV is NULL.

Prototype `INT4 dalNbcsTswInit(void* deviceInfo, sNBCS_DIV_TSW_DAL *pdiv, UINT2 profileNum)`

Inputs
`deviceInfo` : device information handle
`pdiv` : (pointer to) Device Initialization Vector
`profileNum` : profile number (only used if `pdiv` is NULL)

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Updating the Configuration of a Device: dalNbcsTswUpdate

Updates the configuration of the device according to the DIV passed by the Application. The only difference between `dalNbcsTswUpdate` and `dalNbcsTswInit` is that no soft reset will be applied to the device. In addition, a profile number of zero is not allowed.

Prototype `INT4 dalNbcsTswUpdate(void* deviceInfo, sNBCS_DIV_TSW_DAL *pdiv, UINT2 profileNum)`

Inputs
`deviceInfo` : device information handle
`pdiv` : (pointer to) Device Initialization Vector
`profileNum` : profile number (only used if `pdiv` is

NULL)

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Resetting a Device: dalNbcsTswReset

Applies a software reset to the time switch device. This function is typically called before re-initializing the device (via `dalNbcsTswInit`).

Prototype `INT4 dalNbcsTswReset(void* deviceInfo)`

Inputs `deviceInfo` : device information handle

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Activating a Device: dalNbcsTswActivate

Restores the state of a device after a de-activate.

Prototype `INT4 dalNbcsTswActivate(void* deviceInfo)`

Inputs `deviceInfo` : device information handle

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

De-Activating a Device: dalNbcsTswDeActivate

De-activates the device from operation.

Prototype `INT4 dalNbcsTswDeActivate(void* deviceInfo)`

Inputs `deviceInfo` : device information handle

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Reading from Device Registers: dalNbcsTswRead

This function can be used to read a register of a specific time switch device by providing the register number. This function derives the actual address location based on the device handle and register number inputs. It then reads the contents of this address location

Prototype `INT4 dalNbcsTswRead(void* deviceInfo, UINT2 regNum, UINT4* pval)`

Inputs
`deviceInfo` : device information handle
`regNum` : register number
`pval` : pointer to the value read

Outputs `pval` : pointer to the value read

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Writing to Device Registers: dalNbcsTswWrite

This function can be used to write to a register of a specific time switch device by providing the register number. This function derives the actual address location based on the device handle and register number inputs. It then writes the data to the specified address location.

Prototype `INT4 dalNbcsTswWrite(void* deviceInfo, UINT2 regNum, UINT4 value)`

Inputs
`deviceInfo` : device information handle
`regNum` : register number
`value` : value to be written

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Reading from a block of Device Registers: dalNbcsTswReadBlock

This function can be used to read a register block of a specific time switch device by providing the starting register number and the size to read. This function derives the actual start address location based on the device handle and starting register number inputs. It then reads the contents of this data block.

Prototype `INT4 dalNbcsTswReadBlock(void* deviceInfo, UINT2 startRegNum, UINT2 size, UINT4 *pblock)`

Inputs
`deviceInfo` : device information handle
`startRegNum` : starting register number
`size` : size of the block to read

	`pblock`	: (pointer to) the block to read
Outputs	`pblock`	: (pointer to) the block read
Returns	Success = NBCS_SUCCESS Failure = <NBCS error codes>	

Writing to a Block of Device Registers: dalNbcsTswWriteBlock

This function can be used to write to a register block of a specific time switch device by providing the starting register number and the block size. This function derives the actual starting address location based on the device handle and starting register number inputs. It then writes the contents of this data block to the starting address location.

Prototype
```
INT4 dalNbcsTswWriteBlock(void* deviceInfo,
UINT2 startRegNum, UINT2 size, UINT4 *pblock,
UINT4 *pmask)
```

Inputs	`deviceInfo`	: device information handle
	`startRegNum`	: starting register number
	`size`	: size of block to read
	`pblock`	: (pointer to) block to write
	`pmask`	: (pointer to) mask
Outputs	None	
Returns	Success = NBCS_SUCCESS Failure = <NBCS error codes>	

Adding an Initialization Profile: dalNbcsTswAddInitProfile

Creates an initialization profile that is stored by the driver. A device can be initialized by passing the initialization profile number to `dalNbcsTswInit`.

Prototype
```
INT4 dalNbcsTswAddInitProfile(sNBCS_DIV_TSW_DAL
*pProfile, UINT2 *pProfileNum)
```

Inputs	`pProfile`	: (pointer to) initialization profile being added
	`pProfileNum`	: (pointer to) profile number to be assigned by the driver
Outputs	`pProfileNum`	: profile number assigned by the driver
Returns	Success = NBCS_SUCCESS Failure = <NBCS error codes>	

Getting an Initialization Profile: dalNbcsTswGetInitProfile

Gets the content of an initialization profile given its profile number.

Prototype
```
INT4 dalNbcsTswGetInitProfile(UINT2 profileNum,
sNBCS_DIV_TSW_DAL *pProfile)
```

Inputs
`profileNum` : initialization profile number
`pProfile` : (pointer to) initialization profile

Outputs
`pProfile` : contents of the corresponding profile

Returns
Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Deleting an Initialization Profile: dalNbcsTswDeleteInitProfile

Deletes an initialization profile given its profile number.

Prototype
```
INT4 dalNbcsTswDeleteInitProfile(UINT2
profileNum)
```

Inputs
`profileNum` : initialization profile number

Outputs
None

Returns
Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Connection Switch Configuration

Configuring the Time Switch: dalNbcsTswCfgSwhParm

Get/Set configuration of the time switch. Parameters to configure include C1 delay, switching mode, swap mode, and page copy auto update.

Prototype
```
INT4 dalNbcsTswCfgSwhParm(void* deviceInfo,
UINT1 accMode, sNBCS_CFG_SWH_DAL *pconfig)
```

Inputs
`deviceInfo` : device information handle
`accMode` : access control: 0 = get, 1 = set
`pconfig` : (pointer to) configuration structure

Outputs
`pconfig` : configuration structure when `accMode` is 0

Returns
Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Failure = <NBCS error codes>

Setting Up Connections: dalNbcsTswMapSlot

Establish connections in the time switch. This mapping function can operate in three different modes, namely unicast, map, and straight through.

In unicast mode, the user supplies an array of incoming bytes/columns and an array of the corresponding outgoing bytes/columns. One to one mapping is assumed for the `pinSlot` array and the `poutSlot` array i.e. `pinSlot[0]` mapped to `poutSlot[0]`, `pinSlot[1]` mapped to `poutSlot[1]` and so on. `numSlot` specifies the size of the `poutSlot` array.

In map mode, the user provides an array of incoming bytes/columns, `pinSlot`. The size of the array is specified by `numSlot` value. `pOutSlot` points to a single value, which is taken to be the first outgoing byte/column. The software assumes the remaining outgoing bytes/columns to be sequential increments from the first outgoing byte/column up to `numOutSlot` bytes/columns. e.g. if `poutSlot` specifies the value `x`, then the outgoing bytes/columns mapped are `x` to `x+numOutSlot`.

Straight through mode provides a one-to-one direct mapping from input to output ports for each timeslot. In this mode, input port n is mapped to output port n for every port and timeslot. `pinslot`, `poutslot`, and `numSlot` are all ignored in this mode.

Whenever applicable, the range of timeslots is expected to be from 0-1079 and 0-9719 in the case of TeleCombus/SBI column mode and SBI DS0/CAS mode respectively.

Prototype

```
INT4 dalNbcsTswMapSlot(void* deviceInfo, UINT1
dir, eNBCS_SWH_ACCESSMODE_DAL mode, UINT2
*pinslot, UINT2 *poutslot, UINT4 numSlot)
```

Inputs

`deviceInfo`	: device information handle
`dir`	: 0 = ingress, 1 = egress switch
`mode`	: access mode
`pinslot`	: pointer to (array of) in timeslot(s)
`poutslot`	: pointer to (array of) out timeslot(s)
`numSlot`	: number of elements

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Getting Source Connections: dalNbcsTswGetSrcSlot

This function will return the source bytes/columns for the specified outgoing bytes/columns in the incoming or outgoing time switch module.

Prototype

```
INT4 dalNbcsTswGetSrcSlot(void* deviceInfo,
UINT1 dir, UINT2* poutslot, UINT2* pinslot,
```

```
UINT1 dir, UINT2* poutslot, UINT2* pinslot,
UINT2 numSlot)
```

Inputs
- `deviceInfo` : device information handle
- `dir` : 0 = ingress, 1 = egress switch
- `poutslot` : (pointer to) output timeslot
- `pinslot` : (pointer to) input timeslot
- `numSlot` : number of timeslots

Outputs
- `pinslot` : (pointer to) input timeslot(s)

Returns
Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Getting Active Page: dalNbcsTswGetActivePage

Get the active page in the time switch.

Prototype

```
INT4 dalNbcsTswGetActivePage(void* deviceInfo,
UINT1 dir, UINT1* pPage)
```

Inputs
- `deviceInfo` : device information handle
- `dir` : 0 = ingress, 1 = egress switch
- `pPage` : (pointer to) the active page number

Outputs
- `pPage` : the active page number

Returns
Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Setting Active Page: dalNbcsTswSetActivePage

Set the active page in the time switch.

Prototype

```
INT4 dalNbcsTswSetActivePage(void* deviceInfo,
UINT1 dir, UINT1 pageNum)
```

Inputs
- `deviceInfo` : device information handle
- `dir` : 0 = ingress, 1 = egress switch
- `pageNum` : the active page number

Outputs
None

Returns
Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Updating Inactive Page: dalNbcsTswUpdateInactivePage

Copy the connection settings from active to inactive page. This function is designed for manual copy operation when automatic page copy is not activated.

Prototype `INT4 dalNbcsTswUpdateInactivePage(void* deviceInfo, UINT1 dir)`

Inputs `deviceInfo` : device information handle
`dir` : 0 = ingress, 1 = egress switch

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

LVDS Link Controller

Inserting line code violation: dalNbcsTswInsertLkcLcv

This function enables or disables the insertion of line code violations in the LVDS links

Prototype `INT4 dalNbcsTswInsertLkcLcv(void* deviceInfo, UINT1 link, UINT1 enable)`

Inputs `deviceInfo` : device information handle
`link` :0 = working, 1 = protect link
`enable` : 0 = disable, 1 = enable

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = <NBCS error codes>

Centering transmit FIFO: dalNbcsTswCenterLkcFifo

This function is used to center the transmit FIFO in the LVDS links.

Prototype `INT4 dalNbcsTswCenterLkcFifo(void* deviceInfo, UINT1 link)`

Inputs `deviceInfo` : device information handle
`link` :0 = working, 1 = protect link

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = <NBCS error codes>

Failure = <NBCS error codes>

Forcing out-of-character alignment: dalNbcsTswForceLkcOca

This function is used to force out-of-character alignment in the LVDS links.

Prototype
```
INT4 dalNbcsTswForceLkcOca(void* deviceInfo,
UINT1 link)
```

Inputs
`deviceInfo` : device information handle
`link` :0 = working, 1 = protect link

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = <NBCS error codes>

Forcing out-of-frame alignment: dalNbcsTswForceLkcOfa

This function is used to force out-of-frame alignment in the LVDS links.

Prototype
```
INT4 dalNbcsTswForceLkcOfa(void* deviceInfo,
UINT1 link)
```

Inputs
`deviceInfo` : device information handle
`link` :0 = working, 1 = protect link

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = <NBCS error codes>

Enabling/Disabling the LVDS Link: dalNbcsTswCntlLkc

This function enables/disables the specified link.

Prototype
```
INT4 dalNbcsTswCntlLkc(void* deviceInfo, UINT1
dir, UINT1 link, UINT1 enable)
```

Inputs
`deviceInfo` : device information handle
`dir` : 0 = transmit 1 = receive
`link` :0 = working, 1 = protect link
`enable` : 0 = disable, 1 = enable

Outputs None

Returns Success = `NBCS_SUCCESS`
Failure = <NBCS error codes>

Configuring LVDS link parameters: dalNbcsTswCfgLkc

This function allows user to configure the parameters for a specified link. Parameters are: J0 byte insertion, and path termination mode.

Prototype `INT4 dalNbcsTswCfgLkc(void* deviceInfo, UINT1 link, sNBCS_CFG_LKC *pconfig)`

Inputs

`deviceInfo` : device information handle
`link` :0 = working, 1 = protect link
`pconfig` : pointer to the configuration structure

Outputs `None`

Returns Success = `NBCS_SUCCESS`
Failure = <NBCS error codes>

Inserting Test Pattern in LVDS link: dalNbcsTswInsertLkcTp

This function enables/disables the insertion of test patterns into the LVDS links.

Prototype `INT4 dalNbcsTswInsertLkcTp(void* deviceInfo, UINT1 link, UINT2 tp, UINT1 enable)`

Inputs

`deviceInfo` : device information handle
`link` :0 = working, 1 = protect link
`tp` : test pattern tp[0..9], a 10-bit number
`enable` : 0 = disable, 1 = enable

Outputs `None`

Returns Success = `NBCS_SUCCESS`
Failure = <NBCS error codes>

Selecting Active LVDS link: dalNbcsTswSelectLkc

This function selects either the working or protect link on the receive side of a time switch to be active

Prototype `INT4 dalNbcsTswSelectLkc(void* deviceInfo, UINT1 link)`

Inputs

`deviceInfo` : device information handle
`link` :0 = working, 1 = protect link

Outputs `None`

Returns Success = `NBCS_SUCCESS`
Failure = <NBCS error codes>

Failure = <NBCS error codes>

In-band Link Controller

The in-band link controller is provided to facilitate inter-device communication. It is particularly useful to centralize control when the time switch is located in fabric cards and the time switches are located in multiple line cards.

Configuring the In-band Link Controller: dalNbcsTswCfgIlc

Set/Get ILC configuration parameters which include Rx FIFO timeout, and Rx FIFO interrupt threshold.

Prototype

```
INT4 dalNbcsTswCfgIlc(void* deviceInfo, UINT1
link, UINT1 accMode, sNBCS_CFG_ILC *pconfig)
```

Inputs

`deviceInfo` : device information handle
`link` :0 = working, 1 = protect link
`accMode` : access control: 0 = get, 1 = set
`pconfig` : (pointer to) configuration structure

Outputs

`pconfig` : configuration structure when `accMode` is 0

Returns

Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Enabling/Disabling Tx/Rx ILC: dalNbcsTswEnableIlc

When disabled, the Tx/Rx ILC will be in “bypass” mode. No messages will be written or inserted.

Prototype

```
INT4 nbcsIlcTxEnable(void* deviceInfo, UINT1
dir, UINT1 link, UINT1 enable)
```

Inputs

`deviceInfo` : device information handle
`dir` : 0 = Tx, 1 = Rx
`link` :0 = working, 1 = protect link
`enable` : enable flag: 0 = disable, 1 = enable

Outputs

None

Returns

Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Sending Messages in ILC: dalNbcsTswSetIlcTxMsg

This function is used by the application to transmit messages on the in-band link over the working and protect serial links. There is no limitation on the number of messages (each message has a maximum of 32 bytes corresponding to size of data in each transmit FIFO) that can be sent. The application specifies the transmit data buffer(`pBuf` inside the `ptxBufDesc` structure) and the size of the transmit data buffer(`pBufSz` inside the `ptxBufDesc` structure). In addition the application can also specify the number of bytes of data to be sent in each message(`pyldSz`, which ranges from 1 to 32 bytes). If the application wants to use all the available message size, it will specify the pyldSz to be 32. In the event that the application has fixed size messages less than 32, say n (0 < n < 32) bytes, then pyldSz will be specified as n. In this case the function will put n bytes of transmit data in each transmit FIFO. Note that the remaining unused bytes in each transmit FIFO will be uninitialized.

The function returns the number of bytes that have been placed in the transmit FIFO's in pBufSz.

Prototype

```
INT4 dalNbcsTswSetIlcTxMsg(void* deviceInfo,
sNBCS_TXBUF_DESC_ILC* ptxBufDesc, UINT1 pyldSz,
UINT1 numPorts)
```

Inputs

`deviceInfo`	: device information handle
`ptxBufDesc`	: (pointer to) buffer descriptor(s)
`pyldSz`	: payload size (from 1 to 32 bytes)
`numPorts`	: number of links (from 1-32 max)

Outputs

`ptxBufDesc`	: buffer descriptor(s) that include the number of bytes sent for each link.

Returns

Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Querying Free Space in ILC Tx FIFO: dalNbcsTswGetIlcTxFifoLvl

This function is to check the current capacity of the Tx FIFO. This allows the user to find out how many more messages can be written to FIFO for transmission.

Prototype

```
INT4 dalNbcsTswGetIlcTxFifoLvl(void*
deviceInfo, UINT1 link, UINT1* pnumMsg)
```

Inputs

`deviceInfo`	: device information handle
`link`	:0 = working, 1 = protect link
`pnumMsg`	: (pointer to) free FIFO capacity

Outputs

`pnumMsg`	: free FIFO capacity

Returns

Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Setting Tx Message Header: dalNbcsTswSetIlcTxHdr

This function sets the values of the ILC transmit header bytes.

Prototype

```
INT4 dalNbcsTswSetIlcTxHdr(void* deviceInfo,
UINT1 link, sNBCS_HEADER_ILC *phead)
```

Inputs

`deviceInfo` : device information handle
`link` :0 = working, 1 = protect link
`phead` : pointer to header structure

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Getting Number of Messages in Rx FIFO: dalNbcsTswGetIlcRxNumMsg

Query the total number of messages currently stored in the Rx FIFO.

Prototype

```
INT4 dalNbcsTswGetIlcRxNumMsg(void* deviceInfo,
UINT1 link, UINT1 *pnumMsg)
```

Inputs

`deviceInfo` : device information handle
`link` :0 = working, 1 = protect link
`pnumMsg` : (pointer to) the buffer that holds the number of messages stored in FIFO

Outputs

`pnumMsg` : the number of messages in the FIFO

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Getting Messages in Rx FIFO: dalNbcsTswGetIlcRxMsg

This function retrieves data from the received messages on the in-band link over the working or protect link. The application can specify the number of messages to be read by `numMsg` variable (inside prxBufDesc). This function reads the number of messages currently available up to a maximum of `numMsg` messages. For example, if `numMsg` is specified as 8 and only 6 messages are currently available, this function reads the 6 messages and returns. The function updates the `numMsg` variable to indicate the actual number of messages read. The data and the CRC status from each message, are returned in `prxBufDesc`. The number of bytes of data read from each message is specified by `pyldSz`.

Prototype

```
INT4 dalNbcsTswGetIlcRxMsg(void* deviceInfo,
sNBCS_RXBUF_DESC_ILC* prxBufDesc, UINT1 pyldSz)
```

Inputs

`deviceInfo` : device information handle
`prxBufDesc` : (pointer to) buffer descriptors (this

must point to `numPorts` descriptors)

	`pnumDesc`	: (pointer to) number of message descriptors
	`pyldSz`	: payload size (from 1 to 32 bytes)
Outputs	`prxBufDesc`	: buffer descriptor structures which include the received messages and their corresponding CRC status
Returns	Success = NBCS_SUCCESS Failure = <NBCS error codes>	

Getting Rx Header Bytes: dalNbcsGetIlcRxHdr

Gets the header bytes received for a given link.

Prototype `INT4 dalNbcsGetIlcRxHdr(void* deviceInfo, UINT1 link, sNBCS_HEADER_ILC *phead)`

Inputs	`deviceInfo`	: device information handle
	`link`	:0 = working, 1 = protect link
	`phead`	: (pointer to) header structure
Outputs	`phead`	: header structure
Returns	Success = NBCS_SUCCESS Failure = <NBCS error codes>	

Status and Counts

Reading the Device Counters: dalNbcsTswGetCounts

This function retrieves all the device counts. It is the user's responsibility to ensure that this function is called often enough to prevent the device counts from saturating or rolling over.

Prototype `INT4 dalNbcsTswGetCounts(void* deviceInfo, sNBCS_CNTR *pCntr)`

Inputs	`deviceInfo`	: device information handle
	`pCntr`	: allocated memory for counts
Outputs	`pCntr`	: current device counts
Returns	Success = `NBCS_SUCCESS` Failure = <NBCS error codes>	

Getting the Current Status: dalNbcsTswGetStatus

This function retrieves a snapshot of the current status from the device registers. It is the user's responsibility to ensure the buffer indicated by `pStatus` is large enough to hold all the returned status of the members in the group.

Prototype
```
INT4 dalNbcsTswGetStatus(void* deviceInfo,
sNBCS_STATUS *pStatus)
```

Inputs `deviceInfo` : device information handle
`pStatus` : pointer to allocated memory

Outputs `pStatus` : current status

Returns Success = `NBCS_SUCCESS`
Failure = <NBCS error codes>

Interrupt Service Functions

Configuring ISR Processing: dalNbcsTswCfgISRMode

This function allows the user to configure how the interrupts are handled: either in polling (NBCS_POLL_MODE) or interrupt driven (NBCS_ISR_MODE) modes. If polling is selected, the user is responsible for calling periodically `dalNbcsTswPoll` to collect exception data from the device.

Prototype
```
INT4 dalNbcsTswCfgISRMode (void* deviceInfo,
eNBCS_ISR_MODE mode)
```

Inputs `deviceInfo` : device information handle
`mode` : mode of operation

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Getting the Interrupt Enable Mask: dalNbcsTswGetISRMask

Returns the contents of the interrupt mask from the time switch device.

Prototype
```
INT4 dalNbcsTswGetISRMask(void* deviceInfo,
void *pmask)
```

Inputs `deviceInfo` : device information handle
`pmask` : (pointer to) mask structure

Outputs `pmask` : updated mask structure

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Setting the Interrupt Enable Mask: dalNbcsTswSetISRMask

Sets the contents of the interrupt mask of the time switch device. A field set in the passed mask will set the corresponding device interrupt enable. For those zero values in the passed mask, the corresponding interrupt enables are left unaltered.

Prototype
```
INT4 dalNbcsTswSetISRMask(void* deviceInfo,
void *pmask)
```

Inputs `deviceInfo` : device information handle
`pmask` : (pointer to) mask structure

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Clearing the Interrupt Enable Mask: dalNbcsTswClearISRMask

Clears the content of the interrupt mask of the time switch device. A field set in the passed mask will clear the corresponding device interrupt enable. For those zero values in the passed mask, the corresponding interrupt enable are left unaltered.

Prototype
```
INT4 dalNbcsTswClearISRMask(void* deviceInfo,
void *pmask)
```

Inputs `deviceInfo` : device information handle
`pmask` : (pointer to) mask structure

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Polling the Interrupt Status Registers: dalNbcsTswPoll

Commands the driver to poll the interrupt registers in the device. The call will fail unless the device was initialized (via `dalNbcsTswInit`) or configured (via `dalNbcsTswCfgISRMode`) into polling mode.

Prototype
```
INT4 dalNbcsTswPoll(void* deviceInfo)
```

Inputs `deviceInfo` : device information handle

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Enabling/Disabling the C1 Frame Pulse Interrupt: dalNbcsTswEnaIsrC1fp

Enables or disables the C1 frame pulse interrupt in the time switch.

Prototype
```
INT4 dalNbcsTswEnaIsrC1fp(void* deviceInfo,
UINT1 dir)
```

Inputs `deviceInfo` : device information handle
`dir` : 0 = ingress, 1 = egress

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Diagnostics

Testing Register Accesses: dalNbcsTswDiagTestReg

Verifies the hardware access to the device registers by writing and reading back values. The following types of register tests can be performed --- single value write/read and walking ones. In addition, each of these tests can be run on the full range of registers.

Prototype
```
INT4 dalNbcsTswDiagTestReg(void* deviceInfo,
sNBCS_DIAG_TEST_REG *ptestReg)
```

Inputs `deviceInfo` : device information handle
`ptestReg` : (pointer to) test structure

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Testing RAM Accesses: dalNbcsTswDiagTestRam

Verifies the hardware access to the device internal RAM by writing and reading back values. The following types of RAM tests can be performed: single write/read, walking ones, migrating ones, and aliasing. Note that both connection maps are tested for all types of tests listed above. The first three types can be performed on either a user-specified range or the entire RAM. Aliasing is always performed on the entire RAM.

Prototype `INT4 dalNbcsTswDiagTestRam(void* deviceInfo, sNBCS_DIAG_TEST_RAM *ptestRam)`

Inputs
`deviceInfo` : device information handle
`ptestRam` : (pointer to) test structure

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Controlling diagnostic loopbacks: dalNbcsTswDiagLpbk

This function is used to control the diagnostic loopbacks available on a time switch device

Prototype `INT4 dalNbcsTswDiagLpbk(void* deviceInfo, eNBCS_LPBK lpbk, UINT1 enable)`

Inputs
`deviceInfo` : device information handle
`lpbk` : specifies the loopback options
`enable` : 0 – disable, 1 – enable

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

PRBS Generation/Monitoring Control

This section describes the functions that are used to configure the pseudo random pattern generation and monitoring on the working and protect links. The granularity is assumed to be STS-1.

Configuring payload for the PRGM: dalNbcsTswCfgPrgmPyld

This function configures the payload for the PRBS generator or monitor on the working and protect links. The payload can be configured as 12 STS-1's, 4 STS-3c's , a combination of STS-1's and STS-3c's or a single STS-12c.

Prototype

```
INT4 dalNbcsTswCfgPrgmPyld(void* deviceInfo,
UINT1 genMon, UINT1 link, sNBCS_CFG_PRGM_PYLD
*pPyldCfg, UINT1 accMode)
```

Inputs

- `deviceInfo` : device information handle (from `nbcsAdd`)
- `genMon` : 0 – PRBS generator
  1 – PRBS monitor
- `link` : 0 = working, 1 = protect link
- `pPyldCfg` : structure containing the payload configuration
- `accMode` : access control: 0 – get, 1 – set

Outputs

- `pPyldCfg` : returns payload configuration parameters when accMode = 0

Returns

Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Configuring the PRGM: dalNbcsTswCfgPrgm

This function is used to enable/disable the generation and monitoring of the PRBS sequence on each STS-1 on the working and protect links. Besides this the function is used to configure the linear feedback shift register(LFSR), which is used to generate the PRBS sequence. In addition it can also configure the generator and monitor in the invert PRBS sequence mode or sequential mode.

Prototype

```
INT4 dalNbcsTswCfgPrgm(void* deviceInfo, UINT1
genMon, UINT1 link, UINT1 sts1Path,
sNBCS_CFG_PRGM *pCfg, UINT1 accMode)
```

Inputs

- `deviceInfo` : device information handle
- `genMon` : 0 – PRBS generator, 1 –monitor
- `link` : 0 = working, 1 = protect link
- `sts1Path` : the STS-1 path. Valid range is 0-11
- `pCfg` : structure containing the bits to be programmed in LFSR. Also has flags for invert PRBS mode, sequential mode, and autonomous mode
- `accMode` : 0 = disable the generation or monitoring function, 1= enable the generation or monitoring function without setting any configurations; 2 = set up the configuration before enabling the generation or monitoring function; 3 = retrieve the configuration parameters

Outputs

- `pCfg` : returns pattern configuration parameters when accMode = 3 parameters when `accMode` = 3

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Forcing a bit error in the PRBS sequence: dalNbcsTswForcePrgmErr

This function is used to force a bit error in the PRBS sequence on the specified STS-1 data stream on the working or protect link. One bit error is inserted each time the function is invoked.

Prototype
```
INT4 dalNbcsTswForcePrgmErr(void* deviceInfo,
UINT1 link, UINT1 sts1Path)
```

Inputs
`deviceInfo` : device information handle
`link` : 0 = working, 1 = protect link
`sts1Path` : STS-1 number. Valid range is 0-11

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Forcing Resynchronization in incoming PRBS data stream: dalNbcsTswForcePrgmResync

This function is used to force the PRBS monitor on a specified STS-1 on the working or protect link to resynchronize to the incoming data stream.

Prototype
```
INT4 dalNbcsTswForcePrgmResync(void*
deviceInfo, UINT1 link, UINT1 sts1Path)
```

Inputs
`deviceInfo` : device information handle
`link` : 0 = working,1 = protect link
`sts1Path` : STS-1 number. Valid range is 0-11

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Interface/Clock Configuration

This section describes functions that configure the external interfaces and the clocks of the device.

Configuring the TeleCombus/SBI Bus Mode: dalNbcsTswCfgIntfBusMode

This function configures the mode for the incoming, outgoing, transmit and receive buses.

Prototype

```
INT4 dalNbcsTswCfgIntfBusMode(void* deviceInfo,
sNBCS_CFG_BUSMODE_DAL *pIntfCfg, UINT1 accMode)
```

Inputs

`deviceInfo` : device information handle

`pIntfCfg` : structure containing configuration parameters for the incoming, outgoing, transmit and receive buses

`accMode` : access control: 0 – get, 1 – set

Outputs

`pIntfCfg` : returns the configuration parameters when the accMode = 0

Returns

Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Configuring the bus parameters: dalNbcsTswCfgIntfBusParms

This function configures the bus parity for the incoming, outgoing, transmit and receive buses. It also configures the offset of the J1 byte for the telecom bus mode.

Prototype

```
INT4 dalNbcsTswCfgIntfBusParms(void*
deviceInfo, UINT1 busType,
sNBCS_CFG_INTF_BUSPARM_DAL *pBusParm, UINT1
accMode)
```

Inputs

`deviceInfo` : device information handle

`busType` : 0 = incoming parallel, 1 = outgoing parallel, 2 = serial transmit, 3 = serial receive bus

`pBusParm` : parameters to configure the selected bus

`accMode` : access control: 0 – get, 1 – set

Outputs

`pBusParm` : returns the bus parity parameters when the accMode = 0

Returns

Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Configuring the TeleCombus Parameters: dalNbcsTswCfgTelecomParms

This function configures the telecom bus specific parameters for the outgoing and the transmit buses. It configures the behavior of the C1FP signal on occurrence of J1 or V1 byte. The function also allows the user to specify the values of the H1 and H2 pointers.

Prototype `INT4 dalNbcsTswCfgTelecomParms(void* deviceInfo, UINT1 busType, sNBCS_CFG_BUSPARAM *pBusParm, UINT1 accMode)`

Inputs

`deviceInfo`	: device information handle
`busType`	: 1 = outgoing parallel, 2 = serial transmit bus
`pBusParm`	: parameters specifying the behavior of the C1FP signal on occurrence of J1 or V1 byte and also specifying the value of the H1 and H2 pointers
`accMode`	: access control: 0 – get, 1 – set

Outputs `pBusParms` : returns the egress telecom bus parameters when the accMode = 0

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Configuring the TeleCombus Payload: dalNbcsTswCfgTelecomPyld

This function configures the telecom bus specific parameters for the outgoing and the transmit buses. It configures the behavior of the C1FP signal on occurrence of J1 or V1 byte. The function also allows the user to specify the values of the H1 and H2 pointers.

Prototype `INT4 dalNbcsTswCfgTelecomPyld(void* deviceInfo, UINT1 busType, sNBCS_CFG_PYLD_TCB *pTelecomPyld)`

Inputs

`deviceInfo`	: device information handle
`busType`	: 0 = incoming parallel, 1 = outgoing parallel, 2 = serial transmit, 3 = serial receive bus
`pTelecomPyld`	: TeleCombus payload configuration

Outputs `None`

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Configuring the SBI Bus Payload: dalNbcsTswCfgSbiPyld

This function configures the SBI bus payload type. It assumes symmetrical payload on ingress and egress direction.

Prototype `INT4 dalNbcsTswCfgSbiPyld(void* deviceInfo, UINT1 busType, sNBCS_CFG_PYLD_SBI *pSbiPyld, UINT1 accMode)`

Inputs `deviceInfo` : device information handle
`busType` : 0 = incoming parallel, 1 = outgoing parallel, 2 = serial transmit, 3 = serial receive bus
`pSbiPyld` : SBI bus payload configuration
`accMode` : 0 = get,1 = set

Outputs `pSbiPyld` : SBI bus payload configuration when accMode = 0

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Enabling/Disabling CAS in a SBI Bus Tributary: dalNbcsTswEnableCas

This function enables/disables the CAS processing of a specified SBI tributary.

Prototype

```
INT4 dalNbcsTswEnableCas(void* deviceInfo,
UINT1 dir, sNBCS_TRIB_SBI *pTrib, UINT1 enable)
```

Inputs `deviceInfo` : device information handle
`dir` : 0 = ingress,1 = egress
`pTrib` : (pointer to) SBI tributary
`enable` : 0 = disable, 1 = enable

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Enabling/Disabling SBI Bus Tributary Output: dalNbcsTswEnableSbiTribOutput

This function is valid only when the outgoing bus is configured for a quad SBI bus. The function will enable individual tributaries of the SBI bus to be driven on the outgoing bus.

Prototype

```
INT4 dalNbcsTswEnableSbiTribOutput(void*
deviceInfo, sNBCS_TRIB_SBI *pTrib, UINT1
enable)
```

Inputs `deviceInfo` : device information handle
`pTrib` : (pointer to) SBI tributary
`enable` : 0 = disable, 1 = enable

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Configuring the SBI Bus Tributary Mode: dalNbcsTswCfgSbiTribTransMode

This function configures the SBI bus tributary whether it is in transparent virtual tributary mode and/or justification request is enabled.

Prototype

```
INT4 dalNbcsTswCfgSbiTribTransMode(void* deviceInfo, UINT1 dir, sNBCS_CFG_PYLD_SBI *pTrib, UINT1 tvtEna, UINT1 justReqEna)
```

Inputs

`deviceInfo` : device information handle
`dir` : 0 = incoming parallel, 1 = outgoing parallel
`pTrib` : (pointer to) SBI tributary
`tvtEna` : 1 = tributary is a transparent virtual one
`justReqEna` : 0 = disable,1 = enable of justification request

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Configuring the C1 frame pulse delay: dalNbcsTswCfgC1fpDly

This function configures the delay of the C1 frame pulse in the time switch.

Prototype

```
INT4 dalNbcsTswCfgC1fpDly(void* deviceInfo, UINT2 dly)
```

Inputs

`deviceInfo` : device information handle
`dly` : C1 frame pulse delay

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

Controlling the CSU/DLL : dalNbcsTswCntlIntf

This function controls the clock synthesis unit (CSU) and the DLL units in the time switch device.

Prototype

```
INT4 dalNbcsTswCntlIntf (sNBCS_HNDL deviceHandle, sNBCS_CFG_INTF_CSU* pcntl)
```

Inputs

`deviceHandle` : device handle
`pcntl` : (pointer to) control structure

Outputs None

Returns Success = NBCS_SUCCESS
Failure = <NBCS error codes>

LIST OF TERMS

APPLICATION: Refers to protocol software used in a real system as well as validation software written to validate the Narrowband Chipset driver on a validation platform.

API (Application Programming Interface): Describes the connection between this module and the user's Application code.

ISR (Interrupt-Service Routine): A common function for intercepting and servicing device events. This function is kept as short as possible because an Interrupt preempts every other function starting the moment it occurs and gives the service function the highest priority while running. Data is collected, Interrupt indicators are cleared and the function ended.

DPR (Deferred-Processing Routine): This function is installed as a task, at a user configurable priority, that serves as the next logical step in Interrupt processing. Data that was collected by the ISR is analyzed and then calls are made into the application that inform it of the events that caused the ISR in the first place. Because this function is operating at the task level, the user can decide on its importance in the system, relative to other functions.

DEVICE: One Narrowband Chipset Integrated Circuit. There can be many devices, all served by this one driver module.

- DIV (Device Initialization Vector): Structure passed from the API to the device during initialization; it contains parameters that identify the specific modes and arrangements of the physical device being initialized.
- CSDDB (Chipset Device Data Block): Structure that holds the essential data for each device.

GROUP: An arbitrary collection of Narrowband devices. After defining a group, the user can use group-level functions to initialize or configure the devices in the group.

- GIV (GROUP Initialization Vector): Structure used during GROUP initialization; it contains several DIVs, which are used to initialize the devices in the GROUP.
- GDB (GROUP Data Block): Structure that holds pointers to the CSDDBs for each device in the GROUP.

MODULE: All of the code that is part of this driver, there is only one instance of this module connected to one or more Narrowband Chipset chips.

- MIV (Module Initialization Vector): Structure passed from the API to the module during initialization, it contains parameters that identify the specific characteristics of the driver module being initialized.
- CSMDB (Chipset Module Data Block): Structure that holds the Configuration Data for this module.

RTOS (Real-Time Operating System): The host for this driver.

DAL: Driver Abstraction Layer. This is a portable layer to accommodate devices other than SBS/NSE that provide the time and space switching functionality.

ACRONYMS

ADM: Add Drop Multiplexer

API: Application Programming Interface

ASAP: Any Service Any Port

CAS: Channel Associated Signaling

CRC: Cyclic Redundancy Check

CSD: Chipset Driver

CSDDB: Chipset Device Data Block

DAL: Driver Abstraction Layer

DIV: Device Initialization Vector

DPR: Deferred-Processing Routine

FIFO: First In, First Out

GDB: Group data block

GIV: Group initialization vector

ILC: In-band Link Controller

ISR: Interrupt-Service Routine

ISV: Interrupt-Service (routine) Vector

CSMDB: Chipset Module Data Block

MIV: Module Initialization Vector

NSE: Narrowband Switching Element

OPA: Open Path Algorithm

RTOS: Real-Time Operating System

SBI: Scalable Bandwidth Interconnect

SBS: SBI Bus Serializer

TCB: TeleCombus

INDEX

API Functions

Callback Functions

Constants

DAL Functions

Enumerated Types

Example Functions

Header File

Pointers

Porting Functions and Macros

Source Files

Structures

Variables