PM7326, PM7324, PM7350, PM7351

VORTEX CHIPSET DRIVER

DESIGN SPECIFICATION

PROPRIETARY AND CONFIDENTIAL

ADVANCE

ISSUE 3: MARCH 2001

REVISION HISTORY

Issue No.	Issue Date	Originator	Details of Change
Issue 1	Dec. 13, 1999	Keming Chen	Initial version
Issue 2	June 15, 2000	Keming Chen Shiraz Bhalwani	Updated data structure definitions and API descriptions to reflect the actual implementations. Added Porting Guide and Appendix A
Issue 3	March, 2001	Keming Chen	(1) Fixed an error in equation for calculating the shaping parameter, `ShpCdvt`, in Section 13.3. (2) Added a hardware-specific constant definition VCS_SYSCLK_FREQ_ATLAS to Porting Guide

.

PMC-Sierra, Inc.
105-8555 Baxter Place
Burnaby BC Canada V5A 4V7
Phone 604.415.6000, Fax 604.415.6200

Document ID: PMC-1991216 Issue 3

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 Scope

This document is the design specification for the VORTEX chipset (PM7326, PM7324, PM7350 and PM7351) driver software. It describes the features and functionality provided by the chipset driver, the software architecture, and the external interfaces of the chipset driver software. The document also describes how the chipset driver can be ported to a different platform.

1.2 Objectives

The main objectives of this document are as follows:

- Provide a detailed list of the functions supported by the chipset driver.
- Describe the software architecture of the chipset driver (data structures, algorithms, flow diagrams, component descriptions, etc…).
- Describe the external interfaces of the chipset driver. The external interfaces illustrate how the chipset driver interacts with the underlying devices and RTOS as well as external application/validation software.

1.3 Audience

This document has been created to ensure consistency across all of the software written by PMC and is intended for the following audience:

- Applications (when applicable): Applications should review the functions provided by the chipset driver and make sure that the driver's features meet their requirements for use on Applications hardware.
- Marketing: Marketing should review the driver's features and make sure it meets customers' requirements.
- Software Group: The Software group should use this document as a reference for implementing the VORTEX chipset driver.

1.4 References

Table 1: References

S/UNI-ATLAS Driver Manual	PMC-2000949	Issue 1	PMC-Sierra Inc.
S/UNI-ATLAS Long Form Data Sheet.	PMC-1971154	Issue 5	PMC-Sierra Inc.
S/UNI-ATLAS Long Form Data Sheet Errata.	PMC-1981505	Issue 1	PMC-Sierra Inc.
S/UNI-APEX Device Driver Manual	PMC-1991727	Issue 2	PMC-Sierra Inc.
S/UNI-APEX Engineering Document	PMC-1980448	Issue 3	PMC-Sierra Inc.
S/UNI-APEX Hardware Programmer's Guide	PMC-1991454	Issue 2	PMC-Sierra Inc.
S/UNI-DUPLEX Driver Manual	PMC-1990799	Issue 1	PMC-Sierra Inc.
S/UNI-DUPLEX Engineering Document	PMC-1980169	Issue 3	PMC-Sierra Inc.
S/UNI-VORTEX Driver Manual	PMC-1990786	Issue 1	PMC-Sierra Inc.
S/UNI- VORTEX Engineering Document	PMC-1980170	Issue 3	PMC-Sierra Inc.
DSLAM Reference Design: System Design	PMC-1990832	Issue 1	PMC-Sierra Inc.
DSLAM Reference Design: Core Card	PMC-1990815	Issue 1	PMC-Sierra Inc.
DSLAM Reference Design: Line Card	PMC-1990354	Issue 2	PMC-Sierra Inc.
DSLAM Reference Design: WAN Card	PMC-1990474	Issue 1	PMC-Sierra Inc.
B-ISDN OAM Principles and Function Abstract	I.610	Feb. 1999	ITU-T

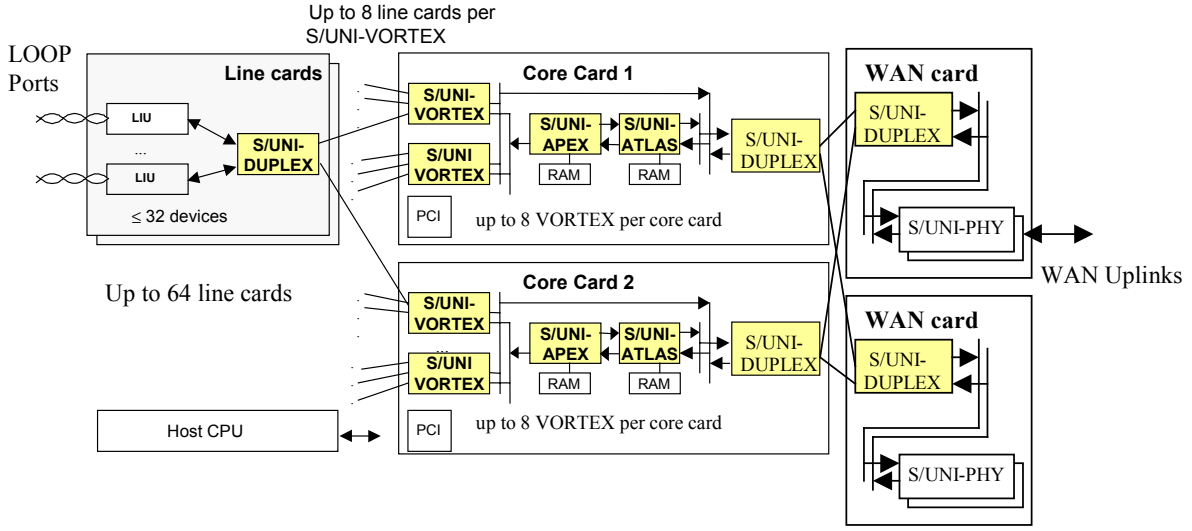
Traffic Management Specification	Af-tm-0056.000	Version 4.0	The ATM Forum

2 VORTEX CHIPSET OVERVIEW

This section briefly describes each VORTEX chipset device, and provides an overview of VORTEX chipset architecture in DSLAM application or other similar architecture system. This shall help understanding the chipset driver and its design architecture.

The VORTEX chipset is ideally suited for Digital Subscriber Line Access Multiplexers (DSLAMs) where the cell processing requirements are centralized on a single core card. Figure 1 depicts a scalable, cost effective DSLAM based on the PMC-Sierra S/UNI-VORTEX chip set.

Figure 1: Reference DSLAM Application



The DUPLEX (PM7351) and VORTEX (PM7350) provide non-statistical, flow controlled multiplexing of ATM cells over point to point high speed serial interconnect between a single centralized core card and up to 64 line cards. Each DUPLEX is capable of supporting up to 32 xDSL PHY devices via a UTOPIA L2 bus or providing termination of the ATM TC layer for up to 16 xDSL PHY devices with clock and data interfaces. The DUPLEX/VORTEX devices are capable of aggregating ATM traffic from 2048 xDSL PHY devices or Loop ports onto a single core card.

The ATLAS (PM7324) performs ATM layer functions including full space address resolution for both up and downstream traffic flows, cell rate policing for both up and downstream traffic flows and full ITU I.610 OAM cell processing for OAM flows on both the loop side and the WAN side.

The S/UNI APEX performs advanced ATM layer traffic management functions including cell switching, per VC queuing, and hierarchical (per VC, per Class of Service, and per port) scheduling and congestion management to up to 2048 loop ports and up to 4 WAN ports.

In a typical DSLAM application, the DSLAM system includes two redundant core cards, and multiple line cards and (up to 2) WAN cards, as shown in Figure 1. The two core cards are operating in either load-sharing or protection mode. In load-sharing mode, a core card acts as the working card for some line cards (typically half of them) and as the spare card for the remaining line cards. In the (1:1) protection mode, one core card is active, while the other one is left in a hot standby mode. If one of two core cards has a failure, the service for the existing connection ports or line cards can be switched and provided by the other core card with a minimum cell loss or corruption. See Ref. 11-14 (document # PMC-1990832, 990815, 990354, 990474) for a more detailed description of the reference design.

The chipset driver is built around the typical DSLAM architecture described in the reference design. A chipset card or core card consists of one APEX , one ATLAS device, multiple VORTEX chips, and one DUPLEX chip. However, the chipset driver is designed to be modular so that it can be easily ported for other different designs with fewer VORTEX or DUPLEX chips on a core card.

3 DRIVER FEATURES AND FUNCTIONALITY

The VORTEX Chipset Driver integrates the four underlying device drivers for the PMC VORTEX chipset devices (consisting of PM7324 S/UNI-ATLAS, PM7326 S/UNI-APEX, PM7350 S/UNI-DUPLEX and PM7351 S/UNI-VORTEX), and provides a synchronized access and control over the devices for DSLAM or other similar applications. The chipset driver directly monitors and controls chipset core cards, not the remote Line/WAN cards. However, it provides communication channels between the core cards and Line/WAN cards for remote control of the Line/WAN card. The definition of communication message content, or how the remote cards are controlled, is beyond the scope of this chipset driver. The following describes the functionality supported by the VORTEX chipset driver.

3.1 Module

Initialization and Shutdown

Allocates all memory needed by the chipset driver and initializes Module level data structures. It also initializes all underlying device driver modules.

Shuts down the chipset driver module gracefully after deleting all chipset (core cards) that are currently registered with the chipset driver.

3.2 Chipset

Chipset Control (Add and Delete)

Adding a chipset involves verifying that the chipset (core card) exists, allocating a memory buffer to store chipset context information, and associating a chipset handle to the user context passed by the Application. The chipset context buffer stores the device handles to each chipset device on the core card. The Application uses this chipset handle as a parameter in most of the API calls to refer to this particular chipset (core card). Reciprocally, the Chipset Driver uses this user context as a parameter when doing a callback to the Application code regarding that particular chipset (core card).

Deleting a chipset involves applying a reset to the chipset (core card) and releasing the chipset handle, as well as device handles within the chipset context.

Chipset Initialization

Initialization of a chipset involves initializing enough memory to store context information about the chipset card. The Chipset driver uses this context information to control and monitor the chipset card or underlying devices. Once the context memory is setup, the chipset driver invokes device initialization routine provided by each underlying device driver, and configures each device interface properly. This involves using the profile number passed by the Application to set the chipset card configuration.

A profile simply serves as a "canned configuration" that is used to initialize a chipset (card) without having to pass all the initialization parameters every time a chipset (card) is configured. Instead, the user passes a profile number, which is an index to an array of profiles that the USER is required to create. The chipset driver indexes this array, obtains the initialization vector corresponding to the profile number and configures the chipset (card) or chipset devices accordingly.

Chipset Reset

It supports two levels of reset:

- Reset the chipset hardware only, but all software context information and connection table are preserved. The card can be recovered to the pre-reset state and all connections can be quickly restored.
- Reset both hardware and software. It applies an internal (soft) reset to each VORTEX chipset device on the card, (or resets whole chipset card at once if the chipset card supports the feature). Also clears the context and statistics information for the devices and the chipset system. All connections will be destructed. The system has to be re-initialized from scratch.

De-Activate / Activate Chipset

The USER can de-activate and activate the operation of a chipset (card) at any time. After activation, the chipset (card) will start to handle cell traffic. The chipset driver still maintains the connection table after the de-activation.

Interrupt Servicing

Interrupt servicing of individual chipset device is implemented and provided by each underlying device driver. Typically, each driver provides an Interrupt Servicing Routine (ISR) and a deferred processing routine (DPR) for handling interrupts of the device.

The ISRs clear the interrupts raised by the devices and store the interrupt status for later processing by the deferred processing routine (DPR). The DPR runs in the context of a separate task within the RTOS and takes appropriate actions based on the interrupt status retrieved by the ISRs.

Interrupt servicing is an optional feature. The user can disable device interrupts and instead poll the device periodically to monitor status and check for alarm/error conditions.

Both polling and interrupt driven approaches detect a change in device status and report this status to a DPR. The DPR then invokes some callback functions based on the status information retrieved. The chipset driver implements the device-level callback functions of each device driver. The chipset driver may handle certain alarms internally, or relay some event conditions to the applications. In the latter case, the chipset driver provides chipset-level callback functions for passing the event information to the application.

Alarms, Status and Statistics

Routines are provided that read the various counts accumulated by the VORTEX chipset devices. These routines are generally called on a periodic basis by the application. It is also possible to set the chipset driver such that some counts are periodically read by the chipset driver watchdog task and presented to the application via a callback function, or stored in context memory for retrieval by the application. The application should ensure that the counters are polled at frequent intervals to prevent counters from rolling over. The various counts include Cell Counts for Tx and Rx, Discarded/Errored Cell Counts. The statistics are available per-VC, per-Port, and per-Chipset. The chipset driver provides an API to determine the VC connection ID of the last cell discarded.

In addition to system level statistics, individual driver statistical numbers, such as count of interrupts for a particular device, are maintained by each device driver. They can be retrieved by application through a chipset API.

Chipset self-test and Device Diagnostics

- Verifies each chipset device with simple read and write checks (register test) and validates the associate SRAM/SDRAM memory access. The chipset driver reports any error condition to the application.
- Conducts loopback tests for integrity check of the chipset card:
    - Set the chipset into a variety of loopback modes, and then insert test cells from an APEX microprocessor port, or Loop, or WAN port, and check to see if the same test cells are looped back to the same testing port. See Figure 22 of Appendix B which shows the loopback data flow via the microprocessor port.

Microprocessor OAM support

Certain types of OAM cells (such as Loopback, System Management and Activation/Deactivation) are not directly supported by the VORTEX chipset devices. However, such support can be provided by the chipset driver through Microprocessor OAM Interface Functions.

Microprocessor OAM Interface Functions configures, controls and monitors the Microprocessor interface of ATLAS device. Ingress OAM cells that are terminating at this VC/VP endpoint and that are NOT handled by the ATLAS device will automatically be passed to the Microprocessor Ingress Cell interface. The chipset driver is responsible for ensuring that cells are extracted from this interface in a timely manner. The chipset driver processes the Ingress OAM requests for OAM functions from a remote system. It may generate some response OAM cells, which are presented to the Microprocessor Egress Cell Interface for transmission, as shown in Appendix B, Figure 26.

Scheduling and Congestion Control Service

Utility routines are provided to calculate appropriate congestion threshold levels, scheduling and cell rate policing parameters for service classes defined by TM 4.0 (CBR, VBR, VBR-RT, GFR, UBR). These functions are normally called by the chipset driver APIs, and therefore should be considered as internal library functions. However, USER may implement them using a different algorithm for the scheduling and congestion control.

3.3 Application Programming Interface

The chipset driver provides some system level API routines to support VC/VP Connection setup/maintenance/ teardown, QOS service, multicasting support, OAM setup and processing, Performance Monitoring (PM) setup and maintenance, Cell rate policing.

WAN-port to Loop-port connection (Upstream/downstream)

- Connection Setup: The connection request contains traffic parameters such as bandwidth and QOS service parameters. The chipset driver's Resource Management determines whether the request should be honored or rejected. The chipset driver is responsible for configuring the appropriate devices for the connection establishment. . Figure 17 and Figure 18 of Appendix B show the data flow paths for the upstream and downstream connections, respectively.
- Connection Teardown: Tears down a connection by re-configuring the appropriate devices, and de-allocate the related resource back to the Resource Management.
- Port Setup and Teardown: Setup port or teardown a port, which clears all the VCs associated with the port.
- Connection Traffic Parameter Retrieval/Modification: Retrieves the traffic descriptor parameters for a specified connection, and/or changes the traffic parameters without tearing down the connection.
- Connection Activation/ Deactivation(or Standby): suspends a VC connection, or activates the connection to a normal operation.
- Connection Status Monitoring: reports connection status.

Loop to Loop-port connection

- Connection Setup: The connection request contains traffic parameters such as bandwidth and QOS service parameters. The chipset driver's Resource Management determines whether the request should be honored or rejected. The chipset driver is responsible for configuring the appropriate devices for the connection establishment. Figure 19 of Appendix shows its data flow path within the chipset.
- Connection Teardown: Tears down a connection by re-configuring the appropriate devices, and de-allocates the related resource back to the Resource Management.
- Port Setup and Teardown: Setup port or teardown a port, which clears all the VCs associated with the port.

- Connection Traffic Parameter Retrieval/Modification - It retrieves the traffic descriptor parameters for a specified connection, and/or changes the traffic parameters without tearing down the connection.
- Connection Activation/ Deactivation(or Standby): suspends a VC connection, or activates the connection to a normal operation.
- Connection Status Monitoring: reports connection status.

Microprocessor-port to WAN/Loop port connection

- Connection Setup: The connection request contains traffic parameters such as bandwidth and QOS service parameters. The chipset driver's Resource Management determines whether the request should be honored or rejected. The chipset driver is responsible for configuring the appropriate devices for the connection establishment. Figure 20 of Appendix B shows the data flow path between Microprocessor port and a WAN port, and Figure 21 shows the data flow path between Microprocessor port and a Loop port
- Connection Teardown: Tears down a connection by re-configuring the appropriate devices, and de-allocates the related resource back to the Resource Management.
- Port Setup and Teardown: Setup port or teardown a port, which clears all the VCs associated with the port.
- Connection Traffic Parameter Retrieval/Modification: Retrieves the traffic descriptor parameters for a specified connection, and/or changes the traffic parameters without tearing down the connection.
- Receive/Transmit Data (via Microprocessor port of APEX device).

Multi-casting support

With software or driver assistance, cells come in from WAN or loop port, are replicated across a list of destination VCs via the APEX microprocessor port, as shown in Figure 24 of Appendix B. Even after a multicasting group is setup, connections can be dynamically added to or dropped from the group.

Inband Control Channel

- Channel Setup: The channel request specifies a destination line/WAN card, VPI/VCI as well as AAL type (AAL0, raw cell or AAL5). The chipset driver allocates resources such as memory buffer for the connection channel, and establishes the channel communication via a HSS link between a VORTEX or DUPLEX device on core card and the DUPLEX device on the line/WAN card. For the channels between a line card and cord card, the chipset driver passes the message through the microprocessor port of the APEX on the core card. For the channels between a WAN card and core card, the messages are routed through microprocessor port of DUPLEX on the core card. Their data flow paths are shown in Figure 23 of Appendix B.
- Channel Teardown: Tears down the channel connection and de-allocates the related resource back to the Resource Management.
- Receive/Transmit Message: Sends a message out over a specified channel, or receives messages from the line/WAN cards.

BOC signaling

BOC signaling over the HSS links between the core card and Line/WAN cards provides a simple communication channel between the core card and a remote card. The BOC code, including Reset and user-defined code, can be sent to or received from the VORTEX or DUPLEX device which is directly connected to the line/WAN card via a HSS link cable.

Retrieving Current VC Connections and Resources

The chipset driver provides API functions to reports the current VC connection status, as well as available resources, such as available VCs, logical channels and ports.

FM function (RDI, AIS, CC) Setup

Setup OAM service through ATLAS device driver. The APEX is configured in a way that all OAM cells can pass through transparently. The OAM data path is shown in Figure 25.

The F4 and F5 OAM cell sourcing/termination can be setup per VC. F4 to F5 processing is also supported.

Performance Monitoring Setup

Individual VC can be configured and associated to one or two of 256 PM sessions for full performance monitoring.

The Application shall be responsible to read the performance information gathered through the chipset driver.

Protection switching

The chipset driver supports the hot switching of the operating modes of two redundant core cards by bringing a spare core card to an active mode, while switching the active core card into a standby mode. The switching procedures are optimized to minimize the cell corruption or cell loss.

It also supports the hot switching of line card connections between two active core cards which operates in a load-sharing mode. A line card and its associated VC connections which were originally serviced by one of two redundant core cards, can be serviced by the other core card after a load switch or transfer.

Addition/deletion of Line cards, WAN card

It manages the addition or removal of line or WAN card. The resource manager updates its resource database to reflect the change in the loop/WAN port availability. The port or connection setup is prevented when the associated line/WAN card is absent from the system.

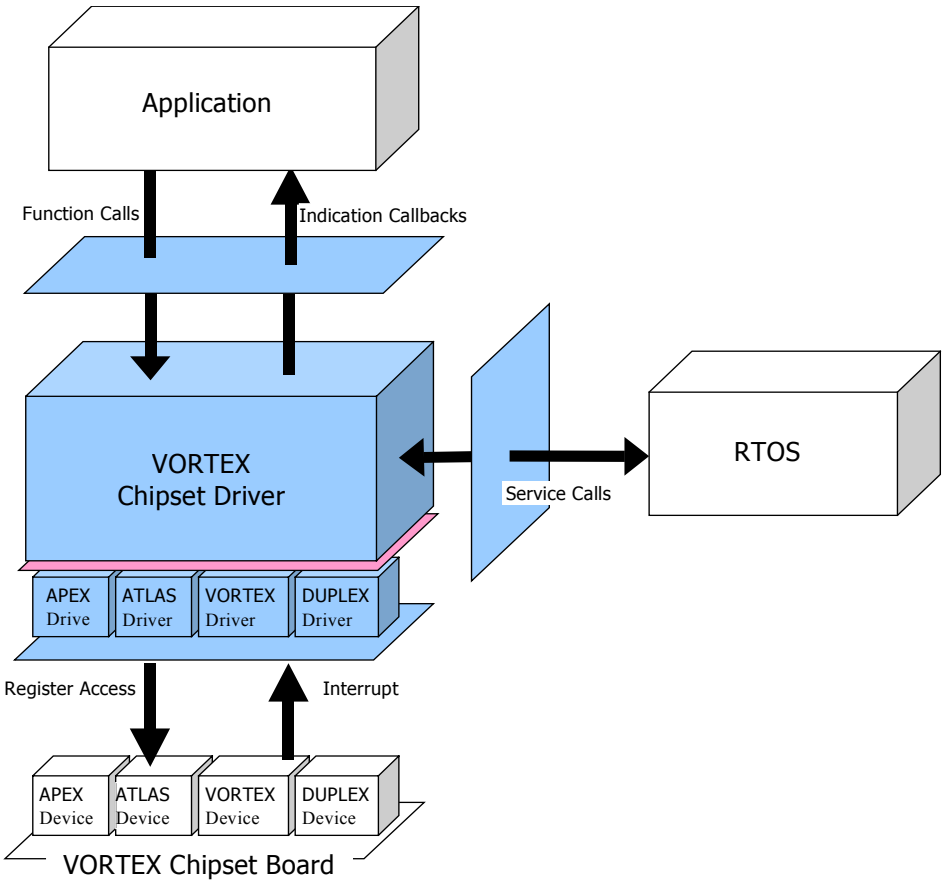
4 ARCHITECTURE OVERVIEW

This section provides an overview of the VORTEX chipset driver's architecture. The chipset driver's external interfaces and its main components are briefly described here.

4.1 External Interfaces

Figure 2 illustrates the external interfaces defined for the VORTEX chipset driver.

Figure 2: External and Internal interfaces



VORTEX chipset Hardware Interface

The hardware interface consists of routines that allow the VORTEX chipset driver to interact with the underlying VORTEX chipset devices. These routines provide read/write access and interrupt handling services. The implementation of these routines is system-dependent. Therefore, the USER typically implements these routines when porting the chipset driver to a specific platform. A reference implementation as well as detailed documentation is provided to help facilitate the implementation of these routines. The reader is referred to Section 6 for further details on the hardware interface routines.

RTOS Interface

The RTOS interface consists of the RTOS services required by the VORTEX chipset driver. The chipset driver requires the following RTOS services:

- Memory: allocate, free
- Timers: sleep
- Semaphores: create, set, clear, delete

The RTOS service calls vary from one RTOS to another. In order to minimize the porting effort (from one RTOS to another), the chipset driver abstracts these service calls using a set of "wrapper" routines. The USER only has to modify these routines while porting the chipset driver to a specific RTOS. For more details on the RTOS interface, the reader is referred to Section 7.

Application Programming Interface

The term "Application" in this document refers to protocol software used in a real system as well as validation software written to validate the VORTEX chipset driver on a validation platform. The Application interfaces with the VORTEX chipset driver via the Application Programming Interface (API). The API consists of functions and indication callback routines.

The application software calls the API functions to perform specific operations on the VORTEX chipset. The API functions typically are not executed in the context of a separate task within the RTOS. Instead, they are executed in the context of the calling software's task. It is important to note that the API functions are not to be modified by the USER. These functions are not platform or RTOS dependent and therefore should remain unchanged during the porting process.

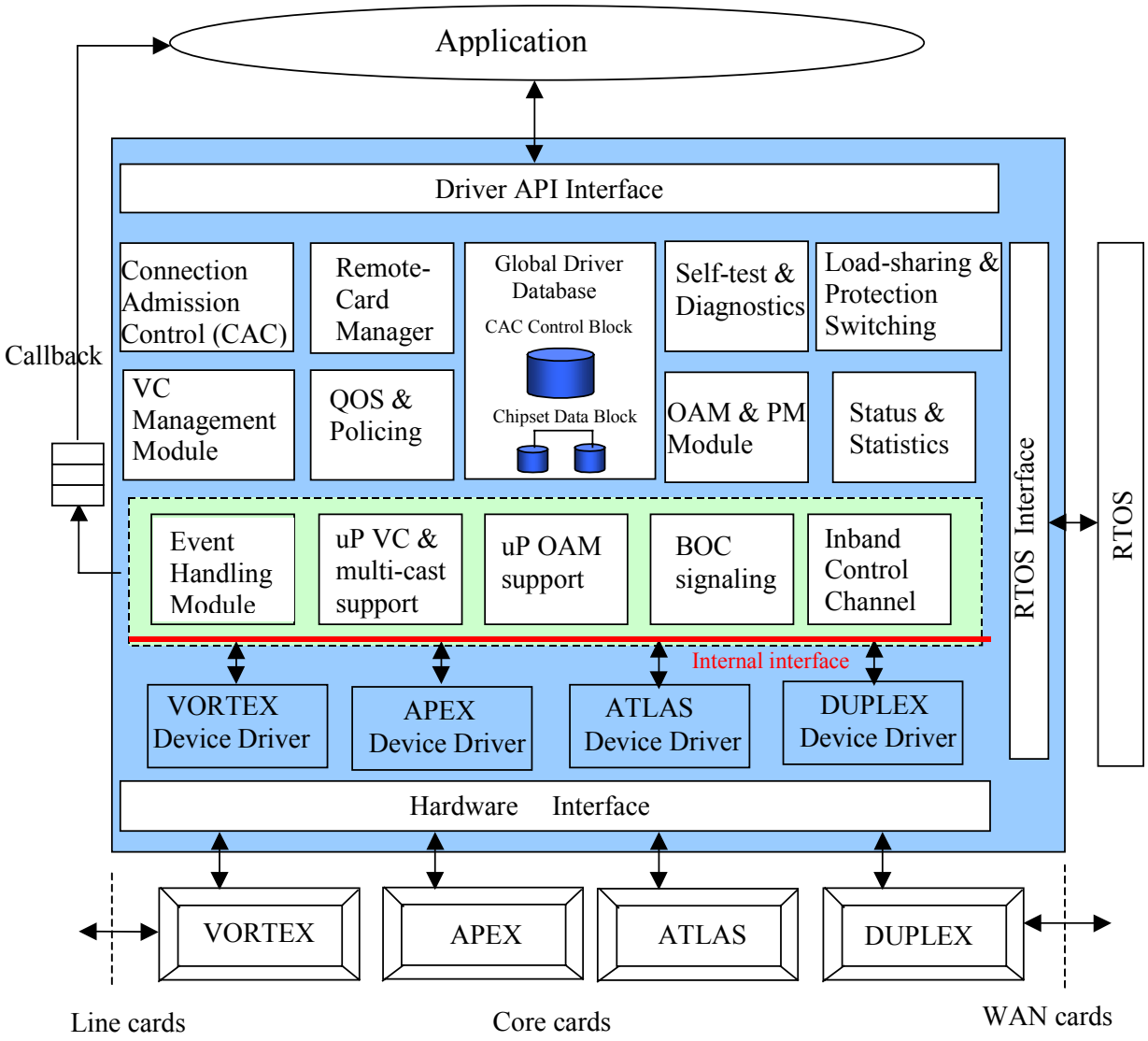
The callback routines are used by the chipset driver to notify the application of events within the device(s) (such as alarms). The callback routines, unlike API functions, are system-dependent and are implemented by the USER.

For more detailed information on the API functions and callback routines, please see Section 8.

4.2 Main Components

Figure 3 illustrates the main components of the VORTEX chipset driver.

Figure 3: Main Components



Global Driver Database (GDD)

The Global Driver Database (GDD) is the top layer data structures, created by the VORTEX chipset driver to keep track of its initialization and operating parameters, modes and dynamic data. The GDD is allocated via an RTOS call, when the chipset driver is first initialized and contains all the Chipset Structures.

The Chipset Data Block (CDB) is contained in the GDD and initialized by the VORTEX chipset Module for each Chipset card that is registered, to keep track of that Chipset's initialization and operating parameters, modes and dynamic data. There is a limit on the number of Chipset Blocks (Devices) and that limit is set by the USER when the Module is initialized.

The GDD also contains the Connection Admission Control (CAC) data block. It consists of VC connection table, multicast groups, and inband control channel information. The structure is mostly used by CAC control Module, and Inband Control Channel Module.

CAC Control Module

The Connection Admission Control Module manages and maintains the system resources such as VC connections and traffic bandwidth. The module determines whether a User request for a connection or channel establishment should be honored or rejected, based on the availability of resources. All the resource information is stored in the CAC data block.

Status & Statistics Module

The Status and Statistics Section is responsible for tracking chipset status information and accumulating statistical counts for each chipset registered with (added to) the chipset driver. This information is stored for retrieval by the application software.

VC Management Module

The VC Management Module provides routines to configure each chipset device for VC connection setup, modification, and teardown. The VC Management module configures the devices of the chipset in different ways depending on the type of connection being set up. In addition, the CAC data block or the VC table is updated each time the service routines are called.

VC QOS & Policing Module

The module calculates the scheduling parameters and congestion threshold levels for VC, Classes and Ports, as well as the cell rate policing parameters based on QOS contract. It is also responsible for manipulating the Queue Engine Schedulers in APEX and configuring the rate policing parameters in ATLAS.

VC OAM & PM Module

The VC OAM & PM module is responsible for configuring individual VCs for OAM support by ATLAS, and full performance monitoring.

Remote-card Manager Module

The module manages the addition or deletion of remote Line/WAN cards, and keeps track of the port availability.

Self-test & Diagnostics Module

The module performs the self-test, such as register and memory port test. It also provides routines to prepare the chipset into a loopback mode for the integrity check purpose.

Load-sharing & Protection Switching

The module manages the load-sharing of the connections or traffics between two redundant chipset card. It provides service for the hot switching of active and spare cards.

Event Handling Module

The Event Handling Module is responsible for handling the event raised by the underlying devices or device drivers. Depending on the type of events, the module may pass the event information directly to the Application.

Microprocessor VC & Multicast Module

The module performs the data transmission/receiving of cells or frame to/from the Microprocessor VC connections. It supports multicasting of VC cells, in which cells coming in from a WAN or Loop port are replicated across a list of destination VCs in a multicast group. The module uses the SAR Assist features of the APEX device to perform the insertion/extraction of cells through its microprocessor interface.

Microprocessor OAM Support Module

The module performs the support for certain type of OAM cells (Loopback, Activation/Deactivation), which are not supported by the ATLAS device.

Inband Control Channel (ICC) Module

The module provides services for the inband control channel messaging between a remote Line/WAN card and a chipset core card. The module is transparent to the message content. Therefore it's up to User to compose and interpret the messages.

BOC Signaling Module

The module provides a simple communication path between a remote Line/WAN card and a chipset core card. The module is transparent to the user BOC code. Therefore it's up to User to define and interpret the BOC code.

Driver API

The Driver Application Programming Interface (API) is a list of high-level functions that can be invoked by application programmers to configure, control and monitor the VORTEX chipset devices. The API functions perform operations that are more meaningful from a system's perspective. The API includes functions that initialize the devices, perform diagnostic tests, validate configuration information to prevent incorrect configuration of the devices, retrieve status and statistics information, and setup/modify/teardown VC connections. The chipset driver API functions use the services of the other driver modules to provide this system-level functionality to the application programmer.

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

Hardware Interface

The Hardware Interface is a list of low-level functions that are invoked by the device drivers to access the VORTEX chipset registers. The Hardware Interface functions are architecture-dependent and are to be implemented by the USER when porting the chipset driver code to a specific platform.

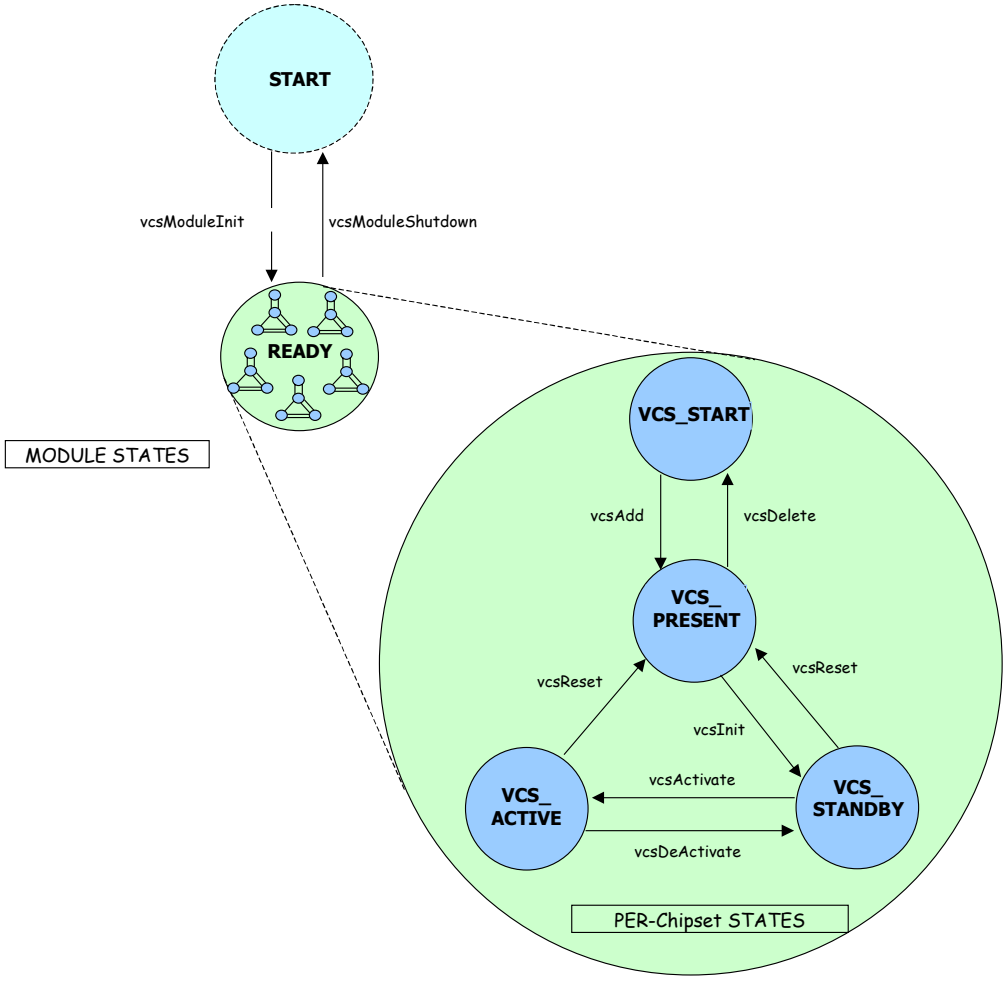
RTOS Interface

The RTOS Interface is a list of low-level functions that are invoked by the device driver itself to allocate or free RTOS resources. The RTOS Interface functions are RTOS-dependent and are to be implemented by the USER when porting the chipset driver code to a specific platform.

4.3 Software State Description

Figure 4 shows the software state diagrams for the VORTEX chipset module and device(s) as maintained by the chipset driver.

Figure 4: State Diagram



State transitions are made on the successful execution of the corresponding transition routines shown. State information helps maintain the integrity of the GDD and CDB(s) by controlling the set of operations that are allowed in each state.

VORTEX chipset Module States

The following is a description of the VORTEX chipset module states.

START

The VORTEX chipset driver Module has not been initialized. The only API function that will be accepted in this state is `vcsModuleInit`. In this state the chipset driver does not hold any RTOS resources (memory, timers, etc…), has no running tasks, and performs no actions.

READY

This is the normal operating state for the chipset driver Module. The VORTEX chipset driver Module has been initialized successfully via the API function `vcsModuleInit`. The Module Initialization Vector (MIV) has been validated, the Global Driver Database (GDD) has been allocated and loaded with current data, the per-chipset data structures have been allocated, and the RTOS has responded without error to all the requests sent to it by the chipset driver.

The chipset driver is ready for chipsets to be added. The chipset driver Module remains in this state while chipsets are in operation. Chipsets can be added via `vcsAdd`. The API function accepted here for Module control is `vcsModuleShutdown`.

VORTEX chipset States

The following is a description of the per-chipset states.

VCS_START

The VORTEX chipset (card) has not been initialized. The only API function that will be accepted in this state is `vcsAdd`. In this state the chipset (card) is unknown by the chipset driver and performs no actions.

VCS_PRESENT

The VORTEX chipset card has been successfully added via the API function `vcsAdd`. A Chipset Data Block (CDB) has been associated to the card and updated with the user context, and a card handle has been given to the USER. In this state, the card performs no actions. The only API functions that will be accepted in this state are `vcsInit` and `vcsDelete`.

VCS_STANDBY

This state is entered via the `vcsInit` and `vcsDeActivate` function calls. In this state the Chipset Card remains configured but all data functions are de-activated including interrupts and Alarms, Status and Statistics functions. `vcsActivate` will return the chipset to the `VCS_ACTIVE` state, while `vcsReset` will de-configure the Chipset.

VCS_ACTIVE

This is the normal operating state for the Chipset Card(s). State changes can be initiated from the `VCS_ACTIVE` state via `vcsDeActivate,` and `vcsReset.`

5 CONSTANTS, AND DATA STRUCTURES

This section describes the elements of the chipset driver that configure or control its behavior and therefore should be of interest to the application programmer. For more information on our naming convention, the reader is referred to Section 12.

5.1 Constants

The following Constants are used throughout the chipset driver code:

- <VCS_ERR_CODES>: a list of error codes used throughout the chipset driver code, returned by the API.
- VCS_MAX_CARDS: defines the maximum number of chipset core cards that can be supported by this chipset driver. This constant must not be changed without a thorough analysis of the consequences to the chipset driver code. It should be either 1 or 2. The default value is 2.
- VCS_MAX_VC: define the maximum number of VCs the system supports. It depends on the card's context memory capacity such as APEX SSRAM size. It should be either 16K or 64K. The default value is 16K.
- VCS_MAX_LOOP_PORTS: define the maximum number of Loop ports the chipset driver supports. The default value is 2048 (2K).
- VCS_MAX_WAN_PORTS: define the maximum number of WAN ports the chipset driver supports. The default value is 4.
- VCS_MAX_CELL_RATE: define the maximum traffic throughput in half duplex in cells per second. The default value is (1420*1024).
- VCS_MAX_CELL_RATE_PER_LOOP: define the maximum traffic throughput for a loop port in cells per second. The default value is (230*1024).
- VCS_MAX_VORTEX: define the maximum number of VORTEX devices on a core card (from 1 to 8). The default value is 2.

5.2 General Structure Definition

These structures are defined for general use by the application and the chipset driver.

Table 2: VORTEX chipset VPI and VCI (sVCS_VPI_VCI)

Field Type	Field Name	Field Description
UINT2	vpi	VPI of ATM cells
UINT2	vci	VCI of ATM cells

Table 3: VORTEX chipset VC and Port Descriptor (sVCS_VC_PORT_DES)

Field Type	Field Name	Field Description
eVCS_PORT_TYPE	ePortType	specifies a port type: VCS_LOOP_PORT, VCS_WAN_PORT, VCS_UP_PORT.
UINT2	u2PortNum	Specify a Loop or WAN port number
UINT2	vpi	VPI value
UINT2	vci	VCI value

Table 4: VC QOS Structure (sVCS_VC_QOS)

Field Type	Field Name	Field Description
eVCS_TRAFFIC_TYPE	eTrfcType	Indicates the VC type: CBR, rtVBR, nrtVBR, GFR, UBR, ABR
UINT4	Pcr	Peak cell rate in cells/second
UINT4	Scr	Sustained Cell Rate in cells/second
UINT4	Mcr	Minimum Cell Rate in cells/second, used for ABR type VC
UINT2	Mbs	Maximum Burst Size at the Peak Cell Rate in cells

Field Type	Field Name	Field Description
UINT2	Mfs	Maximum Frame Size in bytes , used in GFR
UINT2	Cdvt	Cell Delay Variation Tolerance (CDVT) in microsecond
UINT2	Clr	Cell Loss Ratio in percentage
UINT2	maxCTD	Maximum Cell Transfer Delay in cells
UINT1	u1NcAction	Specifies an action for non-compliant cells at ATLAS level, can be one of VCS_PLC_COUNT_ONLY (increment NC cell count) VCS_PLC_TAG_ONLY (tag CLP0 cell) VCS_PLC_TAG_DISCARD (tag CLP0, discard CLP1) VCS_PLC_DISCARD (discard both CLP0 and CLP1) VCS_PLC_DEFAULT (using default policing actions defined by the chipset driver)

Table 5: VC FM Structure (sVCS_VC_OAM_FM)

Field Type	Field Name	Field Description
UINT1	u1EndPoint	Bit 1: 1= terminating segment OAM cells, 0 = pass through Bit 0: 1= terminating end-to-end OAM cells, 0 = pass through

Field Type	Field Name	Field Description
UINT1	u1Config	OAM configuration Bit 7: reserved Bit 6: Send_AIS_segment Bit 5: send_AIS_end-to-end Bit 4: send_RDI_segment Bit 3: send_RDI_end-to-end Bit 2: CC_RDI Bit 1: CC_ACTIVATE_Segemnt Bit 0: CC_ACTIVATE_end-to-end
UINT1	u1DtSelect	Bit 0-3: select one of 16 defect types to be inserted in OAM (AIS, RDI) cells generated by the chipset.

Table 6: VC PM Structure (sVCS_VC_OAM_PM)

Field Type	Field Name	Field Description
UINT1	u1PMId1	Bits [6:0]: the PM session address in bank 1. Bit 7: active flag. If 1, it indicates the PM session is active.
UINT1	u1PMId2	Bits [6:0]: the PM session address in bank 2 Bit 7: active flag. If 1, it indicates the PM session is active.

Table 7: VC Policing Structure (sVCS_VC_POLICING)

Field Type	Field Name	Field Description
UINT2	u2Limit	Limit field for Generic Cell Rate Algorithm parameters.
UINT2	u2Incr	Increment field for Generic Cell Rate Algorithm parameters.

Table 8: Congestion Threshold Level Structure (sVCS_THRSH_LEVEL)

Field Type	Field Name	Field Description
UINT1	u1CLP0Thrsh	Maximum threshold for CLP0 cells.
UINT1	u1CLP1Thrsh	Maximum threshold for CLP1 cells.
UINT1	u1MaxThrsh	Maximum threshold for all cells.

Table 9: Port Threshold Structure (sVCS_PORT_THRSH)

Field Type	Field Name	Field Description
sVCS_THRSH_LEVEL	sPortThrsh	Maximum threshold levels per port. The port threshold levels are coded as 4 bit logarithmic and 4 bit fractional values
sVCS_THRSH_LEVEL	sClassThrsh[4]	(array of) Maximum threshold levels per Class. The class threshold levels are coded as 4 bit logarithmic and 4 bit fractional values.

Table 10: VC Threshold Structure (sVCS_VC_THRSH)

Field Type	Field Name	Field Description
sVCS_THRSH_LEVEL	sVcThrsh	Maximum threshold levels per VC The threshold levels are coded as 4 bit logarithmic and 2 bit fractional values.
UINT1	u1VcCLP0MinThrsh	Minimum number of CLP0 cells guaranteed to be allowed on a per-VC basis. This threshold value is coded as a 3 bit code value

Table 11: Shaped VC Parameters (sVCS_VC_SHPR)

Field Type	Member Name	Description
UINT1	u1ShpPrescale	Resolution of the Incr field
UINT2	u2ShpLateBits	Number of bits used to represent ShpTxSlotsLate
UINT2	u2ShpCdvt	CDVT for the connection
UINT2	u2ShpIncr	Increment field for SCR-GCRA

Table 12: Shaper Control VECTOR (sVCS_ SHPR_VECTOR)

Field Type	Member Name	Description
UINT1	state	State for the shaper: UNUSED or USED (ENABLED).
UINT1	u1PortClass	the port/class to which the shaper is applied. Bit 0-1: port number bit 2-3: Class number
UINT1	u1ShpSlowDownEn	Enable the slow down of the time reference clock used by the shaper.
UINT1	u1ShpThrshEn	Defines the method of speeding up/slowing down the shaper rate.
UINT1	u1ShpMeasInt	Define absolute number of clock cycles over which to measure congestion levels for the shaper.
UINT1	u1ShpThrshVal	Class queue length threshold
UINT1	u1ShpRedFact	Shaper slow down factor
UINT1	u1ShpRTRate	The maximum shaped data rate in clocks/timeslot

Table 13: VC OAM Defect Structure (sVCS_VC_OAM_DEFECT)

Field Type	Member Name	Description
UINT1	u1SegDefType	Received Segment Defect Type
UINT1	u1E2EDefType	Received End-to-end Defect Type

Field Type	Member Name	Description
UINT4	u1E2EdefLocation[4]	Received End-to-end AIS Defect Location. Total 128 bits.
UINT4	u1SegDefLocation[4]	Received Segment AIS Defect Location. Total 128 bits.

Table 14: VC Connection Status Structure (sVCS_CONN_STATUS)

Field Type	Member Name	Description
eVCS_VC_STATE	state	VC state; can be one of the following: VC_UNUSED, VC_STANDBY or VC_ACTIVE
eVCS_VC_CLASS	eVcClass	Bit 0-1: VC class; 00 = Class 0, 01 = Class 1, 10 = Class 2 11= Class 3.
eVCS_VC_TYPE	eVcType	VC connection type; 00 = VCC Cells, 01 = VCC Frame, 10 = VPC Cell, 11=VPC Frame
sVCS_VC_VECTOR	VcIn	specifies incoming cell ID, (VcPortID/vpi/vci)
sVCS_VC_VECTOR	VcOut	specifies outgoing cell ID, (VcPortID/vpi/vci)
UINT1	CardID	specifies an active chipset card, through which the VC cells pass.
sVCS_VC_THRSH	sVcThrsh	Contains threshold values for the VC connection
sVCS_VC_QOS	sQos	specifies QOS parameters for the VC, including peak cell rate, VC type (CBR, VBR etc)

Field Type	Member Name	Description
UINT1	u1VcWeight	VC WFQ weight (linear encoding - 6 bits)
sVCS_VC_OAM	sVcOAM	Contains OAM configuration for the VC.
UINT1	mcFlag	1=the connection belongs to a multicasting group 0=the connection doesn't belong to a multicasting group
UINT2	mcId	multicasting group ID, if the above flag is 1

Table 15: VC Cell Header Structure (sVCS_CELL_HDR)

Field Type	Member Name	Description
UINT1	u1Hdr[4]	4 Cell Header bytes

5.3 Structures Passed by the Application

These structures are defined for use by the application and are passed by reference to functions within the chipset driver.

Module Initialization Vector (MIV)

Passed via the `vcsModuleInit` call, this structure contains all the information needed by the chipset driver to initialize and connect to the RTOS.

- `maxVCs` specifies the maximum number of VCs the chipset driver needs to support.
- `maxChnls` specifies the maximum number of inband control channels the chipset driver shall support.

Since the control channel requires two VC connections for each channel, the VC Record Context resource of APEX and ATLAS are shared between the User's VC connections and Control Channels. Therefore, the values of `maxVCs` and `maxChnls` are limited by the following relationship: `maxVCs` + 2 * maxChnls ≤ VCS_MAX_VC

Table 16: VORTEX chipset Module Initialization Vector (sVCS_MIV)

Field Type	Field Name	Field Description
UINT2	maxVCs	Maximum number of user VCs supported by the chipset driver
UINT2	maxChnls	Maximum number of inband control channels
UINT2	maxInitProfs	Maximum number of initialization profiles
VCS_IND_RX_CELL	indRxDataCell	Callback function pointer for cell Rx on user connections
VCS_IND_RX_FRM	indRxDataFrm	Callback function pointer for frame Rx on user connections
VCS_IND_RX_CTRL_MSG	indRxCtrlMsg	Callback function pointer for message Rx on control channels
VCS_IND_RX_BOC	indRxBOC	Callback function pointer for BOC code Rx

Field Type	Field Name	Field Description
VCS_IND_RX_OAM	indRxOAM	Callback function pointer for OAM (CC and Activation/Deactivation) cell Rx
VCS_IND_COS_STATUS	indCosStatus	Callback function pointer for Change of Status on OAM operation
VCS_IND_INTR	indCritical	Callback function pointer for critical interrupt events
VCS_IND_INTR	indError	Callback function pointer for non-critical interrupt events

Chipset Initialization Vector

Passed via the vcsInit call, this structure contains all the information needed by the chipset driver to initialize (eventually activate) a VORTEX chipset card.

- valid indicates if this is a validated Initialization Vector or not.
- sDevInitVector (where Dev denotes Apx, Atls, Dpx, Vtx) contains the initialization vector for each VORTEX chipset device on the chipset card.

Table 17: VORTEX chipset Initialization Vector (sVCS_INIT_VECTOR)

Field Type	Field Name	Field Description
UINT4	valid	VCS_VALID – indicates that the contents of this vector are validated (VCS_INVALID otherwise)
sAPX_INIT_VECT	sApxInitVect	an Initialization vector for APEX chip
sATLS_INIT_VECT	sAtlsInitVect	an Initialization vector for ATLAS chip
sVTX_INIT_VECTOR	sVtxInitVect	an Initialization vector for VORTEX chip
sDPX_INIT_VECTOR	sDpxInitVect	an Initialization vector for DUPLEX chip

VC Connection Request

Passed via the `vcsConnSetup` call. It contains the necessary information to set up a VC connection.

Table 18: VORTEX chipset VC Request (sVCS_CONN_REQUEST)

Field Type	Field Name	Field Description
`sVCS_VC_PORT_DES`	`InVC`	specifies incoming port and VPI/VCI
`sVCS_VC_PORT_DES`	`OutVC`	specifies output port and VPI/VCI
`sVCS_VC_QOS`	`sQos`	specifies QOS parameters for the VC, including peak cell rate, VC type (CBR, VBR etc)
`eVCS_VC_TYPE`	`eVcType`	bit 0: cell type, Frame or Cell; 1 = Frame, 0 = Cells bit 1: VC type, VPC or VCC; 1 = VPC, 0 = VCC

Port-level Threshold Request

Passed via the `vcsPortSetup` call. It contains port-level threshold request.

Table 19: Port-level Threshold Request (sVCS_PORT_THRSH_REQUEST)

Field Type	Field Name	Field Description
`UINT4`	`u4CLP0Thrsh`	CLP0 per-port threshold level in cells
`UINT4`	`u4CLP1Thrsh`	CLP1 per-port threshold level in cells
`UINT4`	`u4MaxThrsh`	maximum per-port threshold level in cells
`UINT4`	`u4MinThrsh`	minimum guaranteed port threshold level in cells. The value is used by driver software to maintain cell buffer "guarantee" for the port.

VC Multicast Request

Passed via the `vcsMcSetup` call. It contains the necessary information to set up a multicasting group.

Table 20: VORTEX chipset Multicast Request (sVCS_MULTICAST_REQUEST)

Field Type	Field Name	Field Description
`sVCS_VC_PORT_DES`	`InVC`	specifies incoming port and VC
`sVCS_VC_PORT_DES *`	`pOutVC`	Pointer to a list of output ports and VCs
`UINT2`	`numOutVC`	Number of output VCs in the list for multicasting
`sVCS_VC_QOS`	`sQos`	specifies QOS parameters for the VC, including peak cell rate, VC type (CBR, VBR etc)
`UINT1`	`flagFCQ`	Data type: Frame or Cell. 1 = Frame, 0 = Cells

Inband Control Channel Request

Passed via the `vcsChnlSetup` call. It contains the information to set up a control channel between a core card and a WAN/Line card.

Table 21: VORTEX chipset Channel Request (sVCS_CHNL_REQUEST)

Field Type	Field Name	Field Description
`UINT1`	`cardId`	specifies which HSS line, and which VORTEX or DUPLEX device is connected to the Line or WAN card bit 0-2: HSS link number (0 to 7) bit 3-6: device number bit7 : 0=VORTEX, 1 = DUPLEX
`UINT2`	`VpiOut`	VPI for output message cells towards remote cards

Field Type	Field Name	Field Description
UINT2	VciOut	VCI for output message cells towards remote cards
UINT2	VpiIn	VPI for incoming message cells from remote cards
UINT2	VciIn	VCI for incoming message cells from remote cards
UINT4	maxMsgSz	maximum message size (number of bytes)
UINT1	flagFCQ	Data type: Frame or Cell. 1 = Frame, 0 = Cells

VC OAM (FM and PM) Setup Request

Passed via the `vcsVcOAMSetup` call. It contains the information to setup OAM and PM configuration on a VC connection.

Table 22: VC OAM Structure (sVCS_VC_OAM_REQUEST)

Field Type	Field Name	Field Description
`sVCS_VC_OAM_FM`	`sFMcfg`	Contains the FM configuration for the VC
`sVCS_VC_OAM_PM`	`sPMcfg`	Contains the PM configuration for the VC

Device ID

Used to specify a particular device on the chipset core card.

Table 23: Device Identification Structure (sVCS_DEV_ID)

Field Type	Field Name	Field Description
`eVCS_DEV_TYPE`	`eDevType`	Device Type, can be `VCS_APEX, VCS_ATLAS, VCS_VORTEX, VCS_DUPLEX.`
`UINT1`	`u1DevNum`	Specify one of multiple VORTEX devices on the core card. It ranges from 0 to `(VCS_MAX_VORTEXS -1)`

Port ID

Used to specify a particular port.

Table 24: Port Identification Structure (sVCS_PORT_ID)

Field Type	Field Name	Field Description
`eVCS_PORT_TYPE`	`ePortType`	Port Type, can be `VCS_LOOP_PORT, VCS_WAN_PORT, VCS_UP_PORT.`
`UINT2`	`u2PortNum`	port number of a Loop/WAN port.

Structure for OAM Configuration Block

The structure contains the configuration information for OAM control of the chipset ATLAS device. It effects all OAM connections associated with the chipset. In contrast, the structure sVCS_VC_OAM contains the configuration data on per-connection basis

Table 25: VORTEX chipset OAM Configuration Block (sVCS_OAM_CFG)

Field Type	Field Name	Field Description
UINT1	u1Ctl	Control Flag for OAM cell generation and configuration Bit 0: AutoRDI; 1 = automatically generate RDI upon termination of an AIS cell. 0 = otherwise. Bit 1: ForceCC; 1 = CC cells to be inserted regardless of user bandwidth. 0 = no CC cells when high user bandwidth. Bit 2: AISCopy; 1 = copies the Defect Location and Type fields of all received AIS cells to the VC Table. The associated SRAM should be populated in the VC table. 0 = no copying of the fields. The associated SRAM should not be populated in the VC table.
UINT2	u2AisCcCp	AIS and CC cell pacing limit.
UINT4	u4AisPhy	If bit x is set, AIS cells are generated automatically on all associated connections when a PHYx failure occurs.
UINT4	u4RdiPhy	If bit x is set, RDI cells are generated automatically on all associated connections when a PHYx failure occurs.
UINT1	u1DT[VCS_OAM_DEFECT_TYPES]	16 defect types used for non-automatic OAM cell generation.
UINT2	u2DL[8]	The 128 bits of defect location to be inserted into non-automatic OAM cell generation.

Field Type	Field Name	Field Description
UINT2	u2MaxIndex	Maximum VC Table index, reflecting ATLAS SRAM depth
UINT4	u4Aps	Automatic Protection Switching bits for controlling the automatic propagation of a segment AIS flow into an end-to-end AIS flow at a segment end point on per-PHY basis. If PHY x doesn't exist, the bit x should be 1, i.e. no end-to-end AIS generated.
UINT2	u2Bcp	ATLAS Egress OAM cell interface pacing: the number of cell intervals between the transfer of backward OAM cells. Not used for Ingress.
UINT2	u2Bto	The timeout limit before a cell at the head of the ATLAS Egress Backward Cell Interface FIFO is discarded. To prevent a malfunctioning PHY holding a Backward FIFO, consequently blocking all other cells that follow. Unit: cell periods. Not used for Ingress.

VC F4 to F5 OAM Processing Request

Passed via the vcsF4toF5Setup call. It contains a list of F5 (VCC) connections which are associated with one or two terminated F4 (VPC) connections for the F4 to F5 OAM processing.

Table 26: VORTEX chipset F4 to F5 OAM Request (sVCS_F4TOF5_ REQUEST)

Field Type	Field Name	Field Description
UINT2	F4EtoEConnId	specifies an End-to-End OAM VPC connection (VCI = 4)
UINT2	F4SegConnId	specifies an Segment OAM VPC connection (VCI = 3). If the field is set to be the same value as "F4EtoEConnId", it indicates the Segment OAM VPC connection does not participate in the processing.

Field Type	Field Name	Field Description
UINT2	u2NumVcc	specifies number of VCC in the list of sVCS_F4TOF5_VCC data buffer array pointed to by pVcc
sVCS_F4TOF5_VCC*	pVcc	Pointer to the first sVCS_F4TOF5_VCC data buffer in the list

Table 27: VORTEX chipset F4 to F5 VCC (sVCS_F4TOF5_ VCC)

Field Type	Field Name	Field Description
UINT2	u2ConnId	VCC connection ID
UINT1	u1F4ToF5Cfg	F4 to F5 processing Configuration for the VCC. Bit 0: F4toF5AIS, 1= F5 FM cells will be generated at F4 OAM termination; 0= F5 FM cells will not be generated at F4 OAM termination; only CC cells will be generated. Bit 1: SegmentFlow, 1 = An F5 Segment AIS cell will be generated while the F4 connection is in AIS alarm. 0 = an F5 end-to-end AIS will be generated instead. The bit should be set when the VCC connection is within a defined segment or not a VC end-point, i.e. the VCC extends beyond the end-point of the VPC. Note: the bit should not be set to 1 at Segment end-point

Connection Status and Information

Passed out via the `vcsConnStatus` call. It contains status of a VC connection which is maintained in the VC Record table.

Table 28: Connection Status (sVCS_CONN_STATUS)

Field Type	Field Name	Field Description
`eVCS_VC_STATE`	`state`	VC state; can be one of the following: VC_UNUSED, VC_STANDBY or VC_ACTIVE
`eVCS_VC_CLASS`	`eVcClass`	VC class; 00=Class 0, 01=Class 1, 10=Class 2 11=Class 3
`eVCS_VC_TYPE`	`eVcType`	VC connection type; 00=VCC Cells, 01=VCC Frame, 10=VPC Cell, 11=VPC Frame
`sVCS_VC_PORT_DES`	`VcIn`	specifies incoming cell ID, (VcPortID/vpi/vci)
`sVCS_VC_PORT_DES`	`VcOut`	specifies outgoing cell ID, (VcPortID/vpi/vci)
`UINT1`	`CardID`	specifies an active chipset card, through which the VC cells pass.
`UINT1`	`u1VcWeight`	VC queuing weight (linear encoding - 6 bits)
`sVCS_VC_THRSH`	`sVcThrsh`	Threshold values for the connection
`sVCS_VC_QOS`	`sQos`	current QOS parameters for the VC
`UINT1`	`mcFlag`	1=the connection belongs to a multicasting group 0=the connection doesn't belong to a multicasting group
`UINT4`	`mcId`	multicasting group ID if the above flag is 1

Passed out via the `vcsConnInfo` call. It reports current active VCs, and loads on each chipset (core card). It also lists available resources, such as number of empty VCs, logical channels and ports.

Table 29: Connection Information (sVCS_CONN_INFO)

Field Type	Field Name	Field Description
UINT2	activeVCs	number of active VCs
UINT2	inactiveVCs	number of inactive or disabled VCs
UINT2	availableVCs	number of unused VCs
UINT2	activeLoopPorts	number of active loop ports
UINT2	inactiveLoopPorts	number of inactive loop ports
UINT2	availableLoopPorts	number of unused loop ports
UINT2	activeWanPorts	number of active loop ports
UINT2	inactiveWanPorts	number of inactive loop ports
UINT2	availableWanPorts	number of unused loop ports
UINT2	activeChnls	number of Inband Control Channels
UINT2	availableChnls	number of unused Channels
UINT2	loadVCs[VCS_MAX_CARDS]	load or number of active VCs per card
UINT2	loadPorts[VCS_MAX_CARDS]	number of active ports per card

Remote Card Information

Passed out via the `vcsRemoteCardInfo` call. It reports the availability of remote Line/WAN cards at each HSS link and the HSS link status of a specified core card, as well as the total number of remote cards being added.

Table 30: Remote Card Information (sVCS_RCARD_INFO)

Field Type	Field Name	Field Description
UINT1	lineInfo [VCS_MAX_VORTEXS] [VTX_NUM_HSS_LINKS]	contains the remote Line card info at each HSS links Bit 0: 0 = Line card not added 1 = Line card connected or added Bit 1: 0 = inactive HSS link between the Line card and Core Card 1 = active HSS link between the Line card and Core Card
UINT1	wanInfo [VCS_DPX_HSS_LINKS]	contains the remote WAN card info at each HSS links Bit 0: 0 = WAN card not added 1 = WAN card connected or added Bit 1: 0 = inactive HSS link between the WAN card and Core Card 1 = active HSS link between the WAN card and Core Card
UINT2	u2RemoteCardCount	number of remote Line/WAN cards added to the core card

Statistic Counts

Passed out via the `vcsGetStatVcRxCnts` and `vcsGetStatVcNcCnts` calls. It reports the statistic counts as well as the counter configurations on per-VC basis.

Table 31: VC Statistic Counts (sVCS_VC_STAT_CNT)

Field Type	Field Name	Field Description
`UINT1`	`u1CntType]`	programmable count type: A logical 1 in any of bits indicates that counting on that particular stream is enabled. Bit 0 – CLP0 Cells with PTI=111 (F5) or VCI=7 to 15 (F4) Bit 1 – CLP1 Cells with PTI=111 (F5) or VCI=7 to 15 (F4) Bit 2 – CLP0 RM Cells Bit 3 – CLP1 RM Cells Bit 4 – CLP0 OAM Cells Bit 5 – CLP1 OAM Cells Bit 6 – CLP0 User Cells Bit 7 – CLP1 User Cells
`UINT4`	`u4Count`	per-VC Cell Counts

5.4 Structures in the Driver's Allocated Memory

These structures are defined and used by the chipset driver and some are part of the context memory allocated when the chipset driver is opened.

Global Driver Database (GDD)

The GDD is the top-level structure for the Module. It contains configuration data about the Module level code and pointers to card level configuration data structure (CDB) and Connection Admission Control (CAC) block.

Table 32: VORTEX chipset Global Driver Database (sVCS_GDD)

Field Type	Field Name	Field Description
UINT2	u2NumCards	Number of Chipset cards currently registered
sVCS_CDB	sCdb[VCS_MAX_CARDS]	array of Chipset Data Blocks (CDB) in context memory
sVCS_DEV_CTXT	sDevCtxt[VCS_MAX_CARDS][VCS_MAX_DEVS]	Contains device contexts for each underlying device driver
sVCS_CAC	sCAC	Connection Admission Control block.
sVCS_CHNL_TABLE	sChnlTable	control channel table
sVCS_VC_LIST *	pFreeVcList	Pointer to a pre-allocated memory buffer for storing free VC queue table and queue entry pool. This is used to minimize memory fragmentation.
UINT2	maxInitProfs	Maximum number of initialization profiles
sVCS_INIT_VECT *	psInitProfs	(array of) Pointers to different initialization profiles
VCS_IND_RX_CELL	indRxDataCell	Callback function pointer for cell Rx on user connections
VCS_IND_RX_FRM	indRxDataFrm	Callback function pointer for frame Rx on user connections

Field Type	Field Name	Field Description
UINT2	u2NumCards	Number of Chipset cards currently registered
VCS_IND_RX_CTRL_MSG	indRxCtrlMsg	Callback function pointer for message Rx on control channels
VCS_IND_RX_BOC	indRxBOC	Callback function pointer for BOC code Rx
VCS_IND_RX_OAM	indRxOAM	Callback function pointer for OAM (CC and Activation/Deactivation) cell Rx
VCS_IND_COS_STATUS	indCosStatus	Callback function pointer for Change of Status on OAM operation
VCS_IND_INTR	indCritical	Callback function pointer for critical interrupt events
VCS_IND_INTR	indError	Callback function pointer for non-critical interrupt events

Structure for a VC Queue Entry

Table 33: VORTEX chipset VC QUEUE ENTRY (sVCS_VC_INDEX)

Field Type	Field Name	Field Description
sVCS_VC_INDEX *	prev	a pointer to the previous element in the queue table
sVCS_VC_INDEX *	next	a pointer to the next element in the queue table
UINT2	Ici	Index of a VC in the VC Table

Structure for a VC Queue

Table 34: VORTEX chipset VC QUEUE TABLE (sVCS_VC_LIST)

Field Type	Field Name	Field Description
sVCS_VC_INDEX *	head	a pointer to the head of queue table
sVCS_VC_INDEX *	tail	a pointer to the tail of queue table
UINT2	numVCs	number of VCs associated with the queue (list)

Structures for Connection Admission Control

This is a high level, system-independent data structure, used to maintain the VC resources, and control the connection admission.

Table 35: VORTEX chipset Connection Admission Control (sVCS_CAC)

Field Type	Field Name	Field Description
VCS_SEM_ID	semVC	Semaphore object
UINT2	maxVCs	maximum VCs for the system
UINT2	numVCs	number of VCs being setup
sVCS_VC_RECORD *	psVcRecord	pointer to a VC Table with maxVCs number of VC Connection Records.
sVCS_VC_LIST	VcPerLoopPort[VCS_MAX_LOOP_PORTS]	A queue table for the VCs associated with a particular Loop port.
sVCS_VC_LIST	VcPerWANPort[VCS_MAX_WAN_PORTS]	A queue table for the VCs associated with a particular WAN port.
sVCS_VC_LIST	VcPerUpPort	A queue table for the VCs associated with the microprocessor port.
sVCS_MULTICAST_TABLE	sMcTable	A queue table for multicast group record
UINT4	u4TotalCellRate	Total bandwidth (in cell rate) has been used.
UINT4	u4UpCellRate	Total Cell rate in Upstream direction has been used
UINT4	u4DownCellRate	Total Cell rate in Downstream direction has been used.
sVCS_PORT_STATUS	sLoopPortState[VCS_MAX_LOOP_PORTS]	(array of) loop port status information.

Field Type	Field Name	Field Description
sVCS_PORT_STATUS	sWanPortState[VCS_MAX_WAN_PORTS]	(array of) WAN port status information
sVCS_PORT_STATUS	sMpPortState	Microprocessor port status information
sVCS_SHPR_VECTOR	psShaper[4]	Control Vectors for the four APEX shapers

Structure for a VC Table Record

Table 36: VORTEX chipset VC TABLE (sVCS_VC_RECORD)

Field Type	Field Name	Field Description
eVCS_VC_STATE	state	VC state; can be one of the following: VC_UNUSED, VC_STANDBY or VC_ACTIVE
eVCS_VC_CLASS	eVcClass	VC class; 00 = Class 0, 01 = Class 1, 10 = Class 2 11= Class 3.
eVCS_VC_TYPE	eVcType	VC connection type; 00 = VCC Cells, 01 = VCC Frame, 10 = VPC Cell, 11=VPC Frame
sVCS_VC_VECTOR	VcIn	specify incoming cell ID, (VcPortID/vpi/vci)
sVCS_VC_VECTOR	VcOut	specify outgoing cell ID, (VcPortID/vpi/vci)
UINT1	CardID	specify active chipset card, through which the VC cells pass.
sVCS_VC_QOS	sQos	specifies QOS parameters for the VC, including peak cell rate, VC type (CBR, VBR etc)
UINT1	u1VcWeight	VC queuing weight (linear encoding – 6 bits)
sVCS_VC_THRSH	sVcThrsh	Contains per-VC Congestion Control Thresholds

Field Type	Field Name	Field Description
sVCS_VC_OAM *	psVcOAM	Contains OAM configuration for the VC. If Null, OAM is not enabled.
sVCS_F4TOF5_CB *	psF4toF5OAM	Pointer to F4 to F5 OAM processing control block. If NULL, the F4 to F5 processing is not enabled, or the VC is not a member of any F4 to F5 processing list.
UINT1	u1F4ToF5Cfg	F4 to F5 processing Configuration for the VCC. Bit 0: F4toF5AIS, 1= F5 FM cells will be generated at F4 OAM termination; 0= F5 FM cells will not be generated at F4 OAM termination; only CC cells will be generated. Bit 1: SegmentFlow, 1 = An F5 Segment AIS cell will be generated while the F4 connection is in AIS alarm. 0 = an F5 end-to-end AIS will be generated instead. The bit should be set when the VCC connection is within a defined segment or not a VC end-point, i.e. the VCC extends beyond the end-point of the VPC.
sVCS_MULTICAST_RECORD *	psMulticast	A pointer to a multicast record which the VC belongs. If null, it means the ICI doesn't belong to any Multicast group.

Structure for Port Status

Table 37: Loop/WAN Port Status Structure (sVCS_PORT_STATUS)

Field Type	Field Name	Field Description
UINT1	state	Indicate the current port state. Bit 0-1: 00 = not available, (Line/WAN card not present) 01 = inactive, (remote card available, but associated HSS links are inactive) 11 = active (remote card available and an associated HSS link is active) Bit 2-3: specifies an active core card ID Bit 4: 1= the port is configured (in APEX), 0 = Otherwise Bit 5: 1= the port is enabled (in APEX), 0 = Otherwise Bit 6: 1= its associated HSS link is in Loopback mode, 0 = otherwise. Not used for uP port. Bit 7:For Loop port: it indicates whether this port is reserved for control channel to the remote line card. 1= yes. For WAN port: indicates an Active DUPLEX HSS link to a WAN card. 1 = Link 1, 0 = Link 0. For uP port, not used.
UINT4	maxInCellRate	maximum cell rate allowed in Inward (towards chipset) direction.

Field Type	Field Name	Field Description
UINT4	minOutCellRate	minimum cell rate guaranteed in Outward(away from chipset) direction.
UINT4	InCellRate	Bandwidth (cell rate) in Inward direction has been used.
UINT4	OutCellRate	Bandwidth (cell rate) in Outward direction has been used.
UINT1	u1Weight	Per-port polling weight used in APEX queue engine. For loop ports: its value ranges from 0 to 7. For WAN ports: its value range from 0 to 3.
UINT1	u1ThrshForceFlag	Flag indicating whether the port thresholds was specified by the user when the port was set up.
UINT2	minPortThrsh	a minimum guaranteed port threshold level
sVCS_CLASS_SCHEDULER	sClassSchdl	Contains the class scheduling parameters for the port.
sVCS_PORT_THRSH	sPortClassThrsh	Contains the per-port and per-class congestion control thresholds.

Structure for Loopback Control Block

Table 38: VORTEX chipset Loopback Control Block (sVCS_LPBK_CB)

Field Type	Field Name	Field Description

Field Type	Field Name	Field Description
VCS_DEV_HANDLE	DevHandle	Device handle to the loopback device, whose HSS link is in loopback mode. Either VORTEX or DUPLEX handle. If NULL, the chipset is not in Loopback mode.
sVCS_VC_PORT_DES	sTestingVcPort	testing point for testing cell insert/extract
UINT1	HssLinkId	Specifies the HSS Link in loopback mode, bit 7: 0 = VORTEX, 1 = DUPLEX
UINT2	ForwardIci	Index of the forward VC in the VC table
UINT2	BackwardIci	index of the forward VC in the VC table
UINT1	TestPortState	It stores the testing port state before the Loopback mode being setup
UINT1	LpbkPortState	It stores the loopback port state before the Loopback mode being setup
sVCS_LPBK_DATA *	psLpbkData	control block for received loopback data from microprocessor port

Table 39: VORTEX chipset Loopback Data Block (sVCS_LPBK_DATA)

Field Type	Field Name	Field Description
UINT1	maxSz	maximum size of Rx loopback data buffer
UINT1	actualSz	actual size of loopback data received
UINT1 *	pRxData	pointer to the loopback data buffer of ‘maxSz’ bytes

Structure for multicast support

Table 40: VORTEX chipset Multicast Record (sVCS_MULTICAST_RECORD)

Field Type	Field Name	Field Description
VOID *	prev	a pointer to the previous element in the queue table
VOID *	next	a pointer to the next element in the queue table
UINT4	InIci	specifies incoming VC connection ICI
sVCS_VC_LIST	sOutICIList	Contains a list of output connection ICIs

Table 41: VORTEX chipset Multicast Record Table (sVCS_MULTICAST_TABLE)

Field Type	Field Name	Field Description
sVCS_MULTICAST_RECORD *	head	a pointer to the head of queue table
sVCS_MULTICAST_RECORD *	tail	a pointer to the tail of queue table
VCS_SEM_ID	semMc	Semaphore object for the multicast table
UINT2	numGroups	Number of multicast groups in the table

Structure for OAM and F4 to F5 Processing (per VC)

The structure sVCS_VC_OAM contains the OAM configuration and F4 to F5 OAM processing parameters on per-connection basis.

Table 42: VC OAM Structure (sVCS_VC_OAM)

Field Type	Field Name	Field Description
sVCS_VC_OAM_FM	sFMcfg	Contains the FM configuration for the VC
sVCS_VC_OAM_PM	sPMcfg	Contains the PM configuration for the VC
UINT2	OAMBackPath	A connection ID in backward direction, used for OAM backward reporting cells.

Table 43: F4 to F5 OAM Processing Control Block (sVCS_F4TOF5_CB)

Field Type	Field Name	Field Description
UINT1	u1TermType	VPC termination type; could be one of eVCS_VP_ETE_SEG; eVCS_VP_ETE; eVCS_VP_SEG.
UINT2	u2SegConnId	Segment OAM Connection ID (VCI = 3). if 0, not provisioned
UINT2	u2EtEConnId	End-to-End OAM Connection ID (VCI =4). if 0, not provisioned
sVCS_VC_LIST	sVccList	Contains a list of VCC connection Ids which are associated with the VPC

Structure for Inband Control Channel

Table 44: VORTEX chipset Channel Record (sVCS_CHNL_RECORD)

Field Type	Field Name	Field Description
UINT1	state	specifies the state of the record, UNUSED or USED (active)
UINT1	cardId	specifies which HSS line, and which VORTEX or DUPLEX device is connected to the Line or WAN card bit 0-2: HSS link number (0 to 7) bit 3-6: device number bit7 : 0=VORTEX, 1 = DUPLEX
sVCS_VPI_VCI	sVpciTx	VPI and VCI for the Tx message channel
sVCS_VPI_VCI	sVpciRx	VPI and VCI for the Rx message channel
UINT1 *	pRxBuff	Pointer to a Data buffer for the Rx message
UINT4	maxMsgSz	maximum message size
UINT4	dataLength	Data length in the Rx buffer
UINT1	flagFCQ	VC type: Frame or Cell. 1 = Frame, 0 = Cells
UINT2	u2InConnID	Connection ID used for incoming control message from remote Line card to the APEX microprocessor port. The value shall be between (VCS_MAX_VC-2*maxChnls-1) and (VCS_MAX_VC-1).
UINT2	u2OutConnID	Connection ID used for outgoing control message from the APEX microprocessor port to remote Line card. The value shall be between (VCS_MAX_VC-2*maxChnls-1) and (VCS_MAX_VC-1).

Table 45: VORTEX chipset Channel Record Table (sVCS_CHNL_TABLE)

Field Type	Field Name	Field Description
VCS_SEM_ID	semChnl	Semaphore object
UINT2	maxChnls	Maximum number of control channels.
UINT2	numChnls	Number of active control channels
sVCS_CHNL_RECORD *	psChnlRecord	(array of) the control channel records

Chipset Data Block (CDB)

The CDB is the top-level structure for each Chipset card. It contains card level configuration data and device handles to each chipset device on the card.

Table 46: VORTEX chipset Data Block (sVCS_CDB)

Field Type	Field Name	Field Description
UINT4	u4Valid	Indicates whether the CDB is used or not.
eVCS_STATE	eState	indicates one of the chipset state: VCS_START, VCS_PRESENT, VCS_STANDBY, VCS_ACTIVE
VOID *	pSysInfo	Pointer to system specific card information. For example, in PCI bus environment, the bus number, IRQ assignment etc.
VCS_USR_CTXT	usrCtxt	Pointer to user's context for this card. The user passes this pointer while adding the card. The chipset driver passes this context when it invokes the indication callbacks.
sVCS_CIB	sVcsCib	Contains the chipset information, such as base address, memory map and number of VORTEX and DUPLEX chips.
VCS_DEV_HANDLE	ApxHandle	Device handle to the APEX device on the Core card
VCS_DEV_HANDLE	AtlasHandle	Device handle to the ATLAS device on the Core card
VCS_DEV_HANDLE	DpxHandle	Device handle to the DUPLEX device on the Core card
VCS_DEV_HANDLE	pVtxHandle[VCS_MAX_VORTEXS]	(array of) Device handles to the VORTEX devices on the core card
UINT2	u2RemoteCardCount	number of remote line/WAN cards which are actively connected to the core card
sVCS_LPBK_CB	sLpbkCtrl	Contains Loopback Control Block

Chipset Information Vector

This structure contains all the information needed by the chipset driver to access each individual device of the VORTEX chipset.

Table 47: VORTEX chipset Information Block (sVCS_CIB)

Field Type	Field Name	Field Description
UINT4	u4BoardBaseAddr	Base address of the chipset card, i.e. the first accessible address of the card. Stored here for bookkeeping purpose.
UINT4	u4ApexBaseAddr	Base address of the APEX chip on the chipset card
UINT4	u4AtlsBaseAddr	Base address of the ATLAS chip on the chipset card
UINT4	u4DpxBaseAddr	Base address of the DUPLEX chip on the chipset card
UINT4	u4VtxBaseAddr[VCS_MAX_VORTEX]	(array of) Base address of the VORTEX chips on the chipset card

Event Counts

The structure contains the event counts accumulated by each device driver.

Table 48: VORTEX chipset Driver Statistic Counts (sVCS_STAT_CNT)

Field Type	Field Name	Field Description
sDPX_STAT_COUNTS	sDpxCounts	event counts maintained by the DUPLEX device driver
sVTX_STAT_COUNTS	sVtxCounts	event counts maintained by the VORTEX device driver
sAPX_STAT_COUNTS	sApxCounts	event counts maintained by the APEX device driver
sATLAS_STAT_COUNTS	sAtlasCounts	event counts maintained by the ATLAS device driver

6 VORTEX CHIPSET HARDWARE INTERFACE

6.1 Chipset I/O

The VORTEX chipset driver interfaces with the chipset hardware via its underlying device drivers. Each device driver uses the following low-level system specific macro for accessing the device registers.

sysVcsRawRead32

This low-level system specific macro is used to read the 32 bit long contents of a specific address location. This macro should be defined by the user to reflect their system's addressing logic. This macro is used by APEX device driver to access its 32 bit register space.

Format `#define sysVcsRawRead32(addr)`

Inputs `addr` : address location to be read

Outputs `None`

Return Codes value read from the address location

sysVcsRawWrite32

Low-level system specific macro is used to write the 32 bit long contents to a specific address location. This macro should be defined by the user to reflect their system's addressing logic. This macro is used by APEX device driver to access its 32 bit register space.

Format `#define sysVcsRawWrite32(addr, val)`

Inputs `addr` : address location to write

`val` : 32 bit value to be written

Outputs None

Return Codes None

sysVcsRawRead16

Low-level system specific macro is used to read the 16 bit long contents of a specific address location. This macro should be defined by the user to reflect their system's addressing logic. This macro is used by ATLAS device driver to access its 16 bit register space.

Format `#define sysVcsRawRead16(addr)`

Inputs `addr` : address location to be read

Outputs `None`

Return Codes value read from the address location

sysVcsRawWrite16

Low-level system specific macro is used to write the 16 bit long contents to a specific address location. This macro should be defined by the user to reflect their system's addressing logic. This macro is used by ATLAS device driver to access its 16 bit register space.

Format `#define sysVcsRawWrite16(addr, val)`

Inputs `addr` : address location to write

`val` : 16 bit value to be written

Outputs None

Return Codes None

sysVcsRawRead8

Low-level system specific macro is used to read the 8 bit long contents of a specific address location. This macro should be defined by the user to reflect their system's addressing logic. This macro is used by DUPLEX and VORTEX device drivers to access their 8 bit register space.

Format #define sysVcsRawRead8(addr)

Inputs addr : address location to be read

Outputs None

Return Codes value read from the address location

sysVcsRawWrite8

Low-level system specific macro is used to write the 8 bit long contents to a specific address location. This macro should be defined by the user to reflect their system's addressing logic. This macro is used by DUPLEX and VORTEX device drivers to access their 8 bit register space.

Format #define sysVcsRawWrite8(addr, val)

Inputs addr : address location to write

val : 8 bit value to be written

Outputs None

Return Codes None

6.2 Chipset Detection

sysVcsCardDetect

This function is used to detect the chipset core card in the system, and retrieve system specific information about the card and VORTEX chipset devices. The information includes the base address of the card, and memory map for each chipset device, as well as interrupt IRQ number in the case of PCI platform. This function is called within the vcsAdd API function.

Note: the device detection functions (e.g. `sysApexDeviceDetect, sysVtxDevice Detect, sysDpxDeviceDetect, sysAtlasDevice Detect`) in each device driver should be modified to reflect the memory mapping of the card.

Prototype `INT4 sysVcsCardDetect(VCS_USR_CTXT usrCtxt, void **ppSysInfo, sVCS_CIB *pVcsCib )`

Inputs `usrCtxt` : user handle for the core card being added

Outputs `ppSysInfo` : user-maintained system information (e.g., PCI slot, board base address, irq etc.); this pointer is stored by the chipset driver

`pVcsCib` : contains base address of each chipset device on the card, as well as number of VORTEX and DUPLEX device on the card.

Return Codes `VCS_SUCCESS`

`VCS_FAILURE`

6.3 Interrupt Servicing

Interrupt servicing of the VORTEX chipset devices is provided by each underlying device driver. However, the chipset driver specifies which device interrupts should be masked or delivered, based on the chipset initialization vector passed in from Users.

The chipset driver provides a high level service task called "`vcsRxTask`", to process the cells received from the underlying DUPLEX microprocessor port. The task waits for messages, sent from the DPR tasks of DUPLEX device drivers, to arrive at its associate message queue. Once a message has been received, the task extracts cells/frames out of the device buffers and reports the cells/frames to the application via indication callback functions. This task can provide an inband communication channels between the core card and WAN cards.

The `apexSarRxTask` task, provided by APEX device driver, are used to support any data communication over the VC connections between the microprocessor port and a Loop/WAN port, as well as the control channel messaging between the core card and remote line cards.

The `atlasRxCellTask` task, provided by ATLAS device driver, is implemented to support the Microprocessor OAM end-point processing. The supported OAM cells include Loopback, and Activation/Deactivation.

Please refer to Section 10.12 and Figure 16 for a detailed description.

7 RTOS INTERFACE

The VORTEX chipset driver requires the use of some RTOS resources. In this section, 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 writing a function for each item listed. Care should be taken when matching parameters and return values.

7.1 Memory Allocation / De-Allocation

sysVcsMemAlloc

Allocates specified number of bytes of memory.

Format `#define sysVcsMemAlloc(numBytes)`

Prototype `UINT1 *sysVcsMemAlloc(UINT4 numBytes)`

Inputs `numBytes` : number of bytes to be allocated

Outputs None

Returns Pointer to first byte of allocated memory

NULL pointer (memory allocation failed)

sysVcsMemFree

Frees memory allocated using `sysVCSMemAlloc`.

Format `#define sysVcsMemFree(pfirstByte)`

Prototype `void sysVcsMemFree(UINT1 *pfirstByte)`

Inputs `pfirstByte` : pointer to memory region being de-allocated

Outputs None

Returns None

7.2 Timers

sysVcsDelayTask

Suspends execution of a chipset driver task for a specified number of milliseconds.

Format `#define sysVcsDelayTask(time)`

Prototype `void sysVcsDelayTask(UINT4 time)`

Inputs `time` : sleep time in milliseconds

Outputs None

Returns None

7.3 Semaphores

sysmVcsSemCreate

Creates a binary semaphore.

Prototype `VCS_SEM_ID sysmVcsSemCreate(VOID)`

Inputs None

Outputs None

Return Codes pointer to semaphore object OR nul

sysmVcsSemDelete

Deletes a binary semaphore object.

Prototype `VOID sysmVcsSemDelete(VCS_SEM_ID semId)`

Inputs `semId` : semaphore identifier

Outputs None

Return Codes None

sysmVcsSemTake

Acquires a binary semaphore.

Prototype `INT4 sysmVcsSemTake(VCS_SEM_ID semId)`

Inputs `semId` : semaphore identifier

Outputs None

Return Codes 0 : success, -1 : failure

sysVcsSemGive

Relinquishes a semaphore.

Prototype `INT4 sysmVcsSemGive(VCS_SEM_ID semId)`

Inputs `semId` : semaphore identifier

Outputs None

Return Codes 0 : success, -1 : failure

7.4 System-specific Inband Control Channel (ICC) module functions

sysVcsIccInstall

Creates the `vcsIccRx` task, which handles the rx for inband control channel messages from the microprocessor port of the DUPLEX. It also creates the message `VcsRxMsgQ` to allow the application task to communicate with the task.

Prototype `INT4 sysVcsIccInstall(VOID)`

Inputs None

Outputs None

Return Codes 0 : success, -1 : failure

sysVcsIccRemove

This routine deletes the `vcsIccRx` tasks and the corresponding message queue `VcsRxMsgQ`.

Prototype `INT4 sysVcsIccRemove(VOID)`

Inputs None

Outputs None

Return Codes 0 : success, -1 : failure

sysVcsIccRxTaskFn

This routine is spawned as a separate task within the RTOS. It retrieves interrupt status information saved for it by the DPR tasks of DUPLEX device driver. It invokes the `vcsRxTaskFn` routine for each device handle received in the message.

Prototype `VOID sysVcsIccRxTask(VOID)`

Inputs None

Outputs None

Return Codes None

Pseudocode

```
begin
do wait for interrupt status messages sent by DPR task of DUPLEX
driver
dequeue a message when it arrives
for each device handle in the message
        invoke vcsRxTaskFn
loop forever
end.
```

8 APPLICATION PROGRAMMING INTERFACE

This section provides a detailed description of each function that is a member of the VORTEX chipset driver Application Programming Interface (API).

8.1 Module Initialization

vcsModuleInit

Performs module level initialization of the chipset driver. This involves allocating all of the memory needed by the chipset driver and initializing the Global Driver Database (GDD) with the passed Module Initialization Vector (MIV). It also opens each device driver module for the chipset devices on the card.

The whole VC table (with VCS_MAX_VC number of connection entries or IDs) is divided into two parts: connection IDs from 0 to (psMiv->maxVCs) are used for USER data connections, while the top 2*(psMiv->maxChnls) connections are reserved for control channel useges. Hence, there exists the following constraint between maxVCs and maxChnls in the MIV parameters: (maxVCs + 2 * maxChnls) =< VCS_MAX_VC

Valid States START

Side Effects Changes MODULE state to READY

Prototype `INT4 vcsModuleInitn(sVCS_MIV *psMiv)`

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

Outputs `None`

Returns
```
VCS_SUCCESS
VCS_ERR_MODULE_ALREADY_INIT
VCS_ERR_MEM_ALLOC
VCS_ERR_MIV (invalid Module Init Vector)
 VCS_ERR_SEMAPHORE
```

vcsModuleShutdown

Performs module level shutdown of the chipset driver. This involves deleting all chipset devices being controlled by the chipset driver (by calling `vcsDelete` for each chipset card) and de-allocating the VC, Control Channel Table, and GDD.

Valid States ALL STATES

Side Effects Changes MODULE state to VCS_START

Prototype `VOID vcsModuleShutdown(VOID)`

Inputs None

Outputs None

Returns `None`

8.2 Initialization Profile Management

vcsSetInitProfile

Creates an initialization profile that is stored by the chipset driver. A chipset can now be initialized by simply passing the initialization profile number. This function should only be called after "`vcsModuleInit`".

Valid States READY

Side Effects None

Prototype
```
INT4 vcsSetInitProfile(sVCS_INIT_VECTOR
     *pProfile, UINT2 *pProfileNum)
```

Inputs

`pProfile` : (pointer to) initialization profile being added

`pProfileNum` : (pointer to) profile number to be assigned by the chipset driver

Outputs `pProfileNum` : profile number assigned by the chipset driver

Returns
```
VCS_SUCCESS
VCS_ERR_MODULE_NOT_INIT
VCS_ERR_INVALID_INIT_VECTOR
VCS_ERR_PROFILE_FULL
```

Pseudocode
```
begin
check passed profile
reserve unassigned profile number
update corresponding profile
end
```

vcsGetInitProfile

Gets the contents of an initialization profile given its profile number. The user should allocate enough memory to receive the initialization vector.

Valid States READY

Side Effects None

Prototype
```
INT4 vcsGetInitProfile(UINT2 profileNum,
sVCS_INIT_VECTOR *pProfile)
```

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

Outputs `pProfile` : contents of the corresponding profile

Returns
```
VCS_SUCCESS
VCS_ERR_MODULE_NOT_INIT
VCS_ERR_INVALID_PROFILE
```

Pseudocode
```
begin
make sure profile exists
make copy of the profile
end
```

vcsClrInitProfile

Clears an initialization profile given its profile number.

Valid States READY

Side Effects None

Prototype `INT4 vcsClrInitProfile(UINT2 profileNum)`

Inputs `profileNum` : initialization profile number

Outputs None

Returns
```
VCS_SUCCESS
VCS_ERR_MODULE_NOT_INIT
VCS_ERR_INVALID_PROFILE
```

Pseudocode
```
Begin
make sure profile exists
release profile number
End
```

8.3 Chipset Add and Delete

vcsAdd

Verifies the presence of a new Chipset card in the hardware, then return a handle back to the user. The Chipset handle should be used to identify the Chipset card on which the operation is to be performed.

Valid States READY

Side Effects Changes the CHIPSET state to VCS_PRESENT

Prototype
```
INT4 vcsAdd(VCS_USR_CTXT usrCtxt, VCS *pVcs)
```

Inputs `usrCtxt` : User maintained context information for the chipset card being added.

Outputs `pVcs` : (pointer to) Chipset Handle

Returns
```
VCS_SUCCESS
VCS_ERR_MODULE_NOT_INIT
VCS_ERR_CARDS_FULL
VCS_ERR_CHIPSET_NOT_DETECTED
VCS_ERR_CHIPSET_ALREADY_ADDED
```

Pseudocode
```
Begin
make sure Chipset present
reserve a Chipset Handle or CDB
save user context in the CDB
add each device to its device driver module.
Store each device handle in the CDB.
output Chipset handle
End
```

vcsDelete

This function is used to remove a specified Chipset card from the list of chipset cards being controlled by the chipset driver. Deleting a Chipset involves clearing the CDB for that Chipset and releasing its associated Chipset handle.

Valid States VCS_PRESENT

Side Effects The chipset state change to READY

Prototype
```
INT4 vcsDelete(VCS vcs)
```

Inputs `vcs` : Chipset Handle (from `vcsAdd`)

Outputs None

Returns
```
VCS_SUCCESS
VCS_ERR_INVALID_HANDLE
VCS_ERR_INVALID_STATE
```

Pseudocode
```
Begin
validates the handle
delete the chipset devices from device drivers
releases Chipset handle
End
```

8.4 Chipset Initialization and Reset

vcsInit

Initializes the chipset based on an initialization vector passed by the user. Each chipset device is configured according to the contents of the initialization vector. Alternatively, the user can also use an initialization vector profile number. In this case, the device is now initialized as per the profile contents (stored in GDD).

Valid States VCS_PRESENT

Side Effects Changes CHIPSET state to VCS_STANDBY

Prototype
```
INT4 vcsInit(VCS vcs, sVCS_INIT_VECTOR
*psInitVect, UINT2 profileNum)
```

Inputs

`vcs` : chipset Handle (from `vcsAdd`)

`psInitVect` : initialization vector that is used by the chipset driver to configure the chipset devices. The pointer should be set to NULL if an initialization vector profile is being used instead.

`profileNum` : profile number to be used to get the initialization vector from the GDD. This variable should be set to 0xffffffff if an initialization vector is being passed directly instead.

Outputs None

Returns

`VCS_SUCCESS`
`VCS_ERR_INVALID_HANDLE` (invalid chipset handle)
`VCS_ERR_INVALID_STATE` (chipset is not in a valid state)
`VCS_ERR_INVALID_INIT_VECTOR` (invalid initialization vector)
`VCS_ERR_INVALID_PROFILE_NUM` (invalid profile number)
`VCS_ERR_PROFILE_VECTOR_BOTH_VALID` (both profile and vector inputs were valid – not allowed)

Pseudocode
```
Begin
if using profile get a InitVect from profile
validate the InitVect
reset Chipset devices
configure Chipset and initialize all devices
End
```

vcsReset

Applies a software reset to each VORTEX chipset device. Also resets all the CDB contents (except for the user context). This function is typically called before re-initializing the Chipset.

Note that the VC connection table, maintained by the chipset driver module, is not cleared by this function. This allows a quick restore of the connections after the Reset by calling `vcsRebuildVCs`. However, if the associated VC connections are to be cleared, one need exclusively call `vcsClearVCs` to tear down the connections before calling the API `vcsReset`.

Valid States VCS_ACTIVE, VCS_STANDYBY, VCS_PRESENT

Side Effects Changes CHIPSET state to VCS_PRESENT

Prototype `INT4 vcsReset(VCS vcs)`

Inputs `vcs` : chipset handle (from `vcsAdd`)

Outputs None

Returns `VCS_SUCCESS`
`VCS_ERR_INVALID_HANDLE` (invalid chipset handle)

Pseudocode

```
Begin
reset each Chipset device
clear initialization part of the CDB
End
```

8.5 Chipset Activate and De-Activate

vcsActivate

Activates each chipset device by preparing it for normal operation. Activation involves installing and enabling device interrupts, enabling the APEX queue engine's external interfaces. However, the LVDS links to WAN/Line cards are not activated. These links are activated only when calling `vcsAddCard` to add the Line/WAN cards to the system.

Valid States VCS_STANDBY

Side Effects Change the CHIPSET state to VCS_ACTIVE

Prototype `INT4 vcsActivate(VCS vcs)`

Inputs `vcs` : Chipset Handle (from `vcsAdd`)

Outputs None

Returns
`VCS_SUCCESS`
`VCS_ERR_INVALID_HANDLE` (invalid chipset handle)
`VCS_ERR_INVALID_STATE` (chipset is not in a valid state)

Pseudocode
```
Begin
activate each chipset device
End
```

vcsDeActivate

De-activates the Chipset from normal operation. Interrupts are masked and the Chipset is put into a quiet state by disabling APEX queue engine.

Valid States VCS_ACTIVE

Side Effects Changes the CHIPSET state to VCS_STANDBY

Prototype `INT4 vcsDeActivate(VCS vcs)`

Inputs `vcs` : chipset Handle (from `vcsAdd`)

Outputs None

Returns
`VCS_SUCCESS`
`VCS_ERR_INVALID_HANDLE` (invalid chipset handle)
`VCS_ERR_INVALID_STATE` (chipset is not in a valid state)

Pseudocode
```
Begin
deactivate devices by calling device driver API
End
```

8.6 Chipset Device Read and Write

vcsReadReg

This function can be used to read a register in a specific VORTEX chipset device by providing the register number and device ID.

Valid States ALL CHIPSET STATES

Side Effects May affect registers that change after a read operation

Prototype
```
INT4 vcsReadReg(VCS vcs, sVCS_DEV_ID sDevId,
UINT2 u2RegOff, UINT4 * pu4Val)
```

Inputs

`vcs`	: chipset Handle (from `vcsAdd`) to a specified card
`sDevId`	: specifies a chipset device
`u2RegOff`	: register register offset from its device base address

Outputs `pu4Val` : contains value read from the register

Returns

`VCS_SUCCESS`	
`VCS_ERR_INVALID_HANDLE`	(invalid chipset handle)
`VCS_ERR_INVALID_DEV_ID`	(invalid device ID)

Pseudocode
```
Begin
get a device handle from CDB
call device driver API to read the register value
End
```

vcsWriteReg

This function can be used to write to a register in a specific VORTEX chipset device by providing the register number and device ID

Valid States ALL CHIPSET STATES

Side Effects May change the configuration of the chipset device

Prototype
```
INT4 vcsWriteReg(VCS vcs, sVCS_DEV_ID sDevId,
UINT2 u2RegOff, UINT4 u4Val)
```

Inputs

`vcs`	: chipset Handle (from `vcsAdd`)
`sDevId`	: specifies a chipset device
`u2RegOff`	: register offset from its device base address
`u4Val`	: value to be written

Outputs None

Returns

`VCS_SUCCESS`	
`VCS_ERR_INVALID_HANDLE`	(invalid chipset handle)
`VCS_ERR_INVALID_DEV_ID`	(invalid device ID)

Pseudocode

```
Begin
get a device handle from CDB
call device driver API to write the register
End
```

8.7 Chipset Diagnostics and Loopback Self-test

vcsRegisterTest

Verifies the correctness of the microprocessor's access to each chipset device by writing to and reading back values of its registers.

Valid States VCS_PRESENT

Side Effects The chip is reset and kept in the VCS_PRESENT state

Prototype `INT4 vcsRegisterTest(VCS vcs, sVCS_DEV_ID sDevId)`

Inputs

`vcs`	: chipset handle
`sDevId`	: specifies a chipset device.

Outputs None

Return Codes

`VCS_SUCCESS`	
`VCS_ERR_INVALID_HANDLE`	(invalid chipset handle)
`VCS_ERR_INVALID_STATE`	(chipset is not in a valid state)
`VCS_ERR_INVALID_DEV_ID`	(invalid device ID)
`VCS_FAILURE`	(test failed)

vcsMemTest

Verifies the correctness of the microprocessor's access to the external memory associated with the APEX and ATLAS chip.

Valid States VCS_PRESENT

Side Effects the chipset is reset after the test.

Prototype `INT4 vcsMemTest(VCS vcs, sVCS_DEV_ID sDevId)`

Input

`vcs`	: chipset handle
`sDevId`	: specifies a chipset device.

Outputs None

Return Codes

`VCS_SUCCESS`	
`VCS_ERR_INVALID_HANDLE`	(invalid chipset handle)
`VCS_ERR_INVALID_STATE`	(chipset is not in a valid state)
`VCS_ERR_INVALID_DEV_ID`	(invalid device ID)
`VCS_FAILURE`	(test failed)

vcsLpbkSetup

Used for the integrity check of a chipset system. It sets one of HSS link port of VORTEX or DUPLEX on the core card into a Diagnostic Loopback mode, and setup Loopback connections within APEX and ATLAS devices. The loopback point has to be specified at one of HSS links or Loop/WAN ports, while the cell insert/extract point can be one of LOOP/WAN ports or the Microprocessor port (of APEX).

If the specified HSS link port is already in use by some active VC connections, the Loopback request will be rejected.

If the chipset is already in a LOOPBACK state, the current Loopback link port will be reset to normal, before the new HSS link port is configured into a diagnostic Loopback mode. The Loopback VC connections are also re-configured to reflect the new loopback path. In other words, only one loopback path can be setup at any time.

Valid States VCS_ACTIVE

Side Effects Changes a HSS link port state into a LOOPBACK mode. This prohibits any normal VC connections over the link port.

Prototype

```
INT4 vcsLpbkSetup(VCS vcs, sVCS_VC_PORT_DES
sTestingPort, UINT1 u1LpbkLink, sVCS_VC_QOS sQos,
UINT1 u1FrameFlag)
```

Inputs

`vcs` : chipset handle

`sTestingPort` : specifies a testing port, where the testing cell are inserted and loopbacked cells are extracted. The VPI/VCI values of testing cells are also specified.

`u1LpbkLink` : specifies a HSS link to be set up into a diagnostic loopback mode.
- bit 0-2: HSS link number (0 to 7)
- bit 3-6: VORTEX device number. Unused if DUPLEX.
- bit 7 : 0=VORTEX, 1 = DUPLEX

`sQos` : contains QOS parameters for testing cell VC.

`u1FrameFlag` : flag for testing cell type. 1= Frame, 0 = Cell

Outputs None

Return Codes

VCS_SUCCESS	
VCS_ERR_INVALID_HANDLE	(invalid chipset handle)
VCS_ERR_INVALID_STATE	(chipset is not in a valid state)
VCS_ERR_INVALID_PORT_DES	
VCS_ERR_LPBKPORT_IN_USE	
VCS_ERR_INVALID_HSS_LINK_ID	
VCS_ERR_LPBK_INVALID_PARAM	

vcsLpbkClear

Configures the loopback port to a normal mode. It also clears the associated loopback connections.

Valid States VCS_ACTIVE

Side Effects The loopback link port is back to normal, and normal VC connections over the link port can be resumed.

Prototype

```
INT4 vcsLpbkClear(VCS vcs)
```

Inputs `vcs` : chipset handle

Outputs None

Return Codes

VCS_SUCCESS	
VCS_ERR_INVALID_HANDLE	(invalid chipset handle)
VCS_ERR_INVALID_STATE	(chipset is not in a valid state)
VCS_ERR_NO_LPBK	
VCS_ERR_INVALID_HSS_LINK_ID	

vcsMpLpbkTest

When a LOOPBACK state has been setup for the chipset, and its testing port being set at the Microprocessor port, this function can be called to send out a buffer of data through the Microprocessor port, and check if the same testing data is being looped back.

Valid States VCS_ACTIVE

Side Effects None

Prototype

```
INT4 vcsMpLpbkTest (VCS vcs, UINT1 *pData, UINT4
u4Length, UINT4 u4WaitTime)
```

Inputs

`vcs`	: chipset handle
`pData`	: buffer pointer to the data to be sent.
`u4Length`	: the length of testing data in bytes
`u4WaitTime`	: waiting time for receiving the loopback data in milliseconds.

Outputs None

Return Codes

`VCS_SUCCESS`
`VCS_ERR_INVALID_HANDLE` (invalid chipset handle)
`VCS_ERR_INVALID_STATE` (chipset is not in a valid state)
`VCS_ERR_NO_LPBK`
`VCS_ERR_NON_MP_PORT`
`VCS_ERR_TIMEOUT` (timeout for receiving any loopback testing cells)
`VCS_ERR_CELL_MISSING` (received less number of cells which have been transmitted)
`VCS_ERR_CELL_CORRUPTION` (received corrupted testing cells)

8.8 Connection Management

Connection Management at VC level

vcsConnSetup

A connection request from User, which shall contain traffic parameters such as bandwidth and QOS service. The connection path could be from WAN port to Loop port (downstream), Loop port to WAN port (Upstream), or Loop to Loop port, Microprocessor port to WAN or Loop port. The chipset driver may honor or reject the connection request based on the resource availability. The associated Line/WAN card should have already been added to system and the port should be enabled and setup in a non-Loopback mode, or the request would be rejected. If the requested is honored, the chipset driver sets up and enables the VC connection by configuring the appropriate chipset devices, and returns a unique connection ID.

The API configures the Connection Context Tables in both APEX and ATLAS, and enables cell rate policing (by ATLAS), and congestion control service (by APEX) based on the QOS contract.

Note 1: the OAM support is, by default, disabled for the connection by setting the chipset (ATLAS) as a non-termination point. Therefore, all OAM cells will be passed through transparently. To setup and enable the OAM support, one must specifically call `vcsOAMSetup()` API.

Note 2: If the VC is in upstream direction and the VC belongs to a shaped port/class, the per-VC shaping context is determined by calling a utility function "`sysVcsVCShaping`()", which converts the QOS request to the Shaper Rate parameters.

Valid States VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsConnSetup(sVCS_CONN_REQUEST
*psConnRequest, UINT2 *pConnID)
```

Inputs `psConnRequest` : connection request

Outputs `pConnID` : connection ID (from `0` to `maxVCs-1`)

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_HANDLE
VCS_ERR_INVALID_STATE
VCS_ERR_CONN_FULL
VCS_ERR_INVALID_VC_REQUEST
VCS_ERR_OUT_OF_RESOURCE
VCS_ERR_CONN_REDUNDANT
VCS_ERR_PORT_NOT_READY
VCS_ERR_ACTIVE_CORE_CARD
```

Pseudocode
```
Begin
Consult with CAC to see if sufficient resources
are available to accommodate the requested
connection, and determine which core card to
service the VC connection.
If yes, call connection configuration function
End
```

vcsConnTeardown

Tears down a specified VC connection in the chipset. Its associated resource is recycled to CAC.

Valid States VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsConnTeardown(UINT2 u2ConnID)
```

Inputs `u2ConnID` : connection ID for the connection to be shut down

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_CONNID
VCS_ERR_MULTICAST_CONN (the connection can't be
   removed by the API if it is a multicasting VC)
```

Pseudocode
```
Begin
Call an appropriate connection teardown function
Recycle the resource to CAC
End
```

vcsConnQOSRetrieve

Used to retrieve the connection traffic parameter or QOS parameters, such as peak cell rate, class.

Valid States VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsConnQOSRetrieve(UINT2 u2ConnID,
sVCS_VC_QOS *psQos)
```

Inputs
`u2ConnID` : connection ID (from 0 to maxVCs-1).

Outputs
`psQos` : contains current QOS parameters.

Return Codes
`VCS_SUCCESS`
`VCS_ERR_INVALID_STATE`
`VCS_ERR_INVALID_CONNID`

vcsConnQOSUpdate

Used to update the connection traffic parameter or QOS parameters, such as peak bandwidth. The traffic type and/or WFQ VC weight change are not supported by the API. The request might be rejected by the chipset driver due to resource limitation.

Note: If the VC is in upstream direction and the VC belongs to a shaped port/class, the per-VC shaping context is re-determined by calling a utility function "`sysVcsVCShaping`()", which converts the new QOS request to the Shaper Rate parameters.

Valid States VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsConnUpdate(UINT2 u2ConnID, sVCS_VC_QOS
*psQos)
```

Inputs
`u2ConnID` : connection ID.
`psQos` : contains requested QOS parameters.

Outputs None

Return Codes
`VCS_SUCCESS`
`VCS_ERR_INVALID_STATE`
`VCS_ERR_OUT_OF_RESOURCE`
`VCS_ERR_INVALID_CONNID`
`VCS_ERR_TRFC_TYPE` (new QOS require Class Id change)

vcsConnDisable

Disables a VC connection after it has been configured (using `vcsConnSetup`).). All incoming cells of the VC connection will be discarded. It clears the VC context Record of APEX device, while the Active bit in the associated ATLAS Ingress VC table is not changed.

Valid States VCS_ACTIVE

Side Effects None

Prototype `INT4 vcsConnDisable(UINT2 u2ConnID)`

Inputs `u2ConnID` : connection ID for the connection to be disabled

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_CONNID
VCS_ERR_INVALID_VC_STATE
```

vcsConnEnable

Re-enables a disabled VC connection. Cells of the VC connection can now pass through the chipset. It sets up the VC context Record of APEX device, while the Active bit in the associated ATLAS Ingress VC table is not changed.

Valid States VCS_ACTIVE

Side Effects None

Prototype `INT4 vcsConnEnable(UINT2 u2ConnID)`

Inputs `u2ConnID`: connection ID for the connection to be enabled

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_CONNID
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_VC_STATE
```

vcsConnStatus

Checks the current status of a connection.

Valid States VCS_ACTIVE

Side Effects None

Prototype `INT4 vcsConnStatus(UINT2 u2ConnID, sVCS_VC_STATUS *psConnStatus)`

Inputs `u2ConnID` : connection ID

Outputs `psConnStatus` : contains the status information of the connection.

Return Codes
`VCS_SUCCESS`
`VCS_ERR_INVALID_STATE`
`VCS_ERR_INVALID_CONNID`

Connection Management at Port Level

vcsPortSetup

Sets up and enables a specified Loop/WAN port. It also specifies the maximum throughputs which would be allowed in either (Inbound or outbound) direction of the port. If the request is honored by CAC module, the function also setups and enables all four classes for the port.

Valid States VCS_ACTIVE

Side Effects The routine will change the congestion thresholds for all the ports in the direction of the new port (i.e. all active loop ports if a loop port is being configured) and all the classes for these ports.

Prototype
```
INT4 vcsPortSetup(sVCS_PORT_ID sPortId, UINT4
u4InCellRate, UINT4 u4OutCellRate,
sVCS_PORT_THRSH_REQUEST *pPortThrshRqt, UINT1
u1Flag)
```

Inputs

`sPortId` : contains port type (can be WAN, LOOP or uP port), and port number for LOOP/WAN port

`u4InCellRate` : maximum cell rate limit in inbound direction of the port

`u4InCellRate` : minimum cell rate limit in outbound direction of the port

`pPortThrshRqt` : contain per-port threshold level request

`u1Flag` : forcing flag for per-port `maxThreshold` `clp0Threshold`, and `clp1Threshold`.

If 0, these three threshold levels are provided by the chipset driver using the default algorithm in the utility functions (in the file vcs_sys.c).

If 1, these three threshold values in the per-port threshold request (pointed by `pPortThrsh`) are used. No dynamicaly adjustment will be made to the port threshold levels even when spare resource is available for cell buffering..

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_PORT_ID
VCS_ERR_PORT_ALREADY_CFG
VCS_ERR_EXCEED_MAX_PORT_RATE
VCS_ERR_INVALID_THRSH
VCS_ERR_INVALID_CELL_RATE
```

Pseudocode

```
Begin
Consult with CAC to see if the port is available.
Call an appropriate port setup function
update the CAC data block
End
```

vcsPortTeardown

Tears down a specified Loop/WAN port, and all VC connections and classes associated with the port. Its associated resource is recycled to CAC.

Valid States VCS_ACTIVE

Side Effects The routine will change the congestion thresholds for all the ports in the direction of the port being cleared (i.e. all active loop ports if a loop port is being tear down) and all the classes for these ports.

Prototype `INT4 vcsPortTeardown(sVCS_PORT_ID sPortId)`

Inputs `sPortId` : contains port type (can be WAN, LOOP or uP port), and port number for LOOP or WAN port

Outputs None

Return Codes

```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_PORT_ID
VCS_ERR_PORT_NOT_CFG
```

Pseudocode

```
Begin
consult with CAC for a list of related VCs.
teardown the VCs in the list
call an appropriate port teardown function
recycle the associated resource to CAC
End
```

vcsPortDisable

Disables all the VC connections associated with a specified Loop/WAN/Microprocessor port, i.e. cells received from or destined to the port will be discarded. However, the affected VC connections and classes still exist.

The API clears the PortEn bit in the Port context Record of APEX device, which forces the discard of cells destined to the port. It also clears the Active bits in the ATLAS Ingress VC tables for all the VC originated from the port.

Valid States VCS_ACTIVE

Side Effects None

Prototype `INT4 vcsPortDisable(sVCS_PORT_ID sPortId)`

Inputs `sPortId` : contains port type (can be WAN, LOOP or uP port), and port number for LOOP or WAN port

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_PORT_ID
VCS_ERR_PORT_NOT_ENABLED
```

vcsPortEnable

Re-enables a disabled Loop/WAN/Microprocessor port. It sets the PortEn bit in the Port context Record of APEX device, and sets the Active bits in the ATLAS Ingress VC tables for all the VC originated from the port.

Note: those VCs which were disabled by calling `vcsConnDisable`, will remain in the Inactive state even though the connections are associated with the port being enabled. One must call `vcsConnEnable` API to re-enable the individual VC. The approach gives the USER a flexibility to control the connections independently at VC and port levels.

Valid States VCS_ACTIVE

Side Effects None

Prototype `INT4 vcsPortEnable(sVCS_PORT_ID sPortId)`

Inputs `sPortId` : contains port type (can be WAN, LOOP or uP port), and port number for LOOP or WAN port

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_PORT_ID
VCS_ERR_PORT_NOT_CFG
VCS_ERR_PORT_ALREADY_ENABLED
```

vcsPortStatus

It reports the status information of a specified port. The information includes the number of VC connections associated with the port.

Valid States VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsPortStatus(sVCS_PORT_ID sPortId,
sVCS_PORT_INFO *psPortInfo, UINT2 *pu2VcNum,
UINT2 **ppu2ConnId)
```

Inputs
- `sPortId` : contains port type (can be WAN, LOOP or uP port), and port number for LOOP or WAN port

Outputs
- `psPortInfo` : contains the current port status information.
- `pu2VcNum` : contains the number of connections associated with the port
- `ppConnId` : point to an array of ConnId buffer, which is allocated by the API and contains a list of connection ID. The size is "2 * u2VcNum" bytes. It is a responsibility of the caller to free the memory buffer.

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_HANDLE
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_PORT
```

Connection Management at Chipset or Module Level

vcsClearVCs

Tears down all existing connections, channels, shapers and ports. If used together with `vcsReset`, this means that both hardware and software have been reset, and the chipset needs to be re-initialized from scratch.

Valid States VCS_ACTIVE

Side Effects None

Prototype `INT4 vcsClearVCs(VOID)`

Inputs `None`

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INTERNAL (driver internal error)
```

vcsRebuildVCs

Re-builds the existing connections, ports, channels, OAM and F4toF5 processing list which are maintained in the software, after a specified core card has been reset by `vcsReset`. This provides a fast way to recover the system to the pre-reset state. The caller should call `vcsInit` and `vcsActivate` after the reset, before calling the API function.

Valid States VCS_ACTIVE

Side Effects None

Prototype `INT4 vcsRebuildVCs(VCS vcs)`

Inputs `vcs` : chipset Handle

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_HANDLE
VCS_ERR_INVALID_STATE
```

vcsConnInfo

Reports information on the current active VCs, and loads on each chipset (core card). It also indicates available resources, such as number of empty VCs, logical channels and ports.

Valid States VCS_PRESENT, VCS_STANDBY, VCS_ACTIVE

Side Effects None

Prototype `INT4 vcsConnInfo(sVCS_CONN_INFO *psConnInfo)`

Inputs None

Outputs `psConnInfo` : contains the system information for the VC connections and available resources.

Return Codes `VCS_SUCCESS`
`VCS_ERR_CARD_ID` (internal error in active card ID)

8.9 Shaper support

The following APIs are used to configure any of 4 Shapers, and associate 4 out of 16 possible WAN Port/Class to the 4 shapers in APEX.

vcsShprSetup

Configures and enable a shaper based on a shaper-parameters vector passed by the user. It is used for shaping traffic on one of the WAN ports.

Note: A shaper should be configured before any of its associated port-classes are configured. Also, a shaper should be configured before the APEX queue engine is enabled (i.e., before APEX device activation)

Valid States `VCS_STANDBY`

Side Effects None

Prototype
```
INT4 vcsShprSetConfig(UINT1 u1ShprNum,
sVCS_SHPR_VECT *psShprVect)
```

Inputs

`u1ShprId`	: shaper to be configured	(0-3)
`psShprVect`	: shaper parameters vector that is used by driver to program the APEX port context records.	

Outputs None

Return Codes

`VCS_SUCCESS`	
`VCS_ERR_INVALID_STATE`	(chipset is not in a valid state)
`VCS_ERR_INVALID_SHPR_NUM`	(invalid shaper number)
`VCS_ERR_INVALID_SHPR_STATE`	(shaper is already used)
`VCS_ERR_INVALID_SHPR_RATE`	(invalid shaper vector)

vcsShprTeardown

Tears down a shaper. Shaper can only be torn down if the APEX queue engine is disabled (i.e., APEX device is de-activated). Note that a shaper should be torn down only after its associated port is torn down.

Valid States `VCS_STANDBY`

Side Effects None

Prototype
```
INT4 vcsShprTeardown(UINT1 u1ShprId)
```

Inputs `u1ShprId` : shaper to be torn down (0-3)

Outputs None

Return Codes

`VCS_SUCCESS`	
`VCS_ERR_INVALID_STATE`	(chipset is not in a valid state)
`VCS_ERR_INVALID_SHPR_NUM`	(invalid shaper number)
`VCS_ERR_INVALID_SHPR_STATE`	(shaper was not used)

8.10 Data Tx via Microprocessor port

The following APIs are used to transmit cells or frames via a microprocessor data connection, which is established between the APEX microprocessor port and a loop/WAN port.

vcsConnTxCell

Transmit a data cell over a specified microprocessor port connection established by `vcsConnSetup` or `vcsMcSetup` call.

Valid States VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsConnTxCell(UINT2 u2ConnID, sVCS_CELL_HDR
*pHdr, UINT1 *pPyld, UINT1 u1crcFlag)
```

Inputs

`u2ConnID` : connection ID.

`psHdr` : pointer to the cell header structure that contains the header bytes.

`pu1Pyld` : pointer to first byte of cell payload (48 contiguous bytes)

`u1CrcFlg` : this is a control flag; can be one of –

0 - no CRC protection required
1 – overwrite end-of-cell with CRC-10.

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_CONNID
VCS_ERR_INVALID_VC_STATE
VCS_ERR_NO_ACTIVE_CARD
```

vcsConnTxFrame

Transmits a data frame over a specified microprocessor port connection established by `vcsConnSetup` or `vcsMcSetup` call.

Valid States VCS_ACTIVE

Side Effects None

Prototype `INT4 vcsConnTxFrame(UINT2 u2ConnID, sVCS_CELL_HDR *pHdr, UINT1 *pFrame, UINT4 u4Length)`

Inputs

`u2ConnID` : connection ID.

`psHdr` : cell header to be transmitted with each cell in each frame.

`pu1Frame` : pointer to first byte of frame (buffer)

`u4Length` : frame length (in bytes)

Outputs None

Return Codes

```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_CONNID
VCS_ERR_INVALID_VC_STATE
VCS_ERR_NO_ACTIVE_CARD
```

8.11 Multicast support

vcsMcSetup

A multicasting connection request from User, which shall contain traffic parameters such as bandwidth and QOS service. The connection path could be from a WAN port to multiple Loop ports (downstream), from a Loop port to multiple WAN ports (Upstream), from a Loop to multiple Loop ports, or a Microprocessor port to multiple WAN/Loop ports. The chipset driver may honor or reject the connection request based on the resource availability. If honored, the chipset driver sets up and enables the multicasting connection by configuring the appropriate chipset devices, and returns a unique Multicasting ID. The multicasting group is also registered with the Multicast Record Table.

Valid States VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsMcSetup(sVCS_MULTICAST_REQUEST
*psVcsMcRequest, UINT4 *pMcastID)
```

Inputs `psVcsMcRequest`: contains a Multicasting Connection request

Outputs `pMcastID` : multicasting ID.

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_MCAST_PORT
VCS_ERR_INVALID_MCAST_REQUEST
VCS_ERR_MEM_ALLOC
```

vcsMcTeardown

Tears down a specified multicasting connection group. The group is also removed from the Multicast Record Table.

Valid States VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsMcTeardown(UINT4 u4McastID)
```

Inputs `u4McastID` : multicasting ID for the multicasting connection group to be shut down.

Outputs None

Return Codes `VCS_SUCCESS`
`VCS_ERR_INVALID_STATE`
`VCS_ERR_INVALID_MCAST_ID`

vcsMcAddConn

Adds an outgoing connection for a particular vpi/vci/port to an existing multicasting group.

Valid States VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsMcAddConn(UINT4 u4McastID,
sVCS_VC_PORT_DES *psVcPort)
```

Inputs
`u4McastID` : a multicasting ID
`psVcPort` : a VC/port descriptor of an outgoing connection to be added.

Outputs None

Return Codes `VCS_SUCCESS`
`VCS_ERR_INVALID_STATE`
`VCS_ERR_INVALID_MCAST_ID`
`VCS_ERR_INVALID_MCAST_PORT`

vcsMcDropConn

Removes an outgoing connection for a particular VC/portfrom an existing multicasting group.

Valid States VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsMcDropConn(UINT4 u4McastID,
sVCS_VC_PORT_DES *psVcPort)
```

Inputs
`u4McastID` : a multicasting ID

`psVcPort` : a VC/port descriptor of an outgoing connection to be dropped.

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_MC_ID
VCS_ERR_VC_NOT_FOUND
```

vcsMulticastCell

Transmits a cell on each outgoing connection of the multicasting group.

Valid States VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsMulticastCell(UINT4 u4McastGroupId,
sVCS_CELL_HDR *pHdr, UINT1 *pPyld, UINT1
u1CrcFlag)
```

Inputs

`u4McastGroupId`	: ID identifying the multicasting group
`pHdr`	: header for the cell
`pPyld`	: payload of the cell
`u1CrcFlag`	: flag indicating whether the end of the cell should be overwritten with CRC10

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_MCAST_ID
VCS_ERR_NO_ACTIVE_CARD
VCS_ERR_INVALID_VC_STATE
```

vcsMulticastFrame

Transmits a frame on each outgoing connection of the multicasting group.

Valid States VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsMulticastFrame (UINT4 u4McastGroupId,
sVCS_CELL_HDR *pHdr, UINT1 *pFrame, UINT4
u4Length)
```

Inputs

`u4McastGroupId`	: ID identifying the multicasting group
`pHdr`	: header for the cell
`pFrame`	: payload for the frame
`u4Length`	: length of the payload in bytes

Outputs None

Return Codes

```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_MCAST_ID
VCS_ERR_NO_ACTIVE_CARD
VCS_ERR_INVALID_VC_STATE
```

8.12 Inband Control Channels

vcsCtrlChnlSetup

Sets up an inband, bi-directional control channel between a core card and a remote Line or WAN card. It allocates all necessary resource to prepare for the subsequent inband message Tx/Rx.

For a channel between a line and core card, the data path (as shown in Figure 19) requires the establishment of two VC table records in APEX and ATLAS, which route the control messages between the APEX microprocessor port and the microprocessor port of DUPLEX device on the remote line card.

For a channel between a WAN and core card, the data path (as shown in Figure 19) doesn't require any establishment of VC table records in APEX and ATLAS. The messages are passed between the microprocessor ports of two DUPLEX devices on the remote WAN card and core card.

Note the Receive VPI/VCI in the sVCS_CHNL_REQUEST data structure should be unique among the control channels.

Valid States VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsCtrlChnlSetup(sVCS_CHANNEL_REQUEST
*psChnlRequest, UINT2 *pu2ChnlID)
```

Inputs `psChnlRequest` : inband channel request

Outputs `pu2ChnlID` : channel ID (index of the channel record table), from 0 to `maxChnls-1`.

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_MEM_ALLOC
VCS_ERR_CHNL_FULL
VCS_ERR_INVALID_CHNL_REQ
VCS_ERR_CHNL_DUPLICATE_VPCI
```

vcsCtrlChnlTeardown

Shuts down a specified control channel between a core card and a remote Line or WAN card. It also de-allocates the associated resource.

Valid States VCS_ACTIVE

Side Effects None

Prototype `INT4 vcsCtrlChnlTeardown(UINT2 u2ChnlID)`

Inputs `u2ChnlID` : ID of the channel to be shut down

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_CHNL_ID
```

vcsCtrlChnlTx

Sends a message to a remote card over a specified control channel.

Valid States VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsCtrlChnlTx(UINT2 u2ChnlID, UINT1 *pMsg,
UINT4 u4Length)
```

Inputs `u2ChnlID` : channel ID

`pMsg` : pointer to a message buffer to be sent

`u4Length` : length of message in bytes.

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_CHNL_ID
VCS_ERR_CTRL_MSG_LENGTH
```

vcsCtrlChnlRx

Retrieves a message that was received from a remote card over a specified control channel.

Valid States VCS_ACTIVE

Side Effects None

Prototype `INT4 vcsCtrlChnlRx(UINT2 u2ChnlID, UINT1 *pMsg, UINT4 * pLength)`

Inputs `u2ChnlID` : channel ID

Outputs `pMsg` : pointer to a message buffer received.

`pLength` : length of the message in bytes.

Return Codes
`VCS_SUCCESS`
`VCS_ERR_INVALID_STATE`
`VCS_ERR_INVALID_CHNL_ID`

8.13 BOC Signaling

vcsBOCTx

Sends a BOC signal to a remote card over a specified HSS link.

Valid States VCS_ACTIVE

Side Effects None

Prototype `INT4 vcsBOCTx(UINT1 u1CardId, UINT1 u1BOCcode)`

Inputs `u1CardID` : identify a remote Line or WAN card, by specifying HSS line, and VORTEX or DUPLEX device, to which it is connected.
bit 0-2: HSS link number (0 to 7)
bit 3-6: device number
bit 7 : 0=VORTEX, 1 = DUPLEX

`u1BOCcode` : BOC code to be sent.

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_CARD_ID
VCS_ERR_PORT_NOT_READY
```

vcsBOCRx

Retrieves a BOC signal from a remote card over a specified HSS link.

Valid States VCS_ACTIVE

Side Effects None

Prototype `INT4 vcsBOCRx(UINT1 u1CardId, UINT1 *pBOCcode)`

Inputs `u1CardID` : identify a remote Line or WAN card, by specifying HSS line, and VORTEX DUPLEX device, to which it is connected.
bit 0-2: HSS link number (0 to 7)
bit 3-6: device number
bit7 : 0=VORTEX, 1 = DUPLEX

Outputs `pBOCcode` : contains a BOC code received.

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_CARD_ID
VCS_ERR_PORT_NOT_READY
```

8.14 Addition/Deletion of Line/WAN Cards

vcsAddCard

Adds a Line or WAN card to the system. This shall activate the associated HSS link, and mark the availability of the corresponding loop or WAN ports in the resource database of the chipset driver.

Note: the API should not be called when the chipset card is in Loopback testing mode.

Valid States VCS_STANDBY, VCS_ACTIVE

Side Effects None

Prototype `INT4 vcsAddCard(VCS vcs, UINT1 u1CardId)`

Inputs

`vcs` : chipset handler for the core card, to which the remote card is actively connected.

`u1CardID` : specifies a HSS line of VORTEX or DUPLEX device, to which the remote card is connected.
bit 0-2: HSS link number (0 to 7)
bit 3-6: device number
bit7 : 0=VORTEX (Line card),
1 = DUPLEX (WAN card)

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_HANDLE
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_CARD_ID
VCS_ERR_CARD_ALREADY_ADD
VCS_ERR_LPBKPORT_IN_USE
```

vcsRemoveCard

Removes a Line or WAN card from the system. It tears down the ports in the APEX Port and Class Context Records after all related VC connections being deleted. This shall also de-activate the associated HSS link, and mark the non-availability of the corresponding loop or WAN ports in the chipset driver resource database to prevent any future request for connections over the deleted ports.

Note: the API should not be called when the chipset card is in Loopback testing mode.

Valid States VCS_STANDBY, VCS_ACTIVE

Side Effects None

Prototype `INT4 vcsRemoveCard(UINT1 u1CardId)`

Inputs `u1CardID` : specifies a HSS line of VORTEX or DUPLEX device, to which the remote card is connected.
bit 0-2: HSS link number (0 to 7)
bit 3-6: device number
bit 7 : 0=VORTEX (Line card),
1 = DUPLEX (WAN card)

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_CARD_ID
VCS_ERR_CARD_NOT_ADDED
VCS_ERR_LPBKPORT_IN_USE
```

vcsRemoteCardInfo

Reports the availability of remote Line/WAN cards at each HSS link of a specified core card, as well as the HSS link status, and total number of remote cards being added.

Valid States VCS_STANDBY, VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsRemoteCardInfo (VCS vcs, sVCS_RCARD_INFO
            *psCardInfo))
```

Inputs `vcs` : chipset handler for the core card

Outputs `psCardInfo` : contains Remote Card and HSS link status information

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_HANDLE
VCS_ERR_INVALID_STATE
```

8.15 VC OAM (FM and PM) Setup

The OAM support only applies to upstream or downstream connections between Loop and WAN ports, not the connections associated with microprocessor port.

OAM At Connection Level

vcsVcOAMSetup

Sets up and enables an OAM (FM and PM) support over an existing F4 or F5 connection. It configures the chipset (ATLAS) as a terminating point (Segment Point, or End_to_End Point, or both) for OAM cells on the VCconnection, and associates a backward connection for its RDI and Backward Reporting PM cell flow. The connection and its backward connection should be already setup, and their paths must be associated with the same Loop port and WAN port.

Note 1: By default, the F4 to F5 processing is disabled, i.e. the VPC pointer is set to the address of the VC (or point to itself). One must specifically call `vcsF4toF5Setup` API to setup and enable the termination of F4 (VPC) to F5 (VCC).

Note 2: the PM sessions should already be properly configured before the sessions are associated with the connection. Each connection may participate in two active PM sessions at the same time, and multiple connections may share one PM session. A common implementation of PM will involve monitoring and generation at the F4 and F5 level. This means that for an F4 pipe, each constituent F5 connection will have one F5 PM session and one F4 PM session (both active at the same time), while all constituent F5 connections point to the same F4 PM session.

Valid States VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsVcOAMSetup(UINT2 u2InConnID,
sVCS_VC_OAM_REQUEST sVcOAM, UINT2 u2BackConnID)
```

Inputs

`u2InConnID` : connection ID for incoming OAM cell path

`sVcOAM` : contains per-VC OAM configuration parameters

`u2BackConnID`: connection ID for backward OAM cell path.
Note: If the incoming connection is F4 OAM type, the backward connection is required to be the same type of the F4 OAM connection.

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_CONNID
VCS_ERR_BACKWARD_VC
VCS_ERR_OAM_ALREADY_SETUP
VCS_ERR_CONN_MP_ORIGIN
VCS_ERR_MEM_ALLOC
```

vcsVcOAMClear

Disables the OAM support over a specified F4 or F5 connection. It re-configures the chipset (ATLAS) as a non-end-point for OAM cells on the VCC connection. Therefore, all incoming OAM cells will be passed transparently through the chipset. The PM measurement for the connection is disabled too.

Valid States VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsVcOAMClear(UINT2 u2ConnID)
```

Inputs `u2ConnID` : connection ID for incoming OAM path.

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_CONNID
VCS_ERR_OAM_NOT_SETUP
```

vcsVcOAMRetrieve

Used to retrieve the current OAM Configurations (PM and FM) on a specified connection.

Valid States VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsVcOAMRetrieve(UINT2 u2ConnID, sVCS_VC_OAM
*psVcOAM)
```

Inputs `u2ConnID` : connection ID

Outputs `psVcOAM` : contains current OAM parameters.

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_CONNID
VCS_ERR_OAM_NOT_SETUP
```

vcsVcFMUpdate

Used to update/modify the FM (RDI, AIS, CC) part of the OAM Configurations on a specified connection. The backward path is not changed.

Valid States VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsVcFMUpdate(UINT2 u2ConnID, sVCS_VC_OAM_FM
psVcFM)
```

Inputs

`u2ConnID` : connection ID

`psVcFM` : contains per-VC FM configuration parameters

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_CONNID
VCS_ERR_OAM_NOT_SETUP
```

vcsVcPMUpdate

Used to update/modify the PM part of the OAM Configurations on a specified connection. The backward path is not changed.

Valid States VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsVcPMUpdate(UINT2 u2ConnID, sVCS_VC_OAM_PM
psVcPM)
```

Inputs

`u2ConnID` : connection ID

`psVcPM` : contains per-VC PM configuration parameters

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_CONNID
VCS_ERR_OAM_NOT_SETUP
```

vcsVcOAMGetDefect

Used to retrieve the received OAM defect type and location on a specified connection.

Valid States VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsVcOAMGetDefect(UINT2 u2ConnID,
sVCS_VC_OAM_DEFECT *psVcOAMDefect)
```

Inputs `u2ConnID` : connection ID

Outputs `psVcOAMDefect` : contains OAM defect type and location.

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_CONNID
VCS_ERR_OAM_NOT_SETUP
```

OAM At Chipset Level

vcsOAMSetConfig

Used to setup the OAM configuration registers on a specified chipset card. The configuration is not on per-connection basis, i.e. It will globally effect all connections associated with the chipset card.

Valid States VCS_STANDBY, VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsOAMSetConfig(VCS vcs, UINT1 u1Dir,
sVCS_OAM_CFG *psOAM)
```

Inputs `vcs` : chipset handler

`u1Dir` : OAM Flow Direction, either eVCS_ATLAS_EGRESS or eVCS_ATLAS_INGRESS

`psOAM` : contains OAM parameters to be configured.

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_HANDLE
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_FLAG (invalid u1Dir flag)
```

vcsOAMGetConfig

Used to get the OAM configuration information on a specified chipset card.

Valid States VCS_STANDBY, VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsOAMGetConfig(VCS vcs, UINT1 u1Dir,
sVCS_OAM_CFG *psOAM)
```

Inputs `vcs` : Chipset handler

`u1Dir` : OAM Flow Direction, either eVCS_ATLAS_EGRESS or eVCS_ATLAS_INGRESS

Outputs `psOAM` : contains current OAM configurations.

Return Codes

```
VCS_SUCCESS
VCS_ERR_INVALID_HANDLE
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_FLAG (invalid u1Dir flag)
```

8.16 F4 to F5 OAM Processing

vcsF4toF5Setup

Sets up and enables the F4 to F5 OAM processing support.

Note: (1) End-to-end OAM connection of the VPC is required to be setup as a connection end-point before the API being called. i.e. the end-to-end termination OAM support should be already setup and enabled on the F4 (VPC) connection.

(2) Segment OAM connection of F4 (VPC) is optional in the F4-to-F5 OAM processing.Setting the field "F4SegConnId" with the same value as "F4EtoEConnId" in the sVCS_F4TOF5_REQUEST, means that the Segment VPC connection is disabled in the F4toF5 processing chain.

(3) OAM terminations on its constituent F5 (VCC) connections are not required, although they might be setup and enabled if desired.

Valid States VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsF4toF5Setup(sVCS_F4TOF5_REQUEST
*psF4toF5Request)
```

Inputs `psF4toF5Request` : contains the F4 to F5 processing request.

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_VPC_REQUEST
VCS_ERR_F4TOF5_ALREADY_SETUP
VCS_ERR_VC_TYPE (invalid VC type)
VCS_ERR_INVALID_VCC_F4TOF5_CFG
```

vcsF4toF5Clear

Disables the F4 to F5 OAM processing over a specified F4 VPC connection by removing all the associated VCCs from the processing list.

Valid States VCS_ACTIVE

Side Effects None

Prototype `INT4 vcsF4toF5Clear(UINT2 u2F4ConnID)`

Inputs `u2F4ConnID` : F4 OAM connection ID.

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_CONNID
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_VPC_REQUEST
VCS_ERR_F4TOF5_NOT_SETUP
VCS_ERR_INTERNAL
```

vcsF4toF5AddVcc

Adds a VCC connection to a F4 to F5 OAM processing list associated with a specified F4 VPC connection.

Valid States VCS_ACTIVE

Side Effects None

Prototype `INT4 vcsF4toF5AddVcc(UINT2 u2F4ConnID, UINT2 u2VccId, UINT1 u1F4ToF5Cfg)`

Inputs `u2F4ConnID` : F4 OAM connection ID.

`u2VccId` : connection ID of the VC to be added

`u1F4ToF5Cfg` : F4 to F5 processing Configuration for the VCC.
Bit 0: F4toF5AIS,
1= F5 FM cells will be generated at F4 OAM termination;
0= F5 FM cells will not be generated at F4 OAM termination; only CC cells will be generated.
Bit 1: Segment Flow,
1 = An F5 Segment AIS cell will be generated while the F4 connection is in AIS alarm.
0 = an F5 end-to-end AIS will be generated instead. The bit should be set when the VCC connection is within a defined segment or not a VC end-point, i.e. the VCC extends beyond the end-point of the VPC.

Note: the bit should not be set to 1 if the VCC set as a Segment end-point

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_CONNID
VCS_ERR_INVALID_VPC_REQUEST
VCS_ERR_F4TOF5_NOT_SETUP
VCS_ERR_VCC_ALREADY_IN_LIST
VCS_ERR_INTERNAL
VCS_ERR_VC_TYPE
VCS_ERR_INVALID_VCC_F4TOF5_CFG
```

vcsF4toF5DropVcc

Drops a VCC connection from a F4 to F5 OAM processing list associated with a specified F4 VPC connection.

Valid States VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsF4toF5DropVcc(UINT2 u2F4ConnID, UINT2
u2VccId)
```

Inputs

`u2F4ConnID` : an F4 OAM connection ID.

`u2VccId` : connection ID of the VC to be droped

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_CONNID
VCS_ERR_VCC_NOT_IN_LIST
VCS_ERR_F4TOF5_NOT_SETUP
VCS_ERR_INVALID_VPC_REQUEST
VCS_ERR_INTERNAL
```

8.17 PM Session Configuration/Status

The following APIs allow the USER to configure PM sessions, and retrieve the statistic performance record accumulated by the PM sessions. It's up to USER to interpret the performance record data reported by the PM session.

Note: In the architecture of the DSLAM reference design, only the ATLAS Ingress PM sessions are used. Therefore, there are total 256 sessions available to the USER, which are divided into two banks, with128 PM sessions for each bank.

vcsPMSetConfig

Configures a PM session at a specified PM address (0 to 127) on a specified bank (BANK1 or BANK2)

Valid States VCS_STANDBY, VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsPMSetConfig(UINT1 u1SessionID,
eVCS_PM_BANK ePmBank,  UINT2 u2PmCfg)
```

Inputs

`u1SessionID` : PM session ID (0 to 127)

`ePmBank` : PM Bank, could be one of eVCS_PM_BANK1 or eVCS_PM_BANK2.

`u2PmCfg` : contains the 16 bit long PM configuration for the session.

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_SESSION_ID
VCS_ERR_INVALID_BANK
VCS_ERR_INVALID_PM_CFG
```

vcsPMGetConfig

Retrieves the configures of a PM session at a specified PM address (0 to 127) on a specified bank (BANK1 or BANK2)

Valid States VCS_STANDBY, VCS_ACTIVE

Side Effects None

Prototype `INT4 vcsPMGetConfig(UINT1 u1SessionID, eVCS_PM_BANK ePmBank, UINT2 *pu2PmCfg)`

Inputs `u1SessionID` : PM session ID (0 to 127)

`ePmBank` : PM Bank, could be one of eVCS_PM_BANK1 or eVCS_PM_BANK2.

Outputs `pu2PmCfg` : current PM configurations (16 bit long) for the session.

Return Codes
`VCS_SUCCESS`
`VCS_ERR_INVALID_HANDLE`
`VCS_ERR_INVALID_STATE`
`VCS_ERR_INVALID_SESSION_ID`
`VCS_ERR_INVALID_BANK`

vcsPMReadRecord

Reads a Performance record from a specified PM session

Valid States VCS_STANDBY, VCS_ACTIVE

Side Effects None

Prototype `INT4 vcsPMConfig(VCS vcs, UINT1 u1SessionID, eVCS_PM_BANK ePmBank, sVCS_PM_RECORD *psRecord)`

Inputs `vcs` : Chipset handler

`u1SessionID` : PM session ID (0 to 127)

`ePmBank` : PM Bank, could be one of `eVCS_PM_BANK1` or `eVCS_PM_BANK2`.

Outputs `psRecord` : contains the PM record data for the session. Note: the structure `sVCS_PM_RECORD` is the same as `sATLS_PM_RECORD`.

Return Codes
`VCS_SUCCESS`
`VCS_ERR_INVALID_HANDLE`
`VCS_ERR_INVALID_STATE`
`VCS_ERR_INVALID_SESSION_ID`
`VCS_ERR_INVALID_BANK`

8.18 Protection Switching

vcsRemoveLoad

Removes the connection load from a specified chipset card. The card should be in VCS_STANDBY or VCS_ACTIVE state before calling the function. Note: vcsRemoveLoad and vcsAddLoad should be used as a pair.

Valid States VCS_STANDBY, VCS_ACTIVE

Side Effects the core card will be in a hot-standby state.

Prototype `INT4 vcsRemoveLoad (VCS vcs)`

Inputs `vcs` : chipset handler for the core card, whose active connections or load are to be removed

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_HANDLE
VCS_ERR_INVALID_STATE
```

vcsAddLoad

Adds the connection load to a specified chipset card. The card should be in VCS_STANDBY or VCS_ACTIVE state. In addition, The load has to be removed first from its current serving chipset card before calling the function. Note: vcsRemoveLoad and vcsAddLoad should be used as a pair.

Valid States VCS_STANDBY, VCS_ACTIVE

Side Effects the core card takes over all connection load, including those hot-standby connections serviced by other core card.

Prototype `INT4 vcsAddLoad (VCS vcs)`

Inputs `vcs` : chipset handler for the core card which will take over the load.

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_HANDLE
VCS_ERR_INVALID_STATE
```

vcsRemoveLineLoad

Removes a part of the connection load, which is associated with a specified line card, from a chipset card. The active HSS links between the core card and line card are deactivated and the associated VC connection are disabled at ATLAS Ingress VC Table. This effectively prevents the connection data flow through the line card and core card.

Note: `vcsRemoveLineLoad` and `vcsAddLineLoad` should be used as a pair.

Valid States VCS_STANDBY, VCS_ACTIVE

Side Effects the effected active connections switched into hot-standby state on the core card.

Prototype
```
INT4 vcsRemoveLineLoad(VCS vcs,UINT1 u1CardID )
```

Inputs

`vcs` : chipset handler for the core card, whose active connections or load are to be removed

`u1CardId` : specifies a line card, whose associated connections are to be load switched.

Outputs None

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_HANDLE
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_CARD_ID
VCS_ERR_INVALID_LINE_CARD_ID
```

vcsAddLineLoad

Adds a part of the connection load, which is associated with a specified line card, to a chipset card. The part of load has to be removed first from its current serving chipset card before calling the function.

Note: `vcsRemoveLineLoad` and `vcsAddLineLoad` should be used as a pair.

Valid States VCS_STANDBY, VCS_ACTIVE

Side Effects the effected active connections switched into active state on the core card.

Prototype
```
INT4 vcsAddLineLoad(VCS vcs,UINT1 u1CardID )
```

Inputs

`vcs`	: chipset handler for the core card which will take over the load.
`u1CardId`	: specifies a line card, whose associated connections are to be load switched.

Outputs None

Return Codes

```
VCS_SUCCESS
VCS_ERR_INVALID_HANDLE
VCS_ERR_INVALID_STATE
VCS_ERR_INVALID_CARD_ID
VCS_ERR_INVALID_LINE_CARD_ID
```

8.19 Counter Configuration

vcsSetRxCntCfg

Sets up the configuration for the two VC level counters in ATLAS for counting the number of incoming cells (before policing). The two 32-bit cell counters can be programmed to count any combination of the following incoming cells: CLP0 user cells, CLP1 user cells, CLP0 OAM cells, CLP1 OAM cells, CLP0 RM cells, CLP1 RM cells, CLP0 cells with invalid VCI/PTI, CLP1 Cells with invalid VCI/PTI.

Valid States VCS_STANDBY, VCS_ACTIVE

Side Effects None

Prototype

```
INT4 vcsSetRxCntCfgs(VCS vcs,UINT1 u1CntSelect,
UINT1 u1Cnt1Cfg,UINT1 u1Cnt2Cfg)
```

Inputs

`vcs` : chipset handle

`u1CntSelect` :select flag for counting configurations.
if 0, Counting Configuration 1 is selected.
Otherwise, Counting Configuration 2 is selected.

`u1Cnt1Cfg` : configuration for counter 1.

`u1Cnt2Cfg` : configuration for counter 2.

Note: A logic 1 in the configuration bits enables counting on that particular stream.

Bit 7: CLP1 user cells
Bit 6: CLP0 user cells
Bit 5: CLP1 OAM cells
Bit 4: CLP0 OAM cells
Bit 3: CLP1 RM cells
Bit 2: CLP0 RM cells
Bit 1: CLP1 cells with PTI = 111 ,VCI = 7 to 15
Bit 0: CLP0 cells with PTI = 111 ,VCI = 7 to 15

Outputs None.

Return Codes

`VCS_SUCCESS`
`VCS_ERR_INVALID_HANDLE` (invalid chipset handle)
`VCS_ERR_INVALID_STATE` (chipset is not in a valid state)

vcsGetRxCntCfg

This function retrieves the configuration for the two VC level counters in ATLAS.

Valid States VCS_STANDBY, VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsGetRxCntCfgs(VCS vcs,UINT1
u1CntSelect,UINT1 *pu1Cnt1Cfg, UINT1 *pu1Cnt2Cfg)
```

Inputs

`vcs` : chipset handle

`u1CntSelect` : select flag for counting configurations. if 0, Counting Configuration 1 is selected. Otherwise, Counting Configuration 2 is selected.

Outputs

`pu1Cnt1Cfg` : contains configuration for counter 1.

`pu1Cnt2Cfg` : contains configuration for counter 2
Note: A logic 1 in the configuration bits indicates counting on that particular stream is enabled.

Bit 7: CLP1 user cells
Bit 6: CLP0 user cells
Bit 5: CLP1 OAM cells
Bit 4: CLP0 OAM cells
Bit 3: CLP1 RM cells
Bit 2: CLP0 RM cells
Bit 1: CLP1 cells with PTI = 111 ,VCI = 7 to 15
Bit 0: CLP0 cells with PTI = 111 ,VCI = 7 to 15

Return Codes

`VCS_SUCCESS`	
`VCS_ERR_INVALID_HANDLE`	(invalid chipset handle)
`VCS_ERR_INVALID_STATE`	(chipset is not in a valid state)

vcsSetNcCntCfgs

Sets up the configurations for the three VC level counters in ATLAS for counting the number of incoming non-compliant cells (as a result of policing).

The three 16-bit cell counters can be programmed to count offend cells: Non-compliant CLP0 cells, Non-compliant CLP0+1 cells, Tagged CLP0 cells, Discarded CLP0 cells, Discarded CLP0+1 cells.

Valid States VCS_STANDBY, VCS_ACTIVE

Side Effects None

Prototype
```
vcsSetNcCntCfgs(VCS vcs, eVCS_NON_COMPLIANT_TYPE
u1NcCnt1Cfg,  eVCS_NON_COMPLIANT_TYPE
u1NcCnt2Cfg, eVCS_NON_COMPLIANT_TYPE u1NcCnt3Cfg
)
```

Inputs

`vcs` : chipset handle

`u1NcCnt1Cfg` : configuration for counter 1. It can be one of VCS_NC_CLP0, VCS_NC_CLP01, VCS_DISCARD_CLP0, and VCS_DISCARD_CLP01

`u1NcCnt2Cfg` : configuration for counter 2.It can be one of VCS_NC_CLP0, VCS_NC_CLP01, VCS_DISCARD_CLP0, and VCS_DISCARD_CLP01

`u1NcCnt3Cfg` : configuration for counter 3. It can be one of VCS_NC_CLP0, VCS_NC_CLP01, VCS_TAG_CLP0, and VCS_DISCARD_CLP01,

Outputs None.

Return Codes

`VCS_SUCCESS`
`VCS_ERR_INVALID_HANDLE` (invalid chipset handle)
`VCS_ERR_INVALID_STATE` (chipset is not in a valid
`VCS_ERR_INVALID_CFG`

8.20 Statistical Counts

The statistical counts are cell counts that increase monotonically as they accumulate over time. There are four levels of counts: per VC, per port, per Line/WAN card, and per chipset

Cell Counts Per VC

vcsGetStatVcTxCnts

This function retrieves the connection level cell transmission counts. The counts are maintained by the APEX device

Valid States VCS_STANDBY, VCS_ACTIVE

Side Effects None

Prototype
```
INT4 vcsGetStatVcTxCnts(UINT2 u2ICI, UINT4
*pu4VcClp0TxCnt,UINT4 *pu4VcClp1TxCnt)
```

Inputs

u2ICI	: connection ID

Outputs

pu4VcClp0TxCnt	: count of all CLP0 cells transmitted.
pu4VcClp1TxCnt	: count of all CLP1 cells transmitted.

Return Codes

VCS_SUCCESS	
VCS_ERR_INVALID_CONNID	(invalid connection ID)
VCS_ERR_INVALID_STATE	(chipset is not in a valid state)

vcsGetStatVcRxCnts

This function retrieves the cell receive counts at the connection level. The counts are maintained by the ATLAS device. The two 32-bit cell counts are programmed to count any combination of the following incoming cells: CLP0 user cells, CLP1 user cells, CLP0 OAM cells, CLP1 OAM cells, CLP0 RM cells, CLP1 RM cells, CLP0 cells with invalid VCI/PTI. CLP1 Cells with invalid VCI/PTI. Typically, counter1 can be used to count CLP0 cells (including user OAM and RM cells), while the counter2 for CLP1 cells. The count type can be configured by calling API function vcsSetRxCntCfgs().

Valid States VCS_STANDBY, VCS_ACTIVE

Side Effects None

Prototype `INT4 vcsGetStatVcRxCnts(UINT2 u2ICI, sVCS_VC_STAT_CNT *pRxCnt1, sVCS_VC_STAT_CNT *pRxCnt2)`

Inputs `u2ICI` : connection ID

Outputs `pRxCnt1` : contains count type and count of cells received, which is maintained in counter 1.

`pRxCnt2` : contains count type and count of cells received, which is maintained in counter 2.

Return Codes `VCS_SUCCESS`
`VCS_ERR_INVALID_CONNID` (invalid connection ID)
`VCS_ERR_INVALID_STATE` (device is not in a valid state)
`VCS_ERR_CONN_MP_ORIGIN`

vcsGetStatVcNcCnts

This function retrieves the non-compliant cell counts at the connection level. The counts are incremented as a result of cell rate policing by the ATLAS device. The programmable 16 bit counts can be programmed as follows: Non-compliant CLP0 cells, Non-compliant CLP0+1 cells, Tagged CLP0 cells, Discarded CLP0 cells, Discarded CLP0+1 cells. The count type can be configured by calling API function vcsSetNcCntCfgs ().

Valid States `VCS_VCS_STANDBY, VCS_ACTIVE`

Side Effects None

Prototype `INT4 vcsGetStatVcNcCnts(UINT2 u2ICI, sVCS_VC_STAT_CNT *pNcCnt1, sVCS_VC_STAT_CNT *pNcCnt2, sVCS_VC_STAT_CNT *pNcCnt3)`

Inputs `u2ICI` : connection ID

Outputs `pNcCnt1` : contains count type and count for non-compliant counter 1.

`pNcCnt2` : contains count type and count for non-compliant counter 2.

`pNcCnt3` : contains count type and count for non-compliant counter 3..

Return Codes VCS_SUCCESS
VCS_ERR_INVALID_CONNID (invalid connection ID)
VCS_ERR_INVALID_STATE
VCS_ERR_CONN_MP_ORIGIN

vcsResetVcRxNcCnts

This function resets the Rx and non-compliant cell counts at the connection level.

Valid States VCS_VCS_STANDBY, VCS_ACTIVE

Side Effects None

Prototype INT4 vcsResetVcRxNcCnts(UINT2 u2ConnId)

Inputs u2ICI : connection ID, whose connection must be originated from Loop or WAN port.

Outputs None

Return Codes VCS_SUCCESS
VCS_ERR_INVALID_CONNID (invalid connection ID)
VCS_ERR_INVALID_STATE
VCS_ERR_CONN_MP_ORIGIN

Cell Counts Per Port

vcsGetStatPortCnts

This function retrieves counts of VC cells which are transmitted or received through a specified Loop/WAN port. The counts are the sum of VC cell counts over the connections associated with the port.

For the Rx count, the cell type being counted is determined by the count configurations set in `vcsSetRxCntCfg` API. It is suggested that the configurations be set the same for all connections within the port.

Valid States `STANDBY, VCS_ACTIVE`

Side Effects None

Prototype
```
INT4 vcsGetStatPortCnts(sVCS_PORT_ID sPortId,
UINT4 *pu4PortTxCnt, UINT4 *pu4PortRxCnt)
```

Inputs

`sPortId`	: Loop or WAN Port ID

Outputs

`pu4PortTxCnt`	: count of cells transmitted to the port.
`pu4PortRxCnt`	: count of cells received from the port.

Return Codes

`VCS_SUCCESS`	
`VCS_ERR_INVALID_PORTID`	(invalid port ID)
`VCS_ERR_INVALID_STATE`	
`VCS_ERR_CONN_MP_ORIGIN`	
`VCS_ERR_PORT_NOT_READY`	
`VCS_ERR_PORT_NOT_CFG`	

Cell Counts Per Line/WAN card

vcsGetStatCardCnts

This function retrieves counts of cells which are transmitted or received through a specified Line or WAN card. The count is maintained by the VORTEX or DUPLEX devices on the chipset card.

Valid States `VCS_STANDBY, VCS_ACTIVE`

Side Effects None

Prototype
```
INT4 vcsGetStatCardCnts(UINT1 u1CardId, UINT4
*pu4CardTxCnt, UINT4 *pu4CardRxCnt)
```

Inputs `u1CardID` : specifies a remote card by the HSS link, which connects the remote card to.the core card.
bit 0-2: HSS link number (0 to 7 for VORTEX)
(0 to 1 for DUPLEX)
bit 3-6: device number
bit 7 : 0=VORTEX (Line card),
1 = DUPLEX (WAN card)

Outputs `pu4CardTxCnt` : count of cells transmitted to the card.

`pu4CardRxCnt` : count of cells received from the card.

Return Codes
`VCS_SUCCESS`
`VCS_ERR_INVALID_CARD_ID` (invalid card ID)
`VCS_ERR_INVALID_STATE` (chipset is not in a valid state)
`VCS_ERR_NO_ACTIVE_CARD`

Counts Per Chipset

vcsGetStatDiscardCnts

This function is used to retrieve the discard/error counts accumulated by the APEX device. These counts include number of CLP0 and CLP1 cells discarded due to congestion as well as number cells discarded for reasons other than congestion.

This function can be used to maintain a steady count of the types mentioned by invoking it periodically.

Valid States `VCS_STANDBY, VCS_ACTIVE`

Side Effects None

Prototype
```
INT4 vcsGetStatDiscardCnts(VCS vcs,UINT4
     *pu4DiscardCnt UINT4 *pu4Clp0DiscardCnt,
     UINT4 *pu4Clp1DiscardCnt)
```

Inputs `vcs` : chipset handle

Outputs `pu4DiscardCnt` : general discard count of all cells that have been discarded due to reasons other than congestion (i.e., re-assembly timeout, re-assembly max. length error etc.)

`pu4Clp0DiscardCnt` : count of all CLP0 cells discarded due to congestion.

`pu4Clp1DiscardCnt` : count of all CLP1 cells discarded due to congestion.

Return Codes `VCS_SUCCESS`
`VCS_ERR_INVALID_HANDLE` (invalid chipset handle)
`VCS_ERR_INVALID_STATE` (chipset is not in a valid state)

vcsGetStatEventCnts

This function is used to retrieve the event counts accumulated and maintained by the underlying device drivers.

Note: for the current version, only DUPLEX and VORTEX device drivers provide such statistical information. The event counts are reset upon the retrieval.

Valid States `VCS_PRESENT, VCS_STANDBY, VCS_ACTIVE`

Side Effects None

Prototype `INT4 vcsGetStatEventCnts(VCS vcs, sVCS_DEVICE_ID sDevId, sVCS_STAT_CNT *psVcsStatCnt)`

Inputs `vcs` : chipset handle

`sDevID` : specifies the device driver, whose statistical counts are to be retrieved. It can be one of VCS_DUPLEX or VCS_VORTEX

Outputs `psVcsStatCnt` : contains the statistic event counts. The counts are valid only if the function returns VCS_SUCCESS. Also, the only fields in this structure that are valid are those have been requested using the input parameter `sDevID`.

Return Codes `VCS_SUCCESS`
`VCS_ERR_INVALID_HANDLE` (invalid chipset handle)
`VCS_ERR_INVALID_STATE` (chipset is not in a valid state)
`VCS_ERR_DRIVER_NOT_SUPPORT`

8.21 Congestion counts & Status

The congestion counts are snapshots of the current congestion counts on a specified chipset card. They need not increase monotonically. The counts can be polled for real-time performance monitoring or status check of the chipset operation.

vcsGetCongDevCnt

This function returns the total number of cells available for buffering in the chipset device, i.e. APEX (`FreeCnt`).

Valid States `VCS_STANDBY, VCS_ACTIVE`

Side Effects None

Prototype `INT4 vcsGetCongDevCnt(VCS vcs, UINT4 *pu4Cnt)`

Inputs `vcs` : chipset handle

Outputs `pu4Cnt` : snapshot of `FreeCnt`.

Return Codes `VCS_SUCCESS`
`VCS_ERR_INVALID_HANDLE` (invalid chipset handle)
`VCS_ERR_INVALID_STATE` (chipset is not in a valid state)

vcsGetCongDirCnt

This function retrieves the count of all cells queued for all loop/WAN ports in APEX devie.

Valid States `VCS_STANDBY, VCS_ACTIVE`

Side Effects None

Prototype
```
INT4 vcsGetCongDirCnt(VCS vcs, UINT1 u1Dir, UINT4
*pu4Cnt)
```

Inputs

`vcs`	: chipset handle
`u1Dir`	: 0 – loop, 1- WAN

Outputs

`pu4Cnt`	: loop/WAN cells queue count

Return Codes

`VCS_SUCCESS`	
`VCS_ERR_INVALID_HANDLE`	(invalid chipset handle)
`VCS_ERR_INVALID_STATE`	(chipset is not in a valid state)

vcsGetCongPortCnt

This function retrieves the count of all cells queued for the specified port in APEX device.

Valid States `VCS_STANDBY, VCS_ACTIVE`

Side Effects None

Prototype
```
INT4 vcsGetCongPortCnt(VCS vcs, sVCS_PORT_ID
sPortId, UINT4 *pu4Cnt)
```

Inputs

`vcs`	: chipset handle
`sPortId`	: port type (loop, WAN, uP) and number

Outputs

`pu4Cnt`	: cells queued for this port

Return Codes

`VCS_SUCCESS`	
`VCS_ERR_INVALID_HANDLE`	(invalid chipset handle)
`VCS_ERR_INVALID_STATE`	(chipset is not in a valid state)
`VCS_ERR_INVALID_PORT_ID`	(port has not been configured)

vcsGetCongClassCnt

This function retrieves the count of all cells queued for the specified class in the APEX device.

Valid States `VCS_STANDBY, VCS_ACTIVE`

Side Effects None

Prototype
```
INT4 vcsGetCongClassCnt(VCS vcs, sVCS_PORT_ID
sPortId, UINT1 u1ClassNum, UINT4 *pu4Cnt)
```

Inputs

`vcs`	: chipset handle
`sPortId`	: port identifier.
`u1ClassNum`	: class Number (0 to 3)

Outputs `pu4Cnt` : cells queued for this class.

Return Codes

`VCS_SUCCESS`	
`VCS_ERR_INVALID_HANDLE`	(invalid chipset handle)
`VCS_ERR_INVALID_STATE`	(chipset is not in a valid state)
`VCS_ERR_INVALID_PORT_ID`	(invalid port ID)
`VCS_ERR_INVALID_CLASS_ID`	(class not configured)

vcsGetCongConnCnts

This function retrieves the following counts at a VC level

- all CLP0 cells in both VC and class queue (`VcCLP0Cnt`)
- all CLP01 cells in VC queue (`VcQCLP01Cnt`)
- all CLP01 cells in class queue (`VcClassQCLP01Cnt`)

The counts are maintained by the APEX device.

Valid States `VCS_STANDBY, VCS_ACTIVE`

Side Effects None

Prototype
```
INT4 vcsGetCongConnCnts(UINT4 u4ICI, UINT4
*pu4VcClp0Cnt, UINT4 *pu4VcQClp01Cnt, UINT4
*pu4VcClassQClp01Cnt)
```

Inputs `u4ICI` : connection ID.

Outputs `pu4VcClp0Cnt` : snapshot of `VcCLP0Cnt`

`pu4VcQClp01Cnt` : snapshot of `VcQCLP01Cnt`

`pu4VcClassQClp01Cnt`: snapshot of `VcClassQCLP01Cnt`

Return Codes

`VCS_SUCCESS`	
`VCS_ERR_INVALID_HANDLE`	(invalid chipset handle)
`VCS_ERR_INVALID_STATE`	(chipset is not in a valid state)
`VCS_ERR_INVALID_CONNID`	(connection not configured)

vcsGetLastDiscardICI

This function retrieves the connection Ids of the last time a CLP0 or a CLP1 cell was discarded due to congestion

Valid States `VCS_STANDBY, VCS_ACTIVE`

Side Effects None

Prototype
```
INT4 vcsGetLastDiscardICI(VCS vcs, UINT4
*pu4Clp0ConnId,   UINT4 *pu4Clp1ConnId)
```

Inputs `vcs` : chipset handle.

Outputs `pu4Clp0ConnId` : connection ID of last time a CLP0 cell was discarded due to congestion

`pu4Clp1ConnId` : connection ID of last time a CLP cell was discarded due to congestion.

Return Codes

`VCS_SUCCESS`	
`VCS_ERR_INVALID_HANDLE`	(invalid chipset handle)
`VCS_ERR_INVALID_STATE`	(chipset is not in a valid state)

8.22 Callback Functions

The chipset driver uses the following indication routines to notify the applications of events within the chipset devices and chipset driver. These routines need to be implemented by the user.

Microprocessor Data Connection callbacks

indRxDataCell

This callback is invoked by the Microprocessor Data Connection Rx task after it extracts a cell from the microprocessor interface.

Prototype `VOID indRxDataCell(UINT4 u4ECI, sVCS_CELL_HDR *psHdr, UINT1 *pu1Pyld, INT4 result)`

Inputs

`u4ECI` : ID of the connection on which cell was received.

`psHdr` : header of the transmitted cell.

`psPyld` : payload of the transmitted cell.

`result` : result of cell Rx

Outputs None

Return Codes None

IndRxDataFrm

This callback is invoked by the Microprocessor Data Connection Rx task after it extracts an AAL5 frame from the microprocessor interface. A pointer to the first byte of the AAL5 frame buffer chain, the header of the last cell in the payload and the connection ID is passed to the user.

Prototype `VOID indRxDataFrm(UINT4 u4ECI, sVCS_CELL_HDR *psHdr, UINT1 *pu1Frm, UINT4 u4Len, INT4 result)`

Inputs

`u4ECI` : ID of the connection on which frame was received.

`psHdr` : header of the last cell in frame.

`pu1Frm`: payload of the frame (buffer chain)

`u4Len` : length of the frame in bytes

`result` : result of frame Rx

Outputs None

Return Codes None

Inband Control Channel Callbacks

indRxCtrlMsg

This callback is invoked by the Inband Control Channel Rx task after it receives a message from a remote card over an existing control channel. Note this function should be re-entrant, as both ICC Rx task and APEX SAR Rx task could call the function at the same time.

Prototype `INT4 indRxCtrlMsg (UINT2 ChnlID,INT4 result)`

Inputs `ChnlID` : channel ID

`result` : result of message Rx

Outputs None

Return Codes None

OAM Callbacks

indRxOAM

This callback is invoked by the Rx task of underlying ATLAS device driver after it receives an OAM cell from its microprocessor port.

Prototype `void indRxOAM(VCS_USR_CTXT vcsUsrCtxt, sVCS_DEV_ID *psDevId,INT4 u4OamType, UINT1 u1CmdFlag, INT4 arg1, INT4 connId, INT4 result)`

Inputs

`vcsUsrCtxt` : ser context, which is passed in from User when calling `vcsAdd` API.

`psDevId` : contains Device ID

`u4OamType` : type of OAM cell received

`u1CmdFlag` : flag to indicates a COMMAND or RESPONSE

`arg1` : additional OAM cell info. It contains actual message ID in the case of OAM cell type "Activation/Deactivation"

`connId` : connection ID of the OAM cell received

`result` : result of OAM processing by ATLAS device driver

Outputs None

Return Codes None

indCosStatus

This callback is invoked when a valid Change of (alarm) Status extracted from ATLAS Ingress COS FIFO. The function is called from within a watchdog task "`sysVcsWdgPtrlTaskFn`".

Prototype `void indCosStatus(UINT4 u4VcId, UINT2 u2Status)`

Inputs

`u4VcId` : connection ID, whose ingress VC has a valid COS status

`u2Status` : contains COS status bits (bits 0-9)

Outputs None

Return Codes None

Event Callbacks

indRxBOC

This callback is invoked by the DPR task of underlying VORTEX or DUPLEX device driver after it receives a BOC signal from a remote card.

Prototype
```
INT4 indRxBOC(VCS_USR_CTXT usrCtxt, sVCS_DEV_ID
*psDevId, UINT1 u1HssLnk, UINT1 u1BOCcode)
```

Inputs

`vcsUsrCtxt` : User Context, which is passed in from User when celling vcsAdd() API.

`psDevId` : contains Device ID

`u1HssLnk` : HSS link number

`u1BOCcode` : contains a BOC code received

Outputs None

Return Codes None

indVcsCritical

This indication callback function is called by `apexHiDPR` which executes in the context of the DPR task. The DPR taskis spawned by the underlying device driver. They provide the user with device ID, event ID and other supplemental arguments.

Prototype
```
VOID indVcsCritical(VCS_USR_CTXT usrCtxt,
sVCS_DEV_ID *psDeviceID, UINT4 u4EventId, UINT4
arg1, UINT4 arg2, UINT4 arg3)
```

Inputs

`usrCtxt` : user's context for the chipset.

`psDeviceID` : contains device type and device number

`u4EventId` : event ID

`arg1, arg2, arg3` : supplemental information

Outputs None

Return Codes None

indVcsError

This indication callback function is called by `apexLoDPR`, `atlasDPR`, `vortexDPR`, or `dpxDPR`, which execute in the context of the DPR tasks. The DPR tasks are spawned by the underlying APEX, ATLAS, VORTEX and DUPLEX device drivers. They provide the user with an event ID, device identifier ID and other supplemental arguments.

Prototype
```
VOID indVcsError(VCS_USR_CTXT usrCtxt,
sVCS_DEV_ID *psDeviceID, UINT4 u4EventId, UINT4
arg1, UINT4 arg2, UINT4 arg3)
```

Inputs

`usrCtxt`	: user's context for the device.
`psDeviceID`	: contains device type and device number
`u4EventId`	: event ID
`arg1, arg2, arg3`	: supplemental information

Outputs None

Return Codes None

9 SYSTEM-SPECIFIC UTILITY FUNCTIONS

These utility functions are normally called by the chipset driver APIs, and therefore should be considered as internal library functions. However, USER may port them with a different, system-dependent approach.

9.1 Congestion Control Service

The following routines can be used to calculate congestion threshold levels for QOS classes defined by TM4.0 (CBR, RT-VBR, non-RT VBR, GFR and UBR). Please see Appendix A for a detailed description of the implementation. USER may implement them with a new algorithm for the congestion control.

sysVcsPortThresholds

This routine is invoked when a new port is configured or an existing port is being deleted. It determines the per-port threshold levels and associated per-class thresholds for the port which is currently being configured or deleted. Besides this, it will also recalculate the port thresholds and class thresholds for all the ports, which are in the same direction as the port being configured or deleted. For example, all the loop port and associated class thresholds are re-configured if the routine is invoked for a loop port.

Valid States Not Applicable

Side Effects None

Prototype

```
INT4 sysVcsPortThresholds(sVCS_PORT_ID *psPortId,
sVCS_PORT_THRSH_REQUEST *psPortThrshReq, UINT1
forceFlag, sVCS_PORT_THRSH *psPortClThrsh )
```

Inputs

`psPortId` : specify a Loop or WAN port.

`ForceFlag` : when set to 1, the threshold values are set from the `psPortThrshReq` structure without any change.

When set to 0, only the `minPortThrsh` value is taken from the `psPortThrshReq`. The other threshold values are calculated by the routine.

`psPortThrshReq`: this structure contains the minimum port threshold for the port. If the forceFlag is set, this structure should also contain the clp0 threshold, the clp1 threshold and the max threshold for the port.

Outputs `psPortClThrsh`: pointer to the array of `sVCS_PORT_THRSH` data structure buffer, which contains the per-port and per-class thresholds for all ports in the direction of the port being configured or deleted.

Return Codes
`VCS_SUCCESS`
`VCS_ERR_DIR_MAX_THRSH_0`
`VCS_ERR_INVALID_THRSH`
`VCS_ERR_PRT_BW_GUARANTEE`

sysVcsVcThresholds

This routine is invoked when a connection is being configured, or the QOS of an existing connection is being updated or the connection is being deleted. It determines the per-VC threshold levels based on the traffic type and QOS requested for the VC. Besides this the routine will also recalculate the thresholds of all the connections which are in the same port as the connection being configured, updated or deleted. The thresholds for all the classes within this port are also recalculated.

Valid States Not Applicable

Side Effects None

Prototype
```
INT4 sysVcsVcThresholds(sVCS_PORT_ID* psPortId,
UINT2 u2ConnId, sVCS_VC_QOS* psNewVcQOS,
sVCS_PORT_THRSH* psPortThrsh, sVCS_VC_THRSH*
psVcThrsh)
```

Inputs
`psPortId` : specifies the port on which connection is to be configured
u2ConnId : specifies the connection id if connection already configured
psVcQos : pointer to the QOS parameters structure for the connection

Outputs
`psPortThrsh` : contains the port thresholds and updated class thresholds
`psVcThrsh` : pointer to an array of structures containing the per-VC threshold levels of all the connections on the port on which the connection is being configured, updated or deleted.

Return Codes
`VCS_SUCCESS`
`VCS_ERR_VC_BW_GUARANTEE`
`VCS_ERR_OUT_OF_RESOURCE` (out of resources)

9.2 Scheduling Service

The following routines can be used to calculate scheduling parameters for Class of Service defined by TM4.0 (CBR, RT-VBR, non-RT VBR, GFR and UBR). Please see Appendix A for a detailed description of the implementation. USER may implement them with a new scheduling algorithm.

sysVcsLoopPortScheduler

Determines the per-port polling weight based on the minimum cell rate or bandwidth requested for the loop port. This routine is invoked when a loop port is being configured

Valid States Not Applicable

Side Effects None

Prototype
```
INT4 sysVcsLoopPortScheduler (UINT4u4OutCellRate,
UINT1 *pu1Weight)
```

Inputs `u4OutCellRate`: minimum cell rate for the cells transmitted out to the Loop port.

Outputs `pu1Weight` : polling weight, between 0 to 7.

Return Codes
```
VCS_SUCCESS
VCS_ERR_INVALID_CELL_RATE
```

sysVcsWANPortScheduler

Determines the per-port polling weight based on the minimum cell rate or bandwidth requested for the WAN port. . This routine is invoked when a WAN port is being configured.

Valid States Not applicable

Side Effects None

Prototype
```
INT4 sysVcsWANPortScheduler (UINT4 u4OutCellRate,
UINT1 *pu1Weight)
```

Inputs `u4OutCellRate` : minimum cell rate limit for the cells transmitted out to the WAN port.

Outputs `pu1Weight` : polling weight, between 0 to 3.

Return Codes `VCS_SUCCESS`

VCS_ERR_INVALID_CELL_RATE

sysVcsClassVcScheduler

Determines the Class scheduler parameters based on the traffic type and QOS parameters of the connections in each class. This routine is invoked each time a new connection is configured or the QOS of the connection is updated. The routine will also return the connection weight if the connection is not a shaped connection or a frame connection.

Valid States Not Applicable

Side Effects None

Prototype
```
INT4 sysVcsVcClassScheduler(sVCS_VC_PORT_ID
*psPortId, UINT2 u2ConnId, sVCS_VC_QOS *psVcQos,
sVCS_CLASS_SCHEDULER *psClassSch, UINT1 *pu1Wt)
```

Inputs

psPortId	: specifies port on which connection is configured
u2ConnId	: Connection ID
psVcQOS	: QOS parameters for the connection .

Outputs

psClassSch	: contains the Class scheduler parameters.
pu1Wt	: pointer to weight for the connection

Return Codes VCS_SUCCESS

9.3 Shaping Service

The following routines can be used to calculate shaping parameters based on ATMF TM4.1 parameters or QOS request. USER may implement them with a new shaping algorithm.

sysVcsVCShaping

Determines the shaped single rate parameters based on the QOS parameters and the shaper configuration. This routine is invoked when a shaped connection is configured or its QOS parameters are updated.

Valid States Not Applicable

Side Effects None

Prototype `INT4 sysVcsVCShaping(UINT1 u1ShprId, sVCS_VC_QOS *psVcQos, sVCS_VC_SHPR *psVcShpr)`

Inputs `u1ShprId` : shaper used by the connection

`psVcQos` : contains QOS parameters

Outputs `psVcShpr` : contains the per-VC shaper rate context.

Return Codes `VCS_SUCCESS`
`VCS_ERR_SHPR_PARAMETER`

9.4 Policing Service

The following routines can be used to calculate ATLAS policing parameters based on QOS request. USER may implement them with a new policing algorithm.

sysVcsVcPolicing

Determines the per_VC Policing parameters (Increment field and Limit field) based on QOS request.

Valid States Not Applicable

Side Effects None

Prototype
```
VOID sysVcsVcPolicing(sVCS_VC_QOS *psVcQos,
sVCS_VC_POLICING  *pPolicing1, sVCS_VC_POLICING
*pPolicing2)
```

Inputs
`psVcQos` : contains QOS parameters

Outputs
`pPolicing1` : containing the Increment and Limit fields for GCRA1

`pPolicing12` : containing the Increment and Limit fields for GCRA2

Return Codes `None`

9.5 Port Mapping

The following routines can be used to map the HSS link numbers of VORTEX devices to loop port numbers. Note that we have to reserve some port numbers for use in Control Channels.

sysVcsLoopIdToPort

Determines a port number based on Loop IDs (VORTEX device ID, HSS link ID and xdsl ID). The mapping algorithm is system-specific, and not necessarily limited to the one implemented here.

Valid States Not Applicable

Side Effects None

Prototype
```
VOID sysVcsLoopIdToPort(UINT1 u1VtxId, UINT1
u1LinkId, UINT1 u1xdslId, UINT2 *pu2PortNum)
```

Inputs

`u1VtxId` : VORTEX device ID, (from to (VCS_MAX_VORTEXS-1))

`u1LinkId` : HSS link ID of the VORTEX device (from 0 to 7)

`u1xdslId` : logical channel ID within the DUPLEX of a line card which is connected to the VORTEX device via the HSS link. (from 0 to 32). However, if u1LinkId == VCS_CHNL_LINK_ID, u1xdslId shall contains the link number (from 0 to (VCS_CHNL_XDSL_ID - 1)).

Output `pu2PortNum` : Loop Port number

Return Codes None

sysVcsPortToLoopId

Determines Loop IDs (VORTEX device ID, HSS link ID and xdsl ID) based on a port number. The mapping algorithm is system-specific, and not necessarily limited to the one implemented here.

Valid States Not Applicable

Side Effects None

Prototype `VOID sysVcsPortToLoopId(UINT2 u2PortNum, UINT1 *pVtxId, UINT1 *pLinkId, UINT1 *pxdslId)`

Inputs `pu2PortNum` : Loop Port number

Outputs `pVtxId` : contains VORTEX device ID (from 0 to (VCS_MAX_VORTEXS-1))

`pLinkId` : HSS link ID of the VORTEX device (from 0 to 7)

`pxdslId` : Logical channel ID within the DUPLEX of a line card which is connected to the VORTEX device via the HSS link. (from 0 to 32). However, if u1LinkId == VCS_CHNL_LINK_ID, u1xdslId shall contains the link number (from 0 to (VCS_CHNL_XDSL_ID - 1)).

Return Codes None

sysVcsChnlIdToPort

Determines a reserved port number for control channels to a line card which is connected to a specified HSS link of a VORTEX device. The mapping algorithm is system-specific, and not necessarily limited to the one implemented here.

Valid States Not Applicable

Side Effects None

Prototype `VOID sysVcsChnlIdToPort(UINT1 u1VtxId, UINT1 u1LinkId, UINT2 *pu2ChnlPortNum)`

Inputs `u1VtxId` : VORTEX device ID, (from 0 to `(VCS_MAX_VORTEXS-1)`)

`u1LinkId` : HSS link ID of the VORTEX device (from 0 to 7)

Outputs `pu2ChnlPortNum` : reserved Loop Port number for control channels

Return Codes None

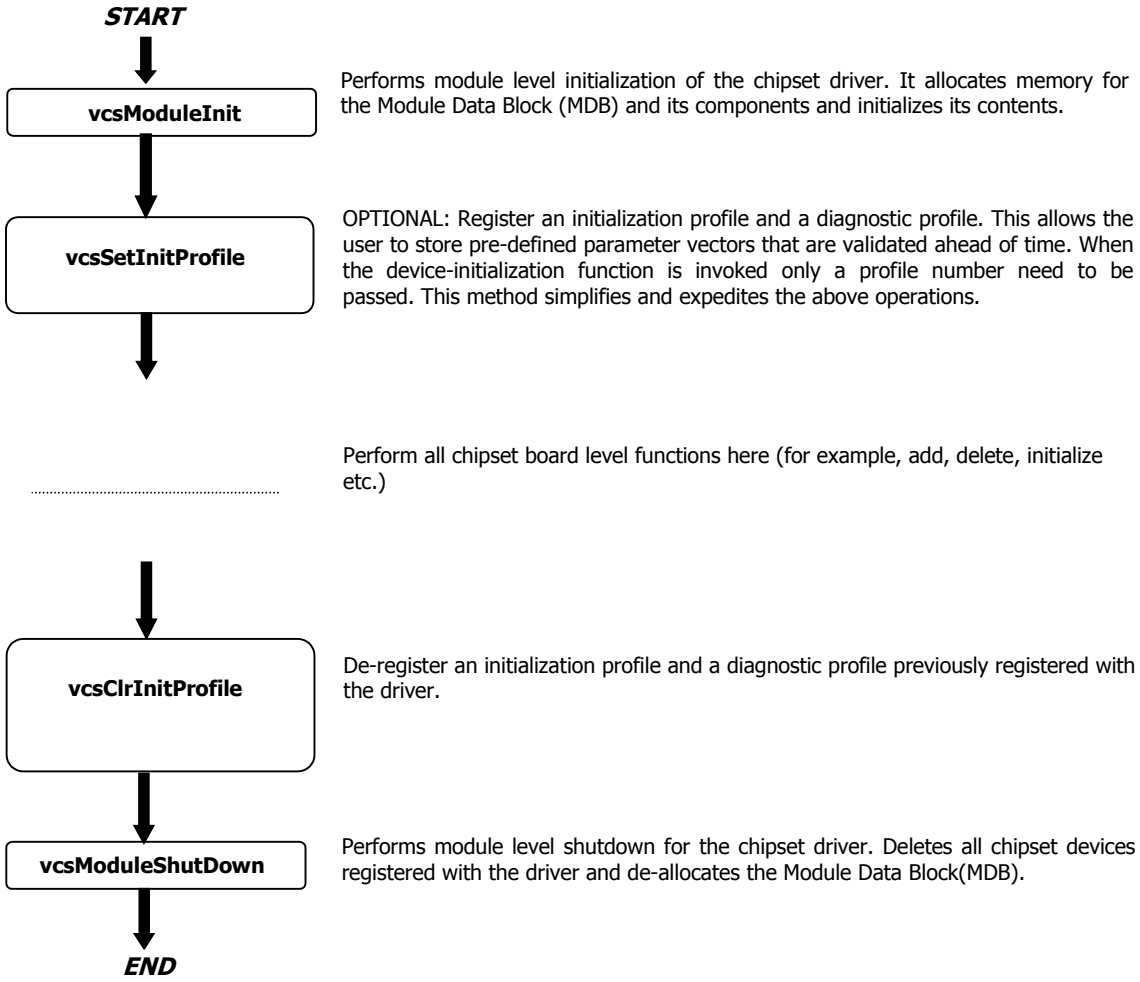
10 THEORY OF OPERATIONS

This section provides some of the implementation details of the VORTEX chipset driver modules. The implementation details include processing flows, data structures and algorithm descriptions where applicable.

10.1 Module Management

The following flow diagram illustrates the typical function call sequences that occur when initializing or shutting down the VORTEX chipset driver module.

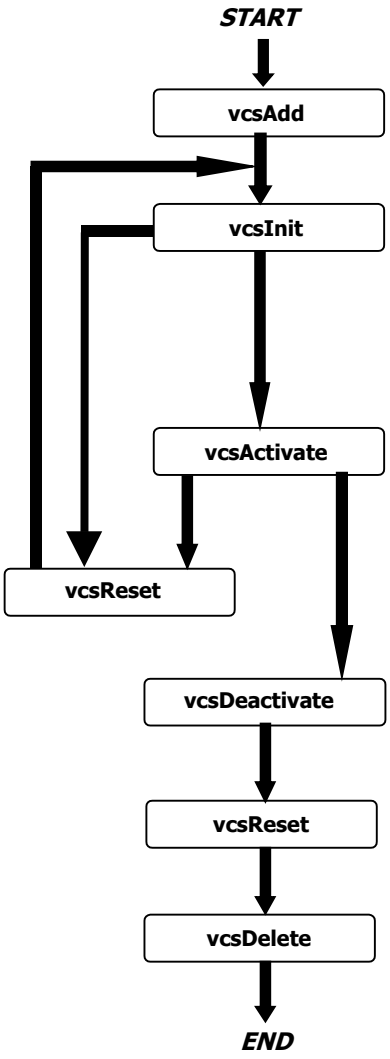
Figure 5: Module Management Flow Diagram



10.2 Chipset Management

Figure 6 illustrates the typical function call sequences that occur when adding, initializing, re-initializing and deleting chipset (card).

Figure 6: Chipset Management Flow Diagram



Detects the chipset being added in the hardware (using *sysVcsBoardDetect*), performs a chipset device check, assigns a chipset handle for storing chipset information and applies a software reset to the chipset.

Initializes the chipset (devices) based on an initialization vector or initialization vector profile provided by the user. The initialization vector is validated by the user and stored by the driver as part of device context information. The chipset registers are then configured accordingly.

Prepares the chipset for normal operation by installing and enabling interrupts, enabling the tx/rx of cells and frames from the microprocessor ports and enabling the APEX queue engine's external interfaces and HSS links of VORTEX and DUPLEX per VC connection request. The chipset is now operational and all other API can be

vcsReset performs a software reset on the device. It also resets the Chipset Data Block (CDB) contents except for the initialization vector. This function can be invoked from any chipset state.

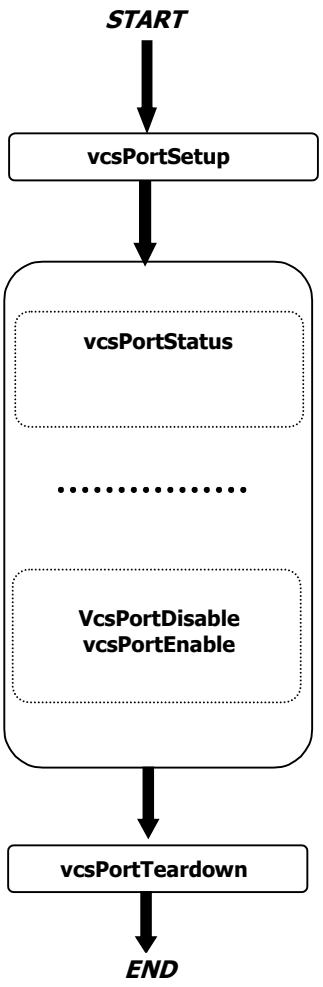
De-activates the chipset and removes it from normal operation. This function disables device interrupts, disables tx/rx of cells/frames from microprocessor port and disables the queue engine's external interfaces.

Removes the chipset from the list of chipsets being controlled by the driver. This function clears the Chipset Data Block for the chipset being deleted and frees the chipset handle assigned for this chipset.

10.3 Port Management

The following flow diagram illustrates the typical function call sequences to setup, and teardown a port.

Figure 7: Port Management Flow Diagram



The chipset must be in ACTIVE state when calling the following APIs.

Pre-requirement: the associated remote Line/WAN card should have already been added to the system by calling vcsAddCard().

Setup a new port, and allocates necessary resources for the port.

OPTIONAL: The APIs can be used to retrieve the current port status, as well as the number of VC connections associated with the port.

OPTIONAL: setup some VC connections through the port, or teardowm any connections associated with the port. See the next section "Connection Management".

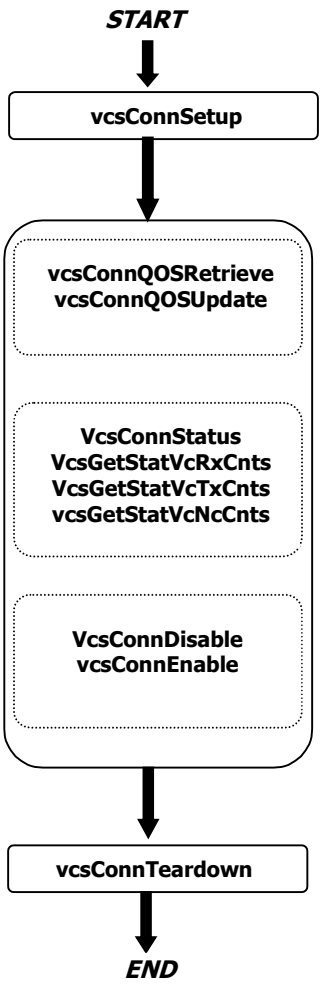
OPTIONAL: To temporary disable the port (no cells can pass through the port), or re-enable the disabled port.

Tear down a port and associated connections, and de-allocate the resource back to the system.

10.4 Connection Management

The following flow diagram illustrates the typical function call sequences to setup, update and teardown a VC connection.

Figure 8: Connection Management Flow Diagram



The chipset must be in ACTIVE state when calling the following APIs.

Pre-requirement: the associated ports should have already been setup.

Setup a new VC connection, and allocates necessary resources for the connections. A unique connection ID (from 0 to 65535) is returned after the connection is successfully established.

OPTIONAL: The APIs can be used to retrieve the current QOS settings of the VC connection, and update the connection with a new set of QOS parameters.

OPTIONAL: The APIs can be used to check the status of the connection, as well as the accumulated Rx and Tx cell count on the connection.

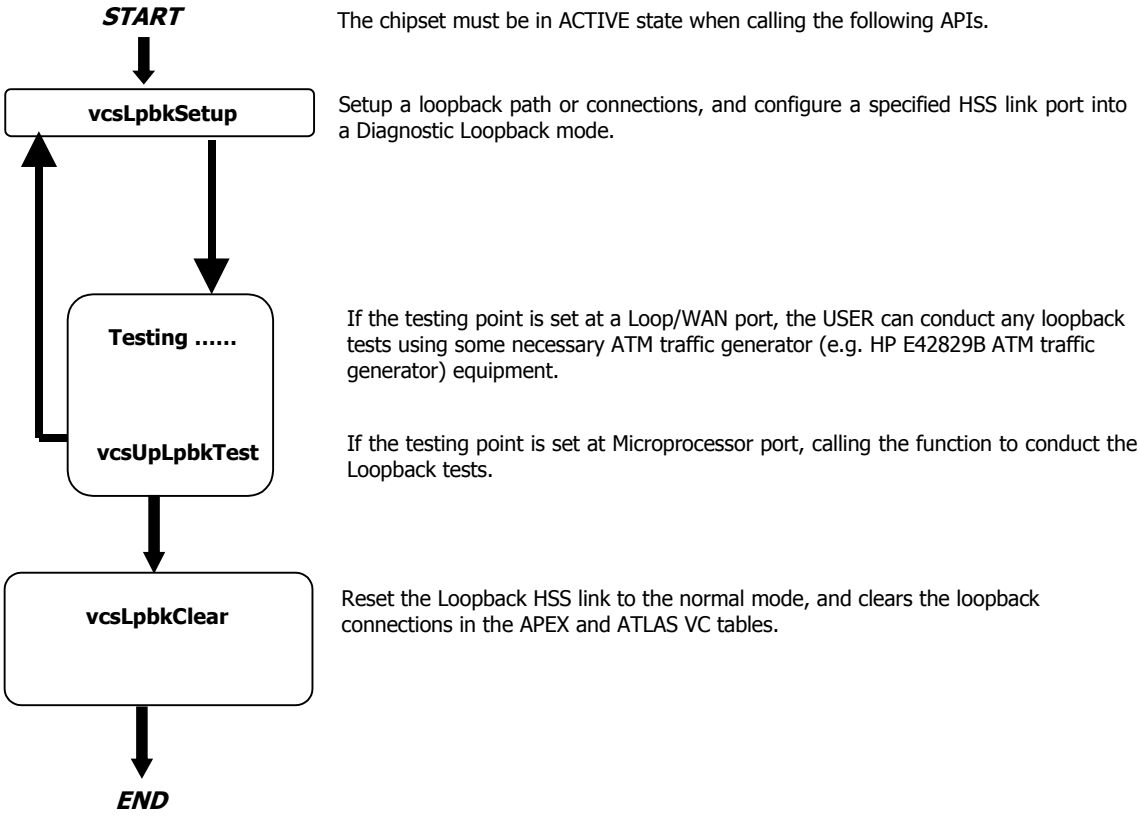
OPTIONAL: To temporary disable the receive and transmission of cells on a connection, or re-enable the disabled connection.

Tear down a connection, and de-allocate the resource back to the system.

10.5 Loopback Test

The following flow diagram illustrates the typical function call sequences to conduct the loopback tests of the chipset system.

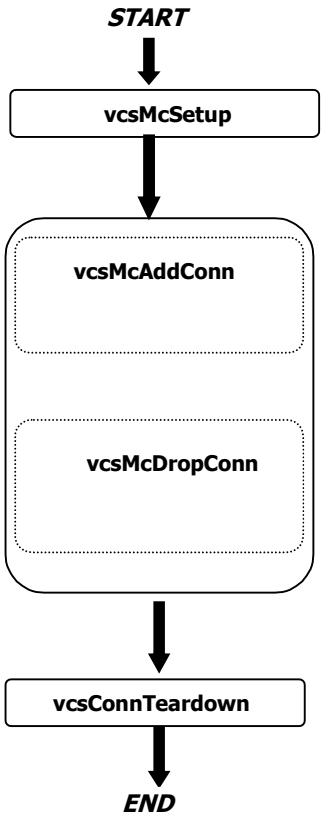
Figure 9: Loopback Test Flow Diagram



10.6 Multicast Support

The following flow diagram illustrates the typical function call sequences to setup, update and teardown a multicast group.

Figure 10: Multicast Support Flow Diagram



The chipset must be in ACTIVE state when calling the following APIs.

Setup a new multicast group, and allocates necessary resources for the the connection group. A unique multicasting connection handle is returned after the multicast group is successfully established.

OPTIONAL: The API can be used to add a VC connection to the existing multicast group.

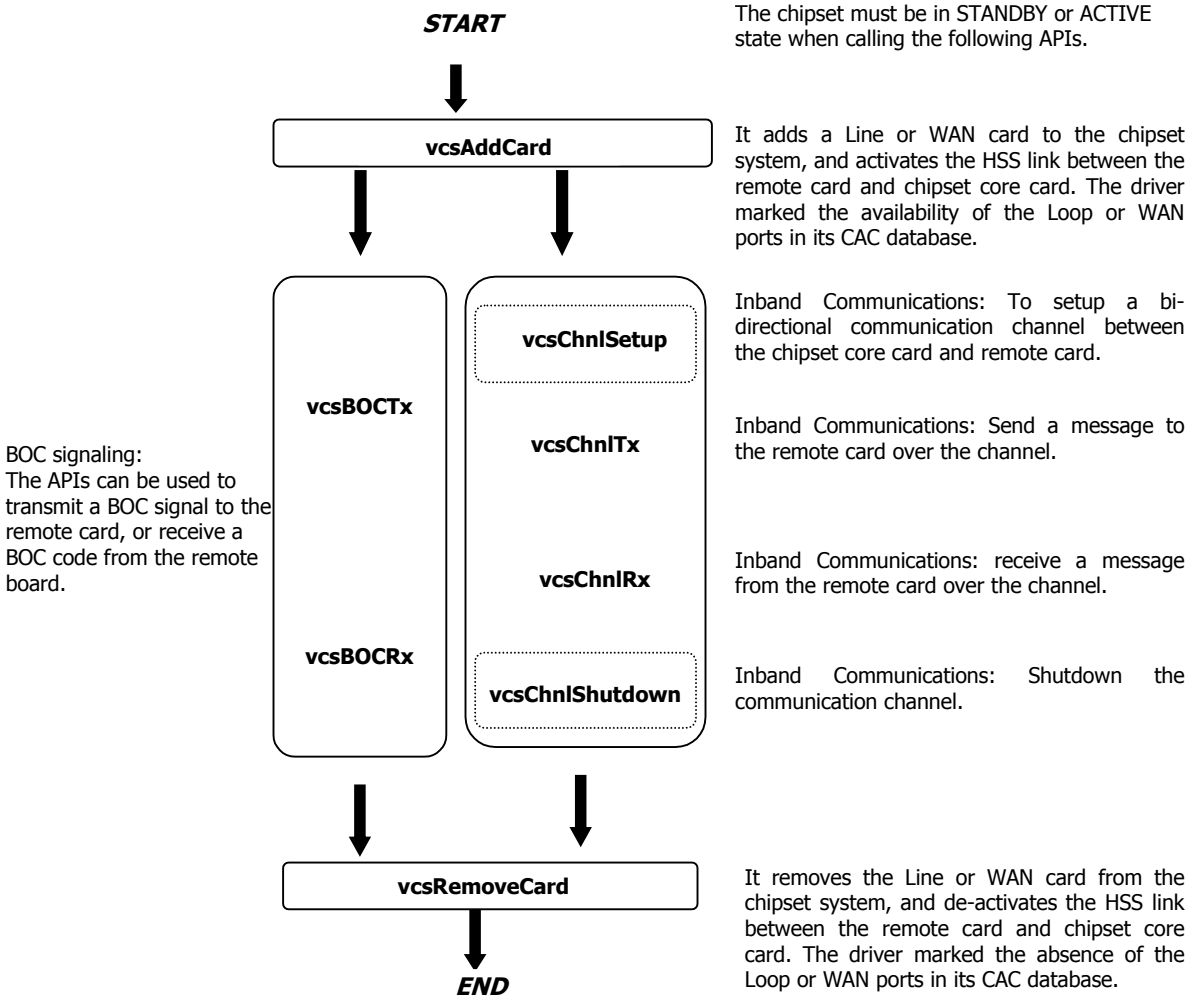
OPTIONAL: The API can be used to drop a VC connection from the existing multicast group.

Tear down the multicast group, and de-allocate the resource back to the system.

10.7 Line/WAN Card Management and Communication

The following flow diagram illustrates the typical function call sequences to add, remove and/or communicate with a remote Line/WAN card.

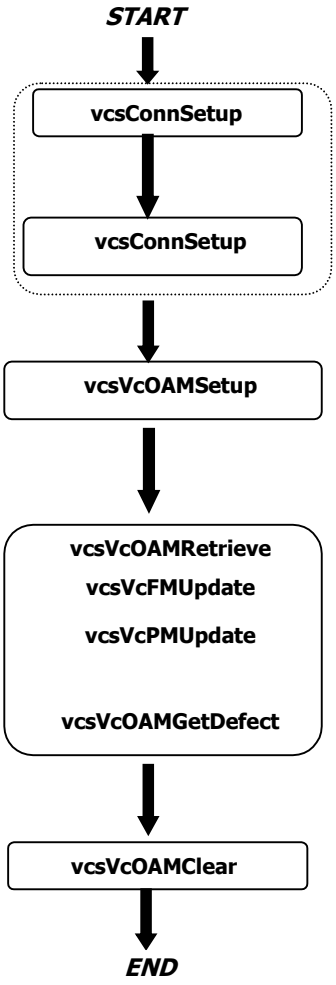
Figure 11: Line/WAN Card Management Flow Diagram



10.8 OAM Management

The following flow diagram illustrates the typical function call sequences to setup, update and clear an OAM configuration over a VC connection.

Figure 12: OAM Management Flow Diagram



The chipset must be in ACTIVE state when calling the following APIs.

Pre-requirement: the VC connection should be already setup by calling vcsConnSetup.

OPTIONAL: Setup another VC connection in a backward direction if a backward VC has not been established yet. OAM backward reporting cells will be routed to the backward path.

Setup and enables a F4 or F5 OAM Support over the connection. For the connection originated from WAN port, a shortcut backward path may be used, which can be setup directly in the ATLAS Egress VC table without passing through the APEX, as shown in Figure 21.

OPTIONAL: these three APIs can be used to retrieve the current setting of OAM configuration, and modify the OAM parameters for the VC.

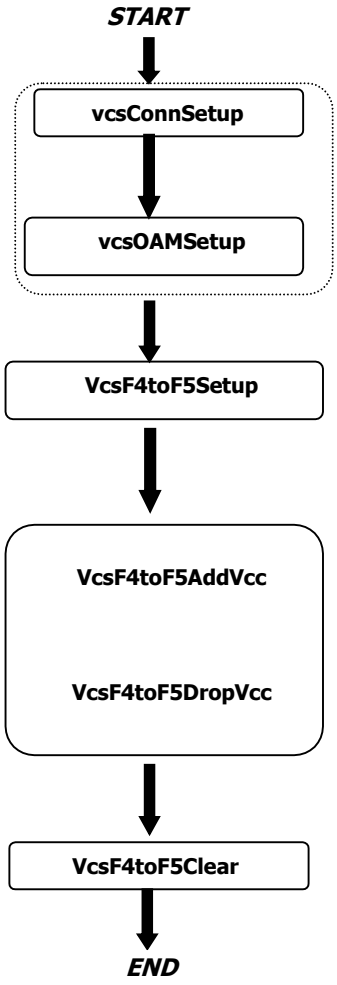
OPTIONAL: the API can be used to poll the received OAM defect type and locations.

Disable the F4 or F5 OAM support, and clear the OAM configuration. Therefore, all incoming OAM cells will be passed transparently through the chipset thereafter

10.9 F4 to F5 Processing

The following flow diagram illustrates the typical function call sequences to setup, update and clear a F4 to F5 Processing.

Figure 13: F4 to F5 Processing Flow Diagram



The chipset must be in ACTIVE state when calling the following APIs.

Pre-requirement 1: the F4 (VPC) connection and all constituent F5 (VCC) connections should have already been setup by calling vcsConnSetup.

Pre-requirement 2: the OAM support over the F4 (VPC) connection(s) should have already been setup by calling vcsOAMSetup. The OAM support over constituent F5 (VCC) connections might be setup too, if desired.

Setup and enables the F4 to F5 OAM Processing. It adds the constituent F5 connections to the processing list by linking the F4 VPC pointer into the F5 connections.

OPTIONAL: The API allows USER to dynamically add a VCC connection to the F4 to F5 Processing list.

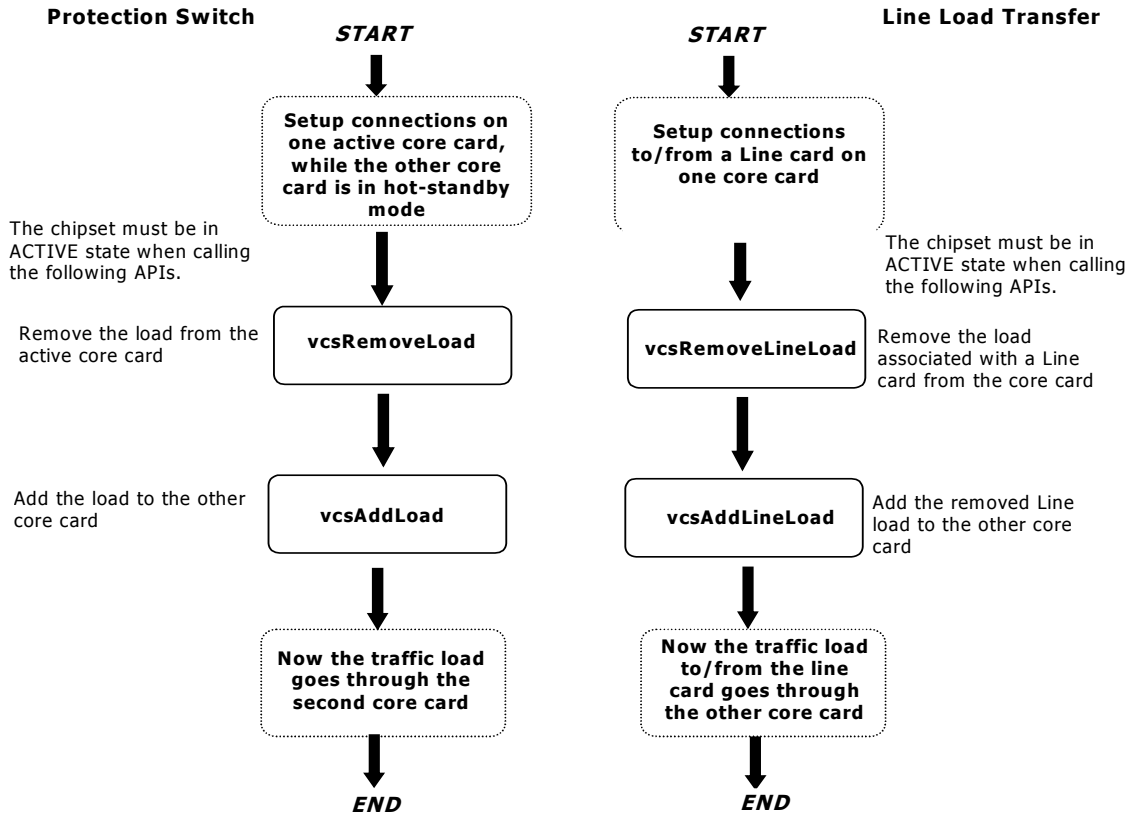
OPTIONAL: The API allows USER to dynamically drop a VCC connection from the F4 to F5 Processing list.

Disable the F4 to F5 OAM processing over the VPC OAM connection by removing the F4 VPC pointer from the constituent F5 connections.

10.10 Protection Switch and Line Load Transfer

The following flow diagram illustrates the typical function call sequences to conduct a protection switch between an active core card and a hot-standby core card, or transfer the connection load of a line card from one serving core card to the other core card.

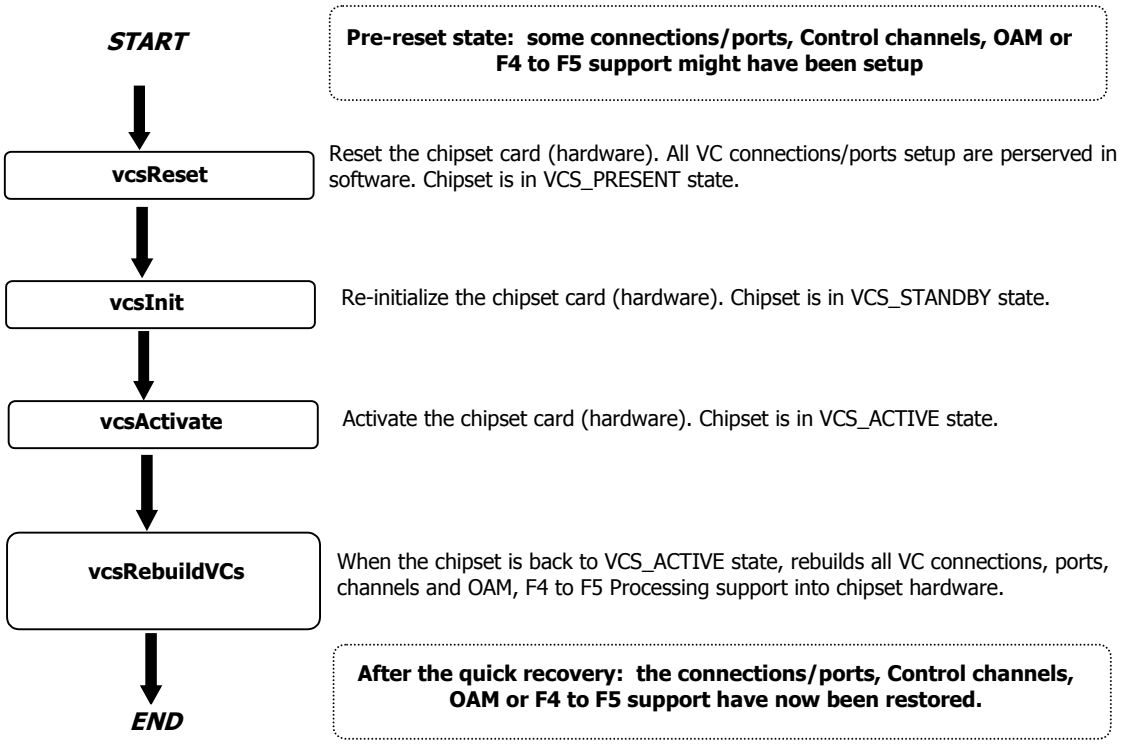
Figure 14: Protection Switch and Load Transfer Flow Diagram



10.11 Chipset Reset and Quick Recovery

Figure 6 illustrates the typical function call sequences that occurs for a quick recovering of all existing VC connections after reset of chipset (core card).

Figure 15: Chipset Reset and Quick Recovery Flow Diagram



10.12 Interrupt service module

Figure 16 illustrates the interrupt service model used in the chipset driver design. Note that the underlying device driver provides the service routines for each chipset device. This section gives an overview of the interrupt service model.

The interrupt service code includes some system specific code (routines prefixed by *sys*) that is typically implemented by the user for their system, as well as some system independent code (prefixed by *chipset device name*) provided by the device drivers that does not change from system to system.

The interrupt handle routines prefixed by *sys* (e.g. `sysApexHiIntHandler` and `sysApexLoIntHandler`) are system-specific, and shall be implemented by the user. They are installed in the interrupt vector table of the system processor. These routines are invoked when one or more chipset devices interrupt the processor.

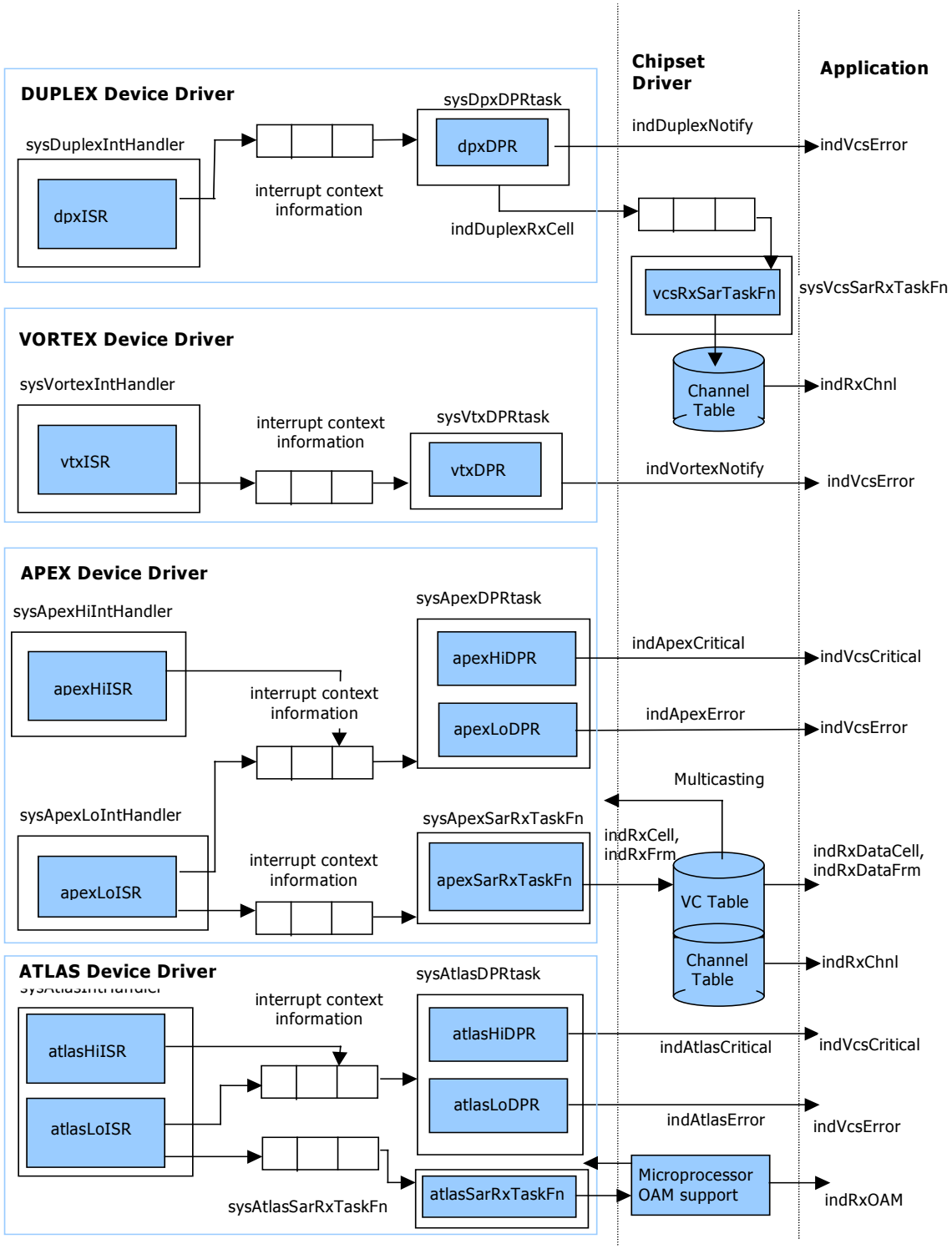
Interrupt servicing

When an interrupt occurs, `sysXXXHandler` (where XXX denotes a chipset device name, e.g. Apex, Atlas, Vortex, Duplex) invokes a device driver provided routine, `XXXISR`, for each device that has interrupt processing enabled. `XXXISR` reads the Interrupt Status register of the chipset device and returns with the status information if a valid error/status bit is set. This status information is then sent by `sysXXXHandler` selectively to one of two tasks – the SAR Receive tasks or the DPR task depending on the nature of the condition(s) detected.

`sysXXXSarRxTaskFn` are system-specific routines that run as separate tasks (SAR Rx tasks) within the RTOS. These tasks wait for messages, sent by `sysXXXIntHandler` (in APEX case) or `sysXXXDPRTask` (in VORTEX and DUPLEX cases), to arrive at their associated message queues. These messages correspond to arrival of cell(s) in the SAR TX Data register(s)

Once a message has been received by `sysXXXSarTaskFn`, it invokes the driver-provided routine, `xxxSarRxTaskFn`. The `xxxSarRxTaskFn` routine takes the appropriate actions based on the status information received in the message. Actions include extracting cells/frames from the SAR TX registers and reporting frame re-assembly timeouts or length errors to the application via indication callback functions. In the case of APEX Rx task, it multicasts the cell/frame to a list of destination VCs if the incoming cell/frame belongs to a multicasting connection. In the case of ATLAS Rx task, it processes the cell and may send out a backward reporting OAM cell if the received cell is of certain OAM types.

Figure 16: Interrupt Service Model



sysXXXDPRtask is another system-specific routine that runs as a separate task (DPR task) within the RTOS. This task also waits for messages, sent by sysXXXIntHandler, to arrive at an associated message queue. These messages correspond to interrupt conditions that are not SAR-related.

When a message is received, the driver-supplied function xxxDPR is invoked. This function updates the interrupt counters for the interrupt events causing the interrupt. If at least one event crosses its threshold, an indication callback is invoked. The input arguments passed to this indication function include the user's context for the device and an indication vector that consists of threshold crossing events. After processing all interrupt events, the DPR reads the interrupt registers again and performs the same operations if more bits are found to be set. Finally, when no more valid bits are set in the interrupt register, the DPR routine exits after enabling the low-priority error interrupt processing.

Note that the driver-provided routines, xxxISR, ,xxxSarRxTaskFn, and xxxDPR routines themselves do not specify a communication mechanism between the ISRs and tasks. Therefore the user is given full flexibility in choosing a communication mechanism between the two. The most common way to implement this communication mechanism is to use a message queue, a service that is provided by most RTOSes.

Installation and removal of interrupt handlers

The system specific routines, sysXXXIntHandler, and sysXXXDPRtask, are implemented by the user. sysXXXIntHandler is installed in the interrupt vector table of the processor using user-implemented routines, sysXXXIntInstallHandler. The sysXXXDPRtask is spawned as a task during the first time invocation of sysXXXIntInstallHandler. In addition, sysXXXIntInstallHandler also creates the communication channels between sysXXXIntHandler and sysXXXDPRtask. This communication channel is usually implemented as a message queue.

Similarly, during removal of interrupts, the sysXXXIntHandler and sysXXXIntHandler routines are removed from the microprocessor's interrupt vector table and the sysXXXDPRtask task is deleted. This code is implemented by the user in system specific functions sysXXXIntRemoveHandler.

11 PORTING GUIDE

This section outlines how to port the VORTEX chipset driver to your hardware and OS platform. However, this manual can offer only guidelines for porting the chipset driver because each platform and application is unique.

11.1 Driver Source Files

The C source files listed in Table 49 and Table 50 contain the code for the chipset driver. You may need to modify the code or develop additional code. The code is in the form of constants, macros, and functions. For ease of porting, the code is grouped into source files (`src`) and include files (`inc`). The src files contain the functions and the inc files contain the constants and macros. A makefile is also included. For all the underlying device driver (APEX, ATLAS, VORTEX and DUPLEX), please refer to their Device Driver User Manual for porting instructions.

Table 49: Chipset Driver Source Files

File	Description
`vcs_api1.c`	Chipset management API functions
`vcs_api2.c`	Connection/port management API functions
`vcs_api3.c`	Control Channel/Multicasting/OAM API functions
`vcs_hw.c`	Hardware interface functions
`vcs_rtos.c`	RTOS interface functions
`vcs_sys.c`	System-dependent utility functions
`vcs_ind.c`	Internal indication callback functions for underlying device drivers
`vcs_buf.c`	Buffer management for cell/frame Rx from uP
`vcs_queu.c`	Generic queue functions
`vcs_util.c`	Internal utility functions
`vcs_test.c`	Example implementation of callback and Chipset Initialization Vector functions

Table 50 : Chipset Driver Include Files

File	Description
vcs_api.h	API function prototypes, data structures, constants, and definitions
vcs_type.h	Variable type definitions
vcs_hw.h	Hardware interface constants and macro definitions
vcs_rtos.h	RTOS interface constants and macro definitions
vcs_sys.h	System-dependent constant and function prototype
vcs_err.h	Error codes returned by the chipset driver
vcs_buf.h	Prototypes of driver's internal functions
vcs_ind.h	Prototypes of driver's internal callback functions
vcs.h	Driver's internal data structures
vcs_queu.h	Data structures, prototypes of generic queue functions
vcs_test.h	Data structures, constants, and definitions used by sample code in vcs_test.c

11.2 Porting Procedure

The following procedures summarize how to port the chipset driver to your platform. The subsequent sections describe these procedures in more detail.

To port the chipset driver to your platform:

Step 1: Port the driver's hardware interface (page 182):

Step 2: Port the driver's OS extensions (page 183):

Step 3: Port the driver's system-dependent utility functions (page 185);

Step 4: Port the driver's application-specific elements (page 185):

Step 5: Build the driver (page 186).

Step 1: Porting the Hardware Interface

This section describes how to modify the chipset driver for your hardware platform.

To port the driver to your hardware platform:

1. Modify the variable type definitions in `vcs_type.h`.

2. Modify the low-level device read/write macros in the `vcs_hw.h file`. You may need to modify the raw read/write access macros (`sysVcsRawRead` and `sysVcsRawWrite`) to reflect the application's addressing logic.

3. Define the hardware system-configuration constants in the `vcs_hw.h` file. Modify the following constants to reflect the application's hardware configuration:

Device Constant	Description	Default
VCS_MAX_DEVS	The maximum number of chipset core cards to be controlled by the driver	2
VCS_MAX_VORTEXS	The number of VORTEX chips on a chipset core card	2
VCS_MAX_VCS	The maximum number of vc's to be supported by the chipset driver depends on the APEX SDRAM size	16K
VCS_MAX_LOOP_PORTS	The maximum number of Loop ports supported by the system	2K
VCS_MAX_WAN_PORTS	The maximum number of WAN ports supported by the system	4
VCS_MAX_CELL_RATE	The maximum traffic throughput in half duplex: in cells/second	1420K
VCS_MAX_CELL_RATE_PER_LOOP	The maximum traffic throughput per loop port: in cells/second	230K
VCS_SYSCLK_FREQ	APEX System clock frequency in Hz	80000000
VCS_SYSCLK_FREQ_ATLAS	ATLAS System clock frequency in Hz	50000000

Device Constant	Description	Default
VCS_MAX_NUM_CELL_BUFFS	The maximum cell buffer size in cells, determined by APEX SRAM size.	64K
VCS_APEX_MEM_OFFSET	Memory offset of APEX device	0x0000
VCS_ATLAS_MEM_OFFSET	Memory offset of ATLAS device	0x8000
VCS_DUPLEX_MEM_OFFSET	Memory offset of DUPLEX device	0x14000
VCS_VORTEX_MEM_OFFSET	Memory offset of first VORTEX device	0xC000
VCS_VORTEX_MEM_RANGE	memory range per VORTEX device	0x4000

4. Modify the sysVcsCardDetect function in vcs_hw.c as per your hardware environment. This function should output the base addresses of the chipset devices. This function also outputs a pointer to system-specific configuration information (for example, IRQ associated with the chipset device interrupt). This output parameter is simply stored by the driver in the CDB can be returned as NULL if not required by other system-specific functions.

Step 2: Porting the RTOS interface

The RTOS interface functions and macros consist of code that is RTOS dependent and needs to be modified as per your RTOS's characteristics.

To port the driver's RTOS interface:

1. Redefine the following macros in `vcs_rtos.h` to the corresponding system calls that your target system supports.

Service Type	Macro Name	Description
Memory	sysVcsMemAlloc	Allocates a memory block
	sysVcsMemFree	Frees a memory block
	sysVcsMemSet	Fills a memory block with a specified value
	sysVcsMemCpy	Copies the contents of one memory block to another

Service Type	Macro Name	Description
Buffer Management	sysVcsGetVcOAMBuff	Get a memory block to store VC OAM
	sysVcsFreeVcOAMBuff	Frees the VC OAM memory block
	sysVcsGetF4toF5Cb	Get a memory block to store F4 to F5 Control Block
	sysVcsFreeF4toF5Cb	Frees the F4 to F5 CB memory block
Semaphores	sysVcsSemCreate	Creates a mutual-exclusion semaphore
	sysVcsSemDelete	Destroys the specified semaphore
	sysVcsSemTake	Acquires the specified semaphore
	sysVcsSemGive	Relinquishes the specified semaphore

2. Modify other OS-specific Constant definition in `vcs_rtos.h`, such as stack size and task priority for the ICC Rx and Watchdog tasks.

3. Modify the system-specific interrupt handler, SAR processing and delay routines in `vcs_rtos.c`:

Service Type	Function Name	Description
ICC Rx task	sysVcsIccInstall	Spawns the ICC Rx task and associated message queues
	sysVcsIccRemove	Deletes the ICC Rx task and associated message queues
	sysVcsIccRxTaskFn	This function is executed in the context of the ICC Rx task. It extracts cells and frames from the underlying DUPLEX device uP interface and sends them to the application task using the `indRxCell`/`indRxFrm` callback functions
	sysVcsIccRxMsg	This routine is invoked by vcsIndDuplexRxCell() to inform the ICC Rx task of the incoming cells/frames.

Service Type	Function Name	Description
Watchdog Polling task	sysVcsWdgInstall	Spawns the Watchdog task
	sysVcsWdgRemove	Deletes the Watchdog task.
	sysVcsWdgPtrlTaskFn	This function is executed in the context of the Watchdog task. It activate watchdog patrol for APEX devices, and poll the Ingress COS Status FIFO of ATLAS devices
Timer	sysVcsDelayTask	Puts the currently executing task to sleep for a specified number of milliseconds

Step 3: Porting the System-Specific utility functions

Porting the system-specific utility function includes modifying the utility functions in `vcs_sys.c` file. You may tailor them to your own network system requirements.

To port the driver's system-specific utility functions:

1. Modify the routines for calculating congestion thresholds, `sysVcsPortThresholds` and `sysVcsVcThresholds`.

2. Modify the routines for calculating scheduling and shaping parameters, which include `sysVcsLoopPortScheduler`, `sysVcsWANPortScheduler`, `sysVcsVcClassScheduler` and `sysVcsVcShaping`.

3. Modify the Tables for the default policing actions vs. traffic type.

4. Modify the port mapping routines, if you wish to use a different mapping algorithm.

Step 4: Porting the Application-Specific Elements

Porting the application-specific elements includes coding the indication callback functions and defining the initialization vector for chipset devices.

To port the driver's application-specific elements:

1. Modify the default device initialization vectors in `vcs_test.c` to meet your application needs, reflect the chipset core card architecture, and provide bus interface consistence between the chipset devices.

2. Code the callback functions according to the application. Example implementations of these callback functions are provided in `vcs_test.c`. The callback functions are the following:

```
void indRxDataCell(UINT4 u4ECI, sVCS_CELL_HDR *psHdr,
UINT1 *pu1Pyld, INT4 result);

void indRxDataFrm(UINT4 u4ECI, sVCS_CELL_HDR *psHdr, UINT1
*pu1Frm, UINT4 u4Length, INT4 result);

void indRxCtrlMsg(UINT2 u2ChnlId, INT4 result);

void indRxBOC(VCS_USR_CTXT vcsUsrCtxt, sVCS_DEV_ID
*psDevId, UINT1 u1HssLnk, UINT1 u1BOCcode);

void indRxOAM(VCS_USR_CTXT vcsUsrCtxt, sVCS_DEV_ID
*psDevId, INT4 u4OamType, UINT1 u1CmdFlag, INT4 arg1,INT4
connId, INT4 result);

void indCosStatus(UINT4 u4ICI, UINT2 u2Status);


void indCritical(VCS_USR_CTXT vcsUsrCtxt, sVCS_DEV_ID
*psDevId, UINT4 u4EventId, UINT4 arg1, UINT4 arg2, UINT4
arg3);

void indError(VCS_USR_CTXT vcsUsrCtxt, sVCS_DEV_ID
*psDevId, UINT4 u4EventId, UINT4 arg1, UINT4 arg2, UINT4
arg3);
```

Step 5: Building the Driver

This section describes how to build the 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 chipset API driver library using your make utility.

4. Link the chipset API driver library to your application code.

12 CODING CONVENTIONS

This section describes the coding conventions used in the implementation of all PMC driver software.

12.1 Variable Type Definitions

Table 51: Variable Type Definitions

Type	Description
UINT1	unsigned integer – 1 byte
UINT2	unsigned integer – 2 bytes
UINT4	unsigned integer – 4 bytes
INT1	signed integer – 1 byte
INT2	signed integer – 2 bytes
INT4	signed integer – 4 bytes
VOID	void

12.2 Naming Conventions

Table 52 presents a summary of the naming conventions followed by all PMC driver software. A detailed description is then given in the following sub-sections.

The names used in the drivers are verbose enough to make their purpose fairly clear. This makes the code more readable. Generally, the device's name or abbreviation appears in prefix.

Table 52: Naming Conventions

Type	Case	Naming convention	Examples
Macros	Uppercase	prefix with "m" and device abbreviation	mVCS_WRITE

Type	Case	Naming convention	Examples
Constants	Uppercase	prefix with device abbreviation	`VCS_REG`
Structures	Hungarian Notation	prefix with “s” and device abbreviation	`sVCS_DDB`
API Functions	Hungarian Notation	prefix with device name	`vcsAdd()`
Porting Functions	Hungarian Notation	prefix with “sys” and device name	`sysVCSReadReg()`
Other Functions	Hungarian Notation		`myOwnFunction()`
Variables	Hungarian Notation		`maxDevs`
Pointers to variables	Hungarian Notation	prefix variable name with “p”	`pmaxDevs`
Global variables	Hungarian Notation	prefix with device name	`vcsGDD`

Macros

- Macro names must be all uppercase.
- Words shall be separated by an underscore.
- The letter `'m'` in lowercase is used as a prefix to specify that it is a macro, then the device abbreviation must appear.
- Example: `mVCS_WRITE` is a valid name for a macro.

Constants

- Constant names must be all uppercase.
- Words shall be separated by an underscore.
- The device abbreviation must appear as a prefix.
- Example: `VCS_REG` is a valid name for a constant.

Structures

- Structure names must be all uppercase.
- Words shall be separated by an underscore.
- The letter ‘***s***’ in lowercase must be used as a prefix to specify that it is a structure, then the device abbreviation must appear.
- Example: sVCS_DDB is a valid name for a structure.

Functions

API Functions

- Naming of the API functions must follow the hungarian notation.
- The device's full name in all lowercase shall be used as a prefix.
- Example: `vcsAdd()` is a valid name for an API function.

Porting Functions

Porting functions correspond to all function that are HW and/or RTOS dependent.

- Naming of the porting functions must follow the hungarian notation.
- The '`sys`' prefix shall be used to indicate a porting function.
- The device's name starting with an uppercase must follow the prefix.
- Example: sysVCSReadReg() is a hardware / RTOS specific.

Other Functions

- Other Functions are all the remaining functions that are part of the driver and have no special naming convention. However, they must follow the hungarian notation.
- Example: `myOwnFunction()` is a valid name for such a function.

Variables

- Naming of variables must follow the hungarian notation.
- A pointer to a variable shall use '`p`' as a prefix followed by the variable name unchanged. If the variable name already starts with a '`p`', the first letter of the variable name may be capitalized, but this is not a requirement. Double pointers might be prefixed with '`pp`', but this is not required.
- Global variables must be identified with the device's name in all lowercase as a prefix.
- Examples: `maxDevs` is a valid name for a variable, `pmaxDevs` is a valid name for a pointer to `maxDevs`, and `vcsBaseAddress` is a valid name for a global variable. Note that both `pprevBuf` and `pPrevBuf` are accepted names for a pointer to the `prevBuf` variable, and that both `pmatrix` and `ppmatrix` are accepted names for a double pointer to the variable matrix.

12.3 File Organization

Table 53 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 should convey their purpose with a minimum amount of characters. If a file size is getting too big one might separate it into two or more files, providing that a number is added at the end of the file name (e.g. `vcs_api1.c` or `vcs_api2.c`).

There are 4 different types of files:

- The API file containing all the API functions
- The hardware file containing the hardware dependent functions
- The RTOS file containing the RTOS dependent functions
- The other files containing all the remaining functions of the driver

Table 53: File Naming Conventions

File Type	File Name
API	vcs_api1.c, vcs_api.h
Hardware Dependent	vcs_hw.c, vcs_hw.h
RTOS Dependent	vcs_rtos.c, vcs_rtos.h
Other	vcs_init.c, vcs_init.h

API Files

- The name of the API files must start with the device abbreviation followed by an underscore and '`api`'. Eventually a number might be added at the end of the name.
- Examples: `vcs_api1.c` is the only valid name for the file that contains the first part of the API functions, `vcs_api.h` is the only valid name for the file that contains all of the API functions headers.

Hardware Dependent Files

- The name of the hardware dependent files must start with the device abbreviation followed by an underscore and '`hw`'. Eventually a number might be added at the end of the file name.

- Examples: `vcs_hw.c` is the only valid name for the file that contains all of the hardware dependent functions, `vcs_hw.h` is the only valid name for the file that contains all of the hardware dependent functions headers.

RTOS Dependent Files

- The name of the RTOS dependent files must start with the device abbreviation followed by an underscore and '`rtos`'. Eventually a number might be added at the end of the file name.
- Examples: `vcs_rtos.c` is the only valid name for the file that contains all of the RTOS dependent functions, `vcs_rtos.h` is the only valid name for the file that contains all of the RTOS dependent functions headers.

Other Driver Files

- The name of the remaining driver files must start with the device abbreviation followed by an underscore and the file name itself, which should convey the purpose of the functions within that file with a minimum amount of characters.
- Examples: `vcs_init.c` is a valid name for a file that would deal with initialization of the device, `vcs_init.h` is a valid name for the corresponding header file.

13 APPENDIX A: CALCULATION OF CONGESTION THRESHOLD AND SCHEDULING PARAMETERS

13.1 Introduction

The queue engine of the APEX has a fixed amount of resources to buffer the cells of the active connections. To allocate these resources fairly and effectively, the application has to set congestion thresholds at the port, class and connection level, which guarantee certain amount of resources to the connection during conditions of congestion. The minimum VC level congestion parameters are determined by the traffic type and QOS parameters for the connection. The actual port/class/VC congestion thresholds are calculated and dynamically updated based on the minimum thresholds of the existing connections in the port/class.

The queue engine of the APEX provides scheduling at three levels, port, class and connection level. The scheduling parameters are determined by the type of traffic and the QOS parameters. The scheduling parameters have to be set in a way that ensures fair scheduling within the same class.

The chipset driver has utility functions to calculate the port, class and connection thresholds for congestion management. It also has utility functions to calculate port, class and connection scheduling parameters to ensure fair scheduling. The algorithm for calculating the congestion thresholds and the scheduling parameters is network specific and thus the utility functions in the chipset driver should only be considered as a sample implementation. The user may modify these routines to tailor them to their own network requirements.

13.2 Calculation of congestion thresholds

The issues to be considered when calculating the congestion parameters for the port, class and connection are as follows:

(1) Based on the QOS parameters the driver has to guarantee a certain amount of resources to the connection. These resources should be available to the connection during conditions of congestion. Therefore once the driver guarantees the resources to a connection, these resources are reserved for the connection and cannot be shared with any other connection.

(2) There are conditions, where all the system resources are not reserved for the existing connections. In this case we have spare resources, which would be wasted. So in order to utilize this spare resources the utility functions should distribute the spare resources fairly among the existing connections based on class types. This would increase the throughput of the existing connections.

The congestion algorithm has to strictly follow the first condition and maintain the QOS resource guarantee under all conditions. As far as sharing the spare resource the driver can use different algorithms to fairly distribute the additional resources. One option is to distribute the spare resource equally amongst all connection. In this scenario, every time a new connection is added, due to a reduction in spare resources, the driver will have to re-calculate the congestion thresholds for all existing connections. This could be computationally intensive since the chip set can have a maximum of 64K connections. Therefore the algorithm has to strike a balance between distributing spare resource fairly and not be too computationally intensive.

In the algorithm implemented by the driver, the application will specify the direction thresholds for the Loop and WAN directions at the time of initialization. The minimum port threshold is specified by the application at the time of the port setup configuration. The driver will calculate the port thresholds to guarantee the requested resources and also to distribute the spare resources available, among all the ports in the direction of the port being configured. The spare resources allocated to the port are then distributed among the classes in these ports. This means that each time a port is added or deleted, the spare resources will change Thus, we will have to recalculate the thresholds for all the configured ports in that direction and also the thresholds for all the classes within these ports.

On the other hand, when a connection is configured, the connection thresholds are calculated such that the minimum resources required for the connection are allocated, based on the QOS parameters, and the spare resources available in each class is shared among all the connections within the class. Each time a connection is added or deleted or the QOS of the connection is updated, the spare resources available in the class changes. Therefore the thresholds of all the classes and all the connections, in the port on which the connection is modified, are recalculated.

The following sections will explain in greater detail as to how the different thresholds are calculated.

Direction threshold

The direction thresholds for the Loop direction and the WAN direction are provided by the application in the initialization vector for the APEX chip. The direction thresholds specified by the user should conform to the following conditions:

- dirClp1Thresh < dirClp0Thresh < dirMaxThresh
- (dirMaxThresh for Loop) + (dirMaxThresh for WAN) = (maxCellBuf)*(z)

`where maxCellBuf` is provided by the application in the initialization vector for APEX reflects the total number of cell buffers available to the APEX chip. Maximum value of `maxCellBuf` is 256K cells.

`z` – statistical multiplexing factor determined by user

`z` > 1 means statistical multiplexing is assumed

z = 1 means no statistical multiplexing

The number of maximum cell buffers is dependent on the amount of SDRAM connected to the APEX. In the reference design implementation the maximum number of cell buffers is 64K.

Port threshold

The application will request a minimum port threshold (portMinThresh) at the time of port configuration. The value of the minimum port threshold would depend on the port profile e.g. in case of a DSLAM implementation it would depend on the bandwidth of the DSL modem, which sends traffic on this port. It should be noted that the portMinThresh is a guarantee that this resource is reserved for the connections configured on this port. Since the driver is guaranteeing this resource, it should first check whether enough resources are available. To do this it will use the following criteria:

(Sum of portMinThresh for all existing ports for the port direction) + (portMinThresh for new port) <= (dirMaxThresh for the port direction)

If this condition is not met, the port configuration request will be rejected. If the condition is met the port configuration request is honored.

Even though the port is guaranteed portMinThresh cell buffers, the available resource could be larger than this number. To share the spare resources fairly amongst all the active ports in a particular direction (Loop or WAN), the driver will calculate the portMaxThresh using the following criteria:

- portMaxThresh = (portMinThresh) * (dirMaxThresh for the direction) / (sum of portMinThresh for all active ports in the direction)

The driver will calculate the portClp0Thresh and portClp1Thresh as follows:

- portClp1Thresh = (portMaxThresh)*(dirClp1Thresh)/ (maxDirThresh)

- portClp0Thresh = (portMaxThresh)*(dirClp0Thresh)/ (dirMaxThresh)

Note: each time the application adds a new port, the spare resources will be reduced by a certain amount and the driver will have to recalculate and update the congestion threshold for all the ports in a particular direction (2048 for loop and 4 for WAN). Similarly when a port is deleted, the spare resources will be increased by a certain amount and all the port thresholds have to be recalculated.

If the application does not wish to use this algorithm, the API for configuring the port can be invoked with the `forceFlag` set to 1. In this case, the application specified values of `portMaxThresh`, `portClp1Thresh` and `portClp0Thresh` are used without any change. No spare resources are allocated to the port and the threshold values for the port are unchanged until the port is deleted.

The port threshold parameters are written to the APEX chip as 4 bit log and 4 bit fractional format. The tables for encoding the values are shown in Table 60. When setting the port thresholds the application should also consider the rounding off error, while converting an integer value to a 4 bit log 4 bit fractional value.

Class threshold

The class thresholds will depend on the following:

(1) It will depend on the connections configured under each class. The class thresholds should be large enough to be able to maintain the 'resource guarantee' given to each connection in the class.

(2) It will also depend on how the spare resources within the port are to be distributed between the 4 classes.

The spare resources within the port is calculated as follows:

Spare resources = `portMaxThresh` –

(sum of (`vcMinThresh`*(1 - `CLR`)) for connections in class 0) –

(sum of (`vcMinThresh`*(1 - `CLR`)) for connections in class 1) –

(sum of `vcMinThresh` for connections in class 2) –

(sum of `vcMinThresh` for connections in class 3)

where `CLR` is the cell loss ratio for the connection.

The spare resources are distributed between the 4 classes based on the value of the constants `VCS_SPARE_RESOURCES_CLASS0`, `VCS_SPARE_RESOURCES_CLASS1`, `VCS_SPARE_RESOURCES_CLASS2`, `VCS_SPARE_RESOURCES_CLASS3`. The constants determine the percentage of the spare resources allocated to a particular class and the sum of these 4 constants should be less than or equal to 100.

The class thresholds for the 4 classes are determined as follows:

Class 0:

classMaxThresh = (sum of (vcMinThresh*(1 - CLR)) for connections in class 0) + (VCS_SPARE_RESOURCES_CLASS0 * spare resources)

classClp0Thresh = (sum of vcClp0Thresh for connections in class 0) + (VCS_SPARE_RESOURCES_CLASS0*spare resources)

classClp1Thresh = (sum of vcClp1Thresh for connections in class 0) + (VCS_SPARE_RESOURCES_CLASS0*spare resources)

Class 1:

classMaxThresh = (sum of (vcMinThresh*(1 - CLR)) for connections in class 1) + (VCS_SPARE_RESOURCES_CLASS1 * spare resources)

classClp0Thresh = (sum of vcClp0Thresh for connections in class 1) + (VCS_SPARE_RESOURCES_CLASS1*spare resources)

classClp1Thresh = (sum of vcClp1Thresh for connections in class 1)+ (VCS_SPARE_RESOURCES_CLASS1*spare resources)

Class 2:

classMaxThresh = (sum of vcMinThresh for connections in class 2) + (VCS_SPARE_RESOURCES_CLASS2*spare resources)

classClp0Thresh = (sum of vcClp0Thresh for connections in class 2)+ (VCS_SPARE_RESOURCES_CLASS2*spare resources)

classClp1Thresh = (sum of vcClp1Thresh for connections in class 2)+ (VCS_SPARE_RESOURCES_CLASS2*spare resources)

Class 3:

ClassMaxThresh = (sum of vcMinThresh for connections in class 3) + (VCS_SPARE_RESOURCES_CLASS3 * spare resources)

classClp0Thresh = (sum of vcClp0Thresh for connections in class 3)+ (VCS_SPARE_RESOURCES_CLASS3*spare resources)

`classClp1Thresh =` (sum of `vcClp1Thresh` for connections in class 3)+(VCS_SPARE_RESOURCES_CLASS3*`spare resources`)

As explained in the next section, the connections in class 0 and 1 are connections of traffic type CBR and RT-VBR. These connections by their nature are latency-senstive type connections and their traffic normally has small latency. They usually do not need spare resources allocated to them. On the other hand the connections in class 2 and 3 are of traffic type NRT-VBR, GFR, UBR and ABR, which are less sensitive in latency and their traffic could have large latency.. Allocating spare resources to these classes will increase the buffering space and therefore reduces congestion of these connections. Therefore, majority of the spare resources should be allocated to classes 2 and 3.

Note: each time a port is added or deleted, the value of the spare resources for each port in the same direction will change. This in turn will affect the class thresholds. On the other hand, each time a connection is added or deleted, the thresholds of the class containing the connection will change. Thus, each time a port is added or deleted, or a connection is added, updated or deleted, the class thresholds will have to be recalculated.

The class thresholds are written to the APEX chip in 4 bit log and 4 bit fractional format. This is the same format as used for the port thresholds. The tables for encoding the values are shown in Table 60.

Connection threshold

When configuring a connection, the application will provide the QOS parameters for the connection. The driver will utilize the QOS parameters to assign the connection to a particular class and calculate the thresholds depending on traffic type. The QOS parameters provided by the application are as follows:

Table 54: QOS parameters provided by the application

QOS parameter	Description of QOS parameter
Traffic type	Whether connection is CBR, rt-VBR, nrt-VBR, GFR, ABR, UBR
PCR	Peak cell rate
SCR	Sustained cell rate
MBS	Maximum burst size at peak cell rate
CDVT	Cell delay variance tolerance
MaxCTD	Maximum cell transfer delay

QOS parameter	Description of QOS parameter
CLR	Cell Loss Ratio
MFS	Maximum frame size

The driver will assign the connection to a certain class based on the traffic type. The table below shows the mapping between class and traffic type.

Table 55: Class assignment for different traffic types

	CBR	rt-VBR	nrt-VBR	GFR	UBR	ABR
Class	0	1	2	2	3	3

Table 56: Calculation of connection congestion thresholds

Traffic	VcMinThrsh	VcCLP1Thrsh	VcCLP0Thrsh	VcMaxThrsh (EPD/PPD only)	Comment
CBR	maxCTD	MaxCTD	maxCTD	maxCTD	No value storing more than the maxCTD. Tagging not applied to this traffic, hence CLP0 & CLP1 Thrsh values are identical.
rt-VBR	maxCTD	MaxCTD	maxCTD	maxCTD	From congestion perspective, rt-VBR and CBR are identical. Difference lies in CDV tolerance reflected in class scheduling.

Traffic	VcMinThrsh	VcCLP1Thrsh	VcCLP0Thrsh	VcMaxThrsh (EPD/PPD only)	Comment
nrt-VBR	MBS	MBS * (PCR – SCR) / PCR	MBS * n, n > 1	VcCLP0Thrsh + est. MFS	Minimum CLR is the focus. Always want sufficient resources to capture a burst. VcCLP1Thrsh set to a level where the VC is within traffic contract. VcCLP0Thrsh set to a level where the VC has some burstiness caused by the network. VcMaxThrsh set to accept the last frame permitted under VcCLP0Thrsh.
GFR	MBS	MBS * (PCR-MCR) / PCR	MBS * n, n > 1	VcCLP0Thrsh + MFS	Very similar to nrt-VBR, except packet centric.
UBR	0	1	est. MFS	VcCLP0Thrsh + est. MFS	No minimum resource guarantees.
ABR	0	1	1	est. MFS + 1	The CLP0 & 1 thresholds are kept to very small values, relying on the connection

Traffic	VcMinThrsh	VcCLP1Thrsh	VcCLP0Thrsh	VcMaxThrsh (EPD/PPD only)	Comment
					s/w to regulate traffic rates and avoid congestion.

Once the driver determines the class to which a connection is assigned and the thresholds are calculated (according to the above table), it distributes the spare resources available in the class equally among the connections configured in the class. The equations for calculating the spare resources in the class are the same as shown in the section of class thresholds. Thus the connection thresholds are calculated by the following equations:

`vcMaxThrsh = vcMaxThrsh` (as calculated from above table) +
(spare resources available in class / number of connections in class)

`vcClp1Thrsh = vcClp1Thrsh` (as calculated from above table) +
(spare resources available in class / number of connections in class)

`vcClp0Thrsh = vcClp0Thrsh` (as calculated from above table) +
(spare resources available in class / number of connections in class)

Note that each time a connection is added, deleted or the QOS parameters are updated, the spare resources allocated to each class in the port will change. Thus, the spare resources allocated to each connection within the port will change. Therefore, the driver will recalculate the congestion thresholds for all the connections in the port.

The connection thresholds `vcClp0Thresh`, `vcClp1Thrsh` and `vcMaxThrsh` are written to the APEX chip in 4 bit log and 2 bit fractional format. The tables for encoding the values are shown in Table 61.

13.3 Calculation of scheduling parameters

Assigning port weights

During configuration of a loop port or WAN port, a weight has to be assigned to the port, which determines the relative polling frequency for that port. Lower polling weight means higher polling frequency and therefore higher throughput. For the loop port the weights range from 0-7, whereas for the WAN port the weights range from 0-3. The maximum polling frequency of the scheduler is dependent on the system clock (e.g. for `sysClk` of 80MHz the maximum polling frequency is 1.25MHz). Given the maximum polling frequency of the scheduler, the maximum polling frequency for a port with a particular weight is given by

max polling freq. for weight `n` = (max polling freq. for the system) / (2^{n})

Assuming that the maximum port rate is 400 Kcells we will use the following lookup table to assign the port weights for the loop port

Table 57: Loop port lookup table

Loop port weight	Port Rate (Kcells/sec)
0	350-400
1	300-349
2	150-299
3	75-150
4	40-74
5	20-39
6	10-19
7	0-9

The following lookup table will be used to assign the port weights for the WAN port

Table 58: WAN port lookup table

WAN port weight	Port Rate (Kcells/sec)

0	300-400
1	75-299
2	20-74
3	0-19

Calculating class scheduler parameters

The APEX chip provides us with a mechanism to fix the scheduling priority between the different classes within a port. Each class (except class 0) has a parameter `classXCellLmt`, which determines the amount of time assigned to the class by the scheduler. To determine the `classXCellLmt` parameter, the driver needs to determine the bandwidth required by the connections under each class. The bandwidth can be determined by:

class bandwidth = (Sum of bandwidth guaranteed for each connection in the class)

The bandwidth for each connection will depend on the type of traffic i.e. for CBR traffic use the PCR(Peak cell rate), for rt-VBR and nrt-VBR traffic use SCR (sustained cell rate), for ABR and GFR use MCR (minimum cell rate), for UBR the bandwidth guaranteed is zero.

The percentage of time that should be allocated to each class by the scheduler is given by

Percentage for class x = (class bandwidth for class x) / (sum of all class bandwidth),

where Class Number x ranges from 0 to 3.

Based on the percentage of utilization, the driver calculates the Limit field `classXCellLmt` parameter by using the following table:

Table 59: Class Limit field (`ClassXCellLmt`) setting

Limit Field	Percentage of Usage
0	100.00%
1	50.00%

Limit Field	Percentage of Usage
2	33.33%
3	25.00%
4	20.00%
5	16.67%
6	14.28%
7	12.50%
8	11.11%
9	9.09%
10	7.69%
11	6.66%
12	5.88%
13	4.76%
14	4.00%
15	3.44%

Calculating the weight for a WFQ connection

The weight for a WFQ connection determines the number of transmit opportunities the WFQ connections is given relative to the other connections in the same class. The driver will calculate the weight using the following equation:

queue weight = (126 * VC-PCR) / (Port Rate),

where VC-PCR is the peak cell rate of the connection.

The actual value to be programmed into the APEX chip is encoded, which should be given by

If (queue weight = 1)

actual value = 0

else

actual value = (queue weight/2)

Calculating the shaping parameters for a SFQ connection

Before a shaping connection is configured, the application has to configure the shaper. While configuring the shaper the application will specify the parameter `QShpNRTRate`, which represents the maximum shaped data rate calculated in the number of clock cycles per timeslot. The other shaping parameters for the SFQ connection are calculated as follows:

`ShpIncr` = f(SYSCLK) / (`QShpNRTRate` * `SCR`)

`ShpCdvt` = `ShpIncr` - f(SYSCLK) / (`QShpNRTRate` * `PCR`)

`ShpLateBits` = log (`MBS` * `ShpCdvt`) / log 2

where f(SYSCLK) is the clock frequency for the system, `SCR` is sustained cell rate, `PCR` is peak cell rate and `MBS` is maximum burst size at peak cell rate.

13.4 Conversion tables

Encoding a value to 4 bit log 4 bit fractional value: Used for port and class threshold parameters.

Table 60: 4 Bit Logarithmic, 4 Bit Fractional encoding

	4 bits fractional																
4 Bits Log		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	2	32	34	36	38	40	42	44	46	48	50	52	54	56	58	60	62
	3	64	68	72	76	80	84	88	92	96	100	104	108	112	116	120	124
	4	128	136	144	152	160	168	176	184	192	200	208	216	224	232	240	248
	5	256	272	288	304	320	336	352	368	384	400	416	432	448	464	480	496
	6	512	544	576	608	640	672	704	736	768	800	832	864	896	928	960	992
	7	1024	1088	1152	1216	1280	1344	1408	1472	1536	1600	1664	1728	1792	1856	1920	1984
	8	2048	2176	2304	2432	2560	2688	2816	2944	3072	3200	3328	3456	3584	3712	3840	3968
	9	4096	4352	4608	4864	5120	5376	5632	5888	6144	6400	6656	6912	7168	7424	7680	7936
	10	8192	8704	9216	9728	10240	10752	11264	11776	12288	12800	13312	13824	14336	14848	15360	15872
	11	16384	17408	18432	19456	20480	21504	22528	23552	24576	25600	26624	27648	28672	29696	30720	31744
	12	32768	34816	36864	38912	40960	43008	45056	47104	49152	51200	53248	55296	57344	59392	61440	63488
	13	65536	69632	73728	77824	81920	86016	90112	94208	98304	102400	106496	110592	114688	118784	122880	126976
	14	131072	139264	147456	155648	163840	172032	180224	188416	196608	204800	212992	221184	229376	237568	245760	253952
	15	262143															

Encoding a value to 4 bit log 2 bit fractional: Used for `vcClp0Thresh`, `vcClp1Thresh` and `vcMaxThresh` parameters.

Table 61: 4 Bit Logarithmic, 2 Bit Fractional encoding

	2 bits fractional				
4 Bits Log		0	1	2	3
	0	0	1	2	3
	1	4	5	6	7
	2	8	10	12	14
	3	16	20	24	28
	4	32	40	48	56
	5	64	80	96	112
	6	128	160	192	224
	7	256	320	384	448
	8	512	640	768	896
	9	1024	1280	1536	1792
	10	2048	2560	3072	3584
	11	4096	5120	6144	7168
	12	8191			

Encoding `vcMinThresh` parameter:

Table 62: 3 bit encoding for vcMinThresh

Encoded value	Actual value
000	0
001	24
010	32
011	48
100	64

Encoded value	Actual value
101	96
110	128
111	256

14 APPENDIX B

The appendix contains several diagrams showing the data flow of user cells, OAM cells as well as control channel paths between remote Line/WAN cards and chipset core card.

Figure 17: Upstream Data Flow

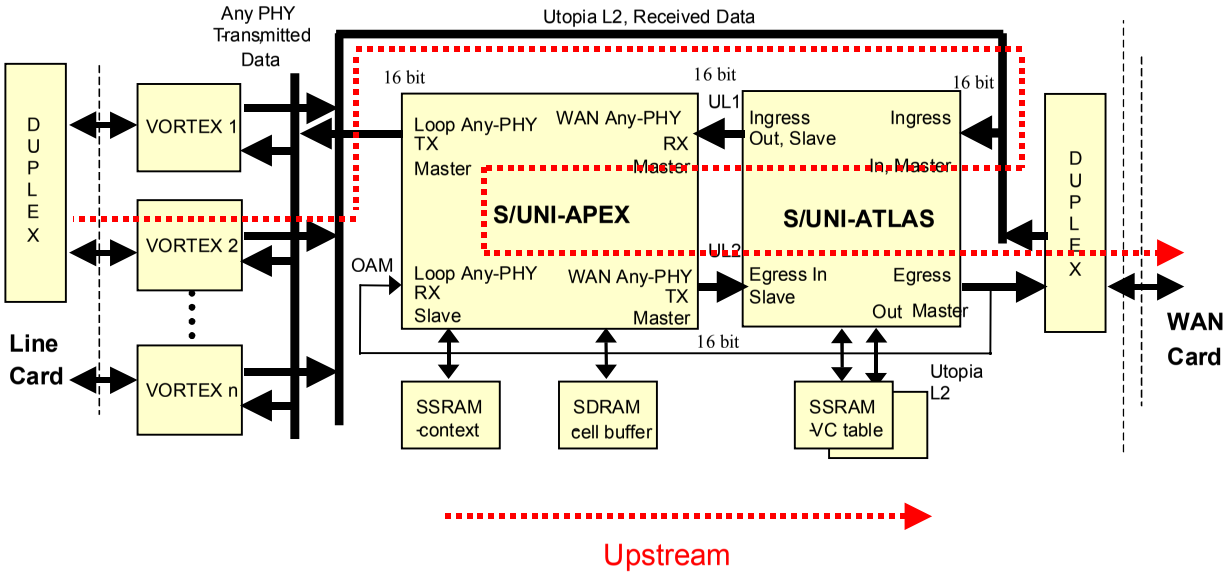


Figure 18: Downstream Data Flow

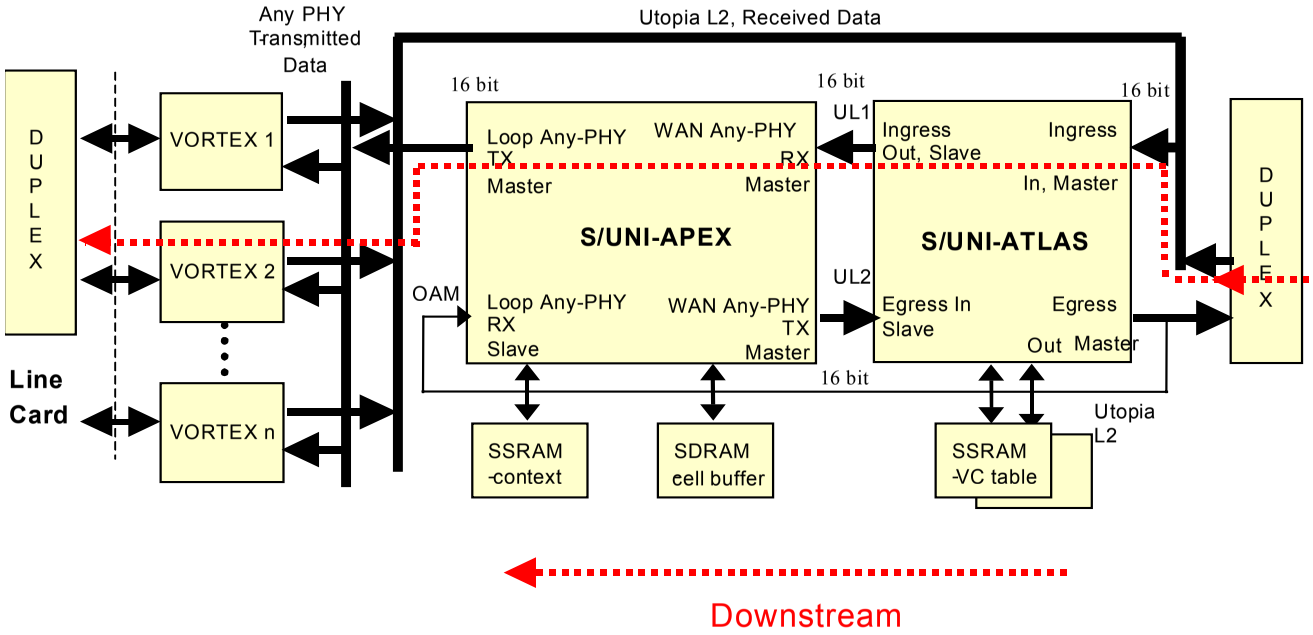


Figure 19: Loop-to-Loop Data Flow

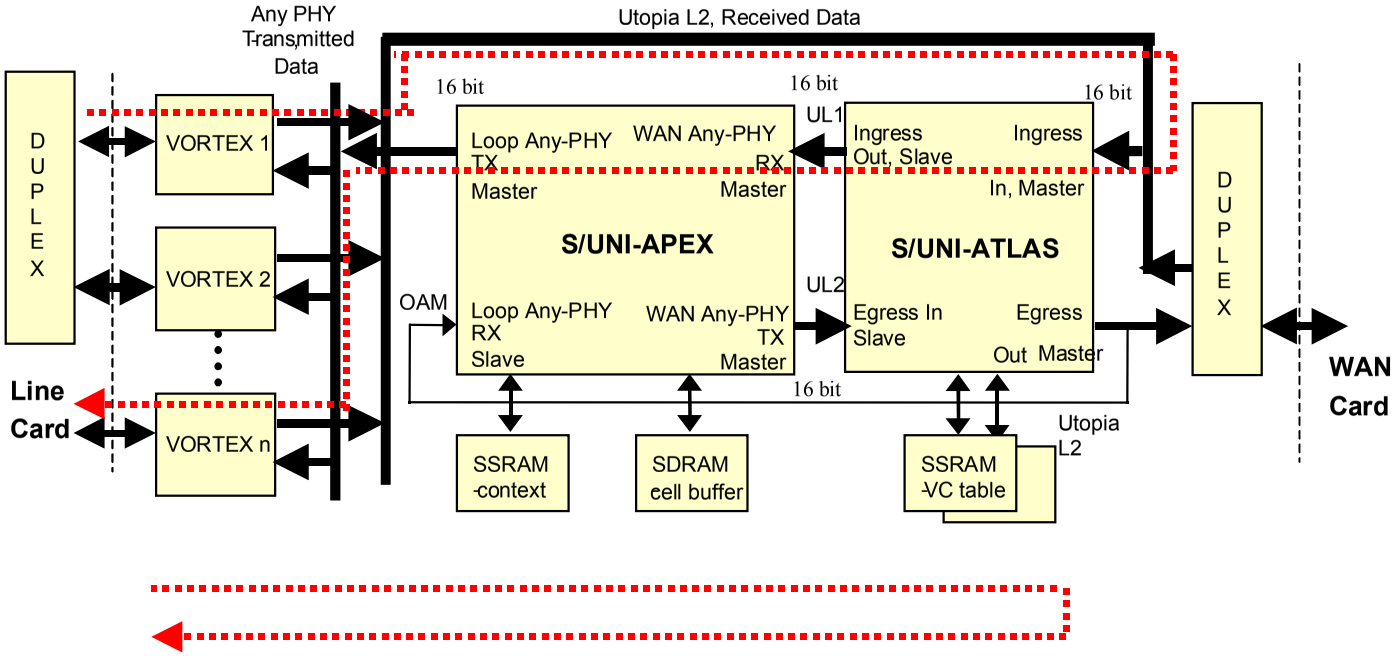


Figure 20: uP-to-WAN and WAN-to-uP Data Flow

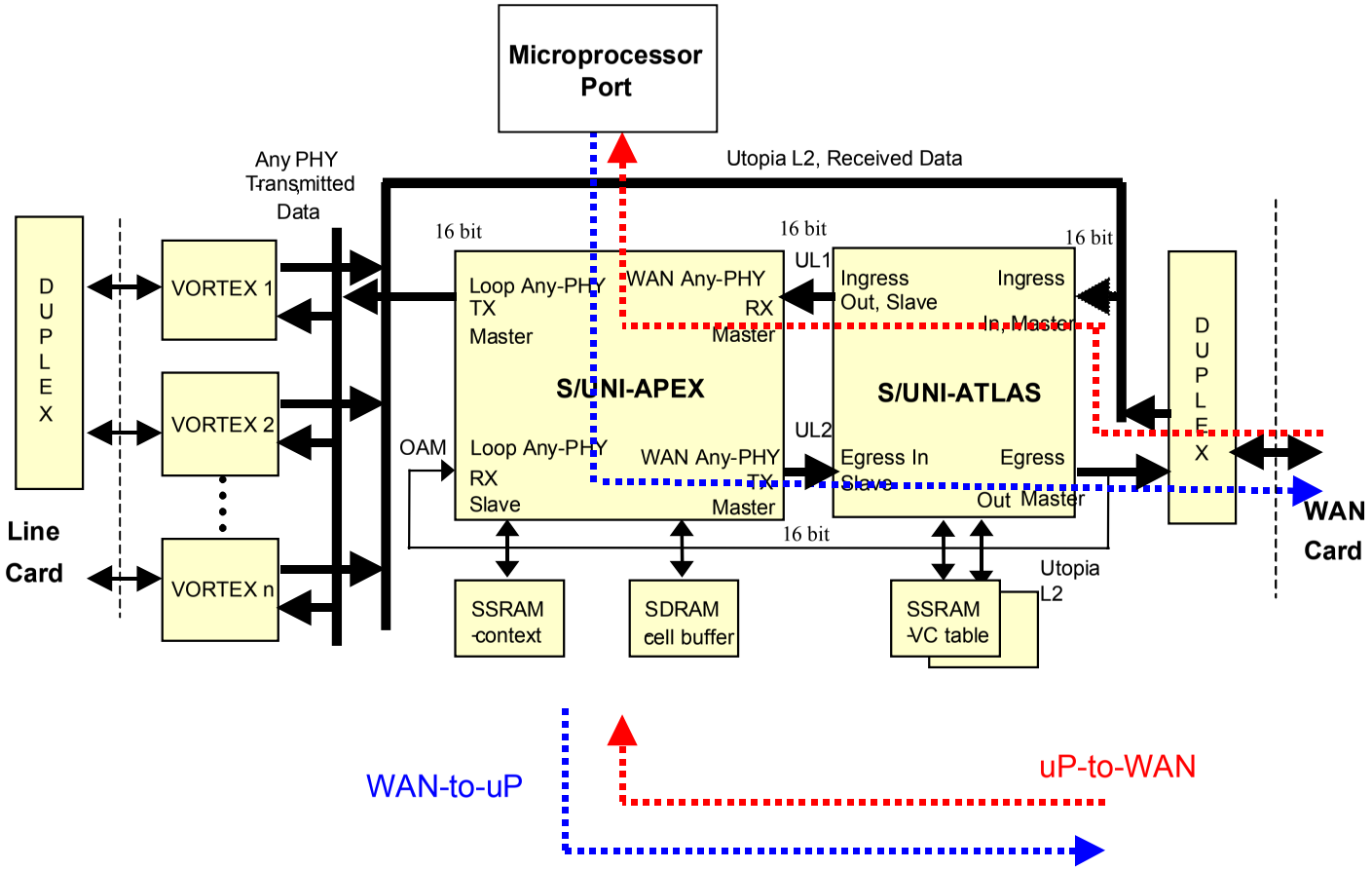


Figure 21: uP-to-Loop and Loop-to-uP Data Flow

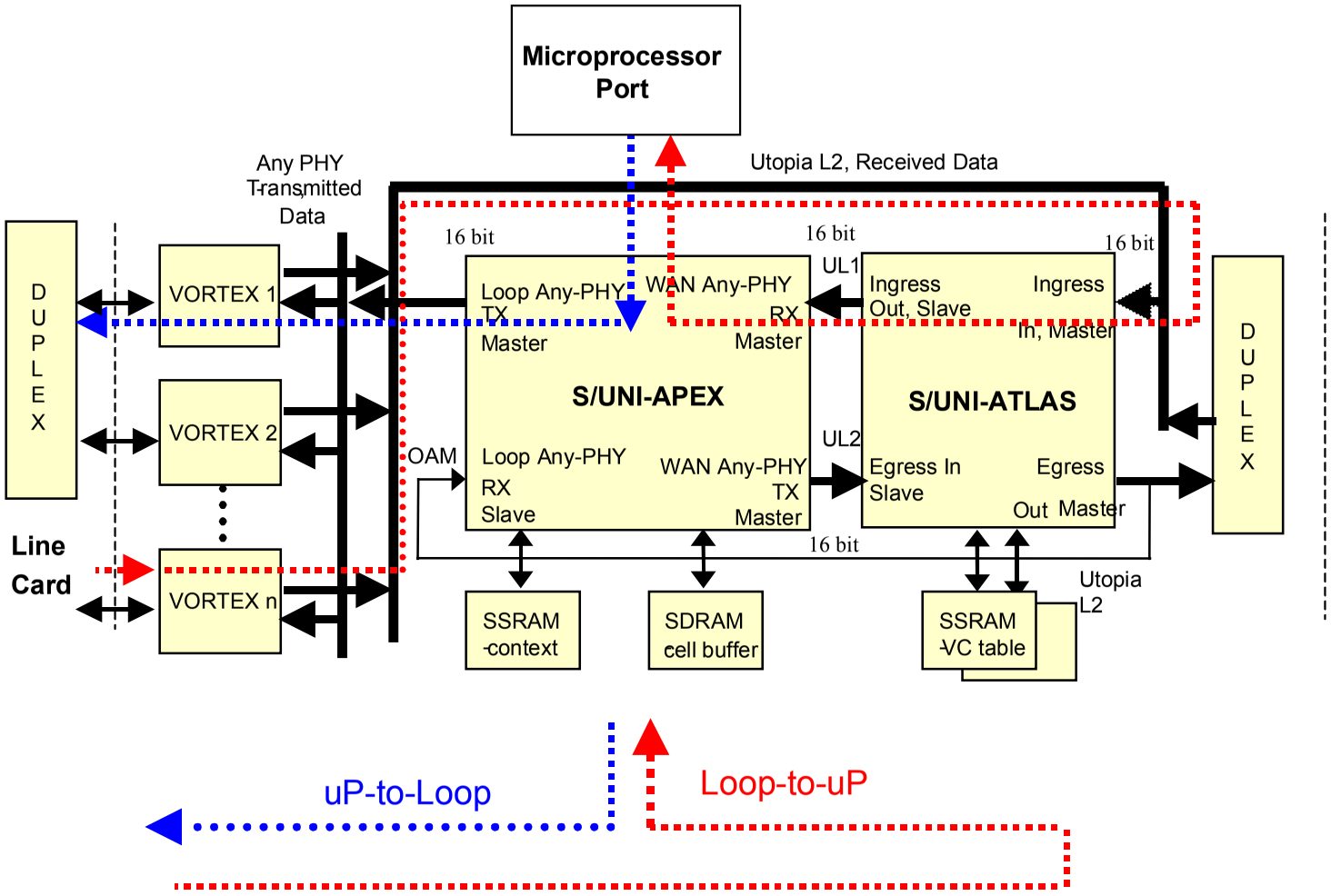


Figure 22: Loopback Data Flow via microprocessor port

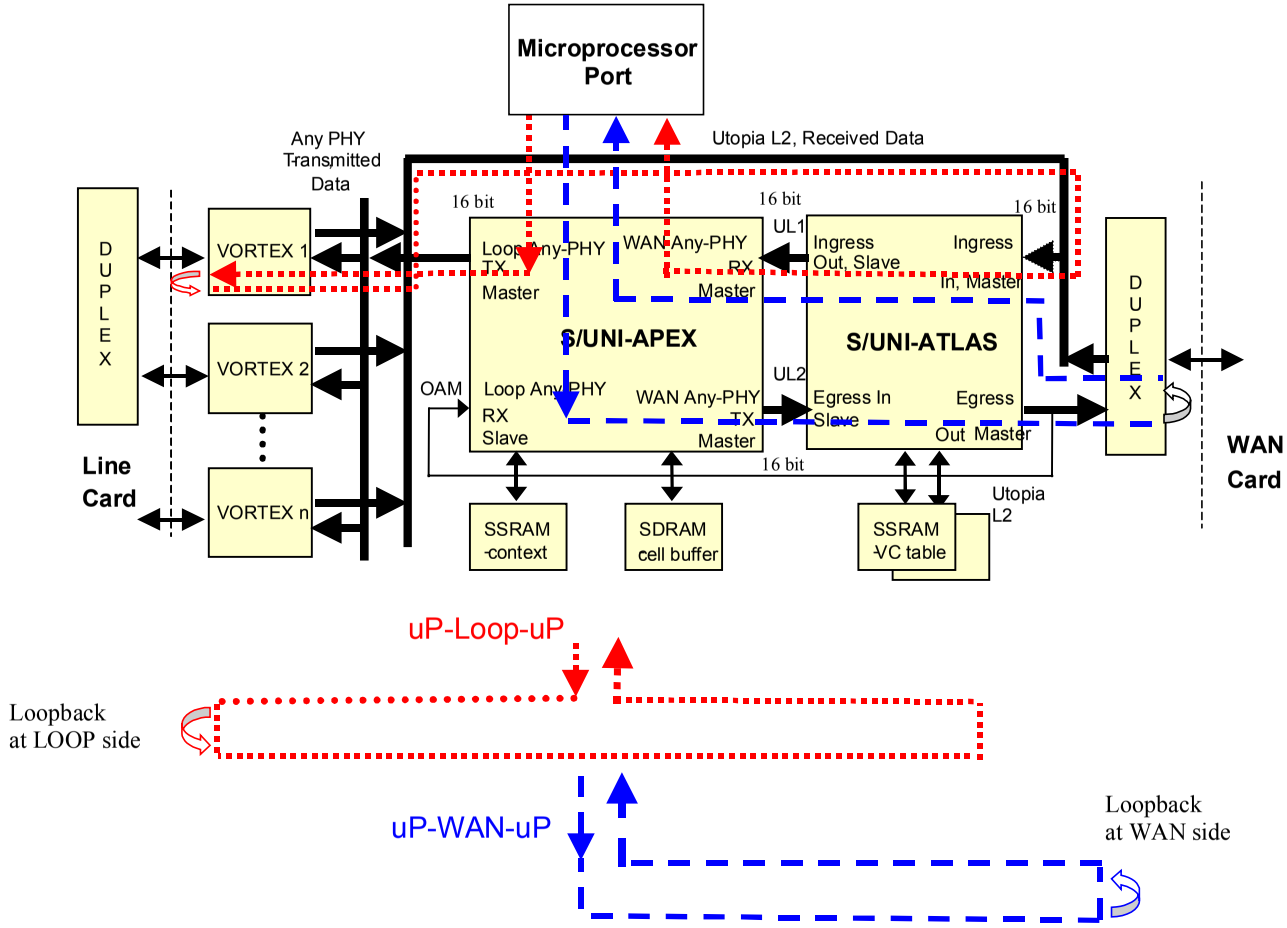


Figure 23: Inband Control Channel Data Flow

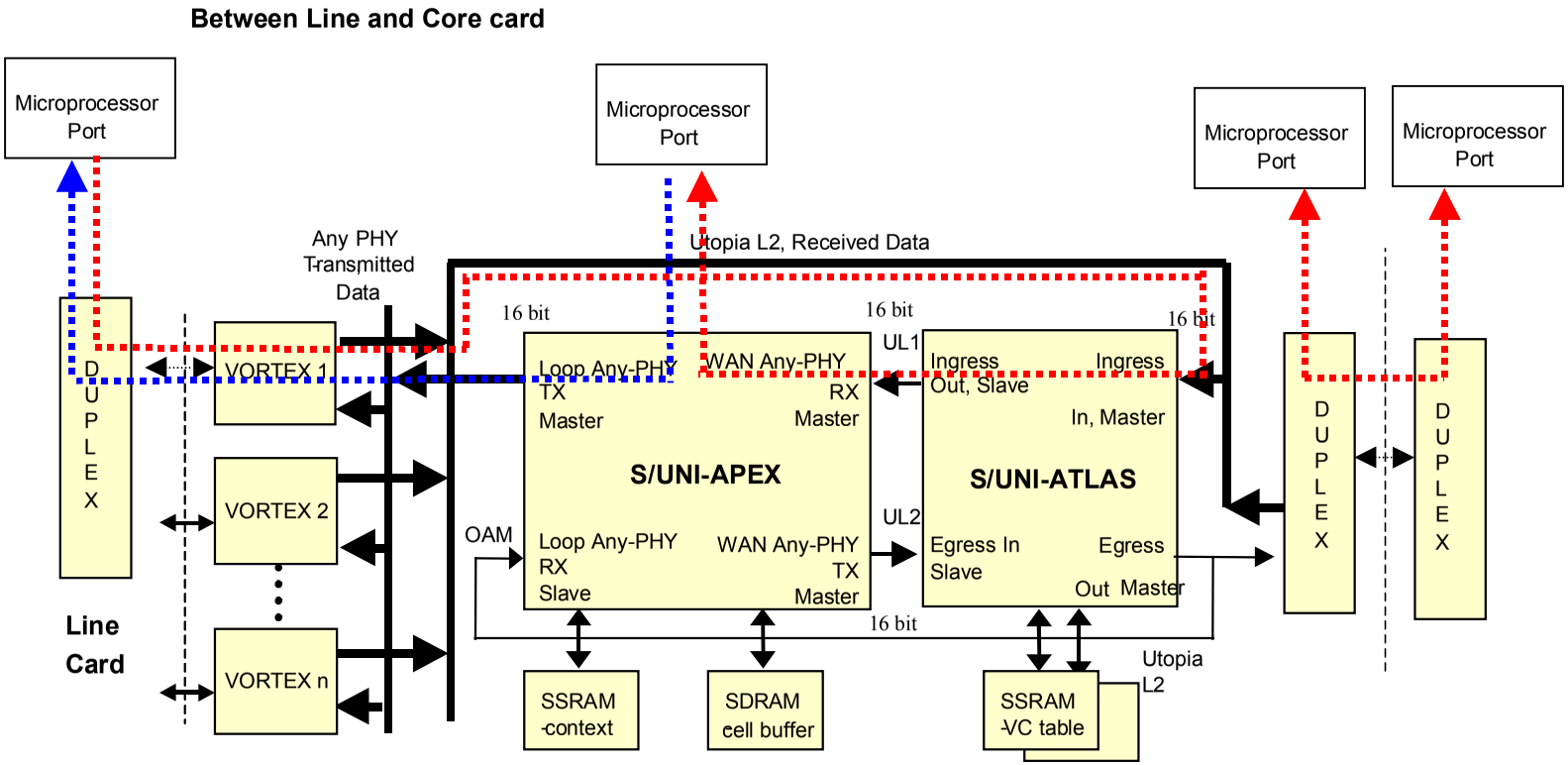


Figure 24: Multicasting Data Flow

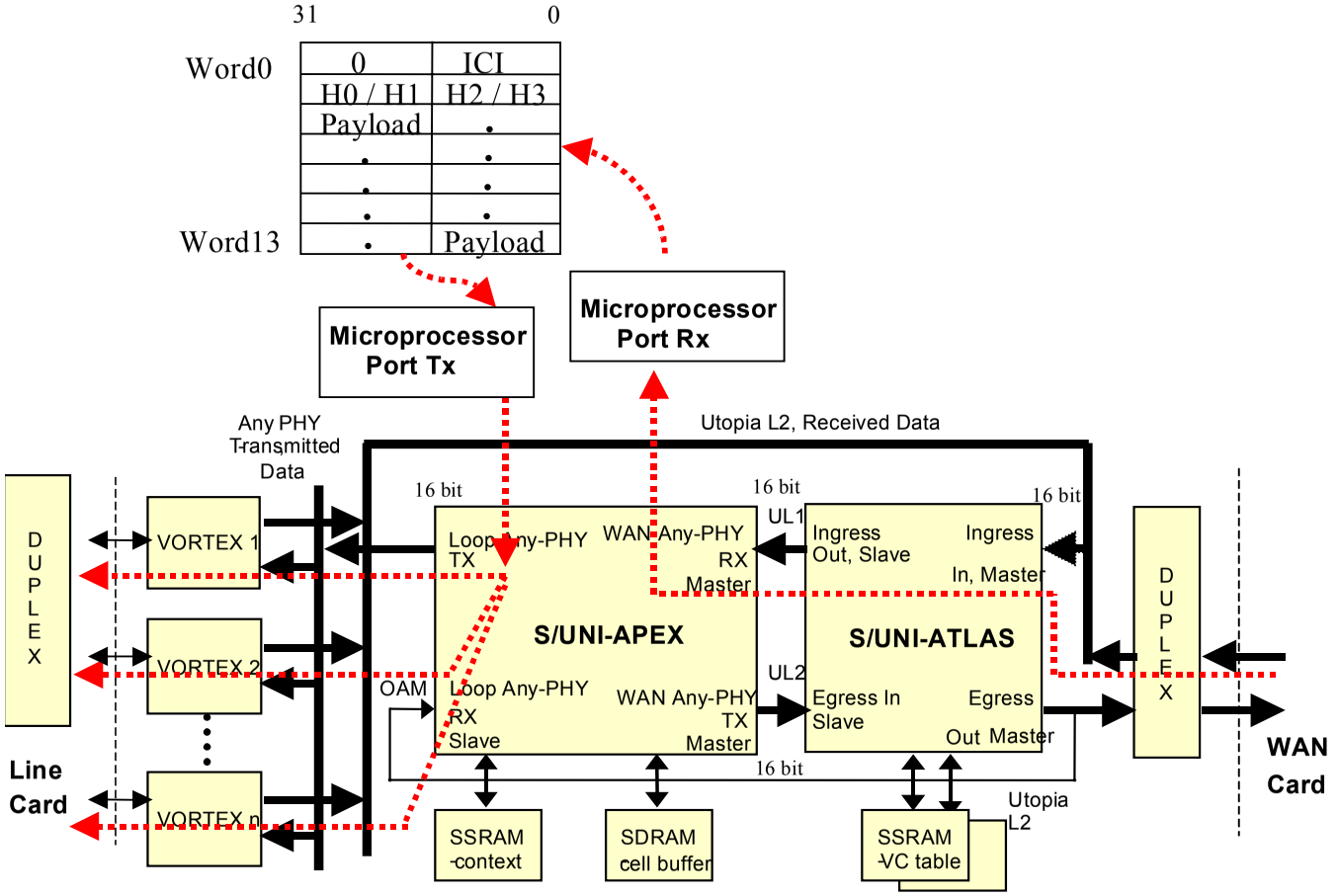


Figure 25: OAM Data Flow

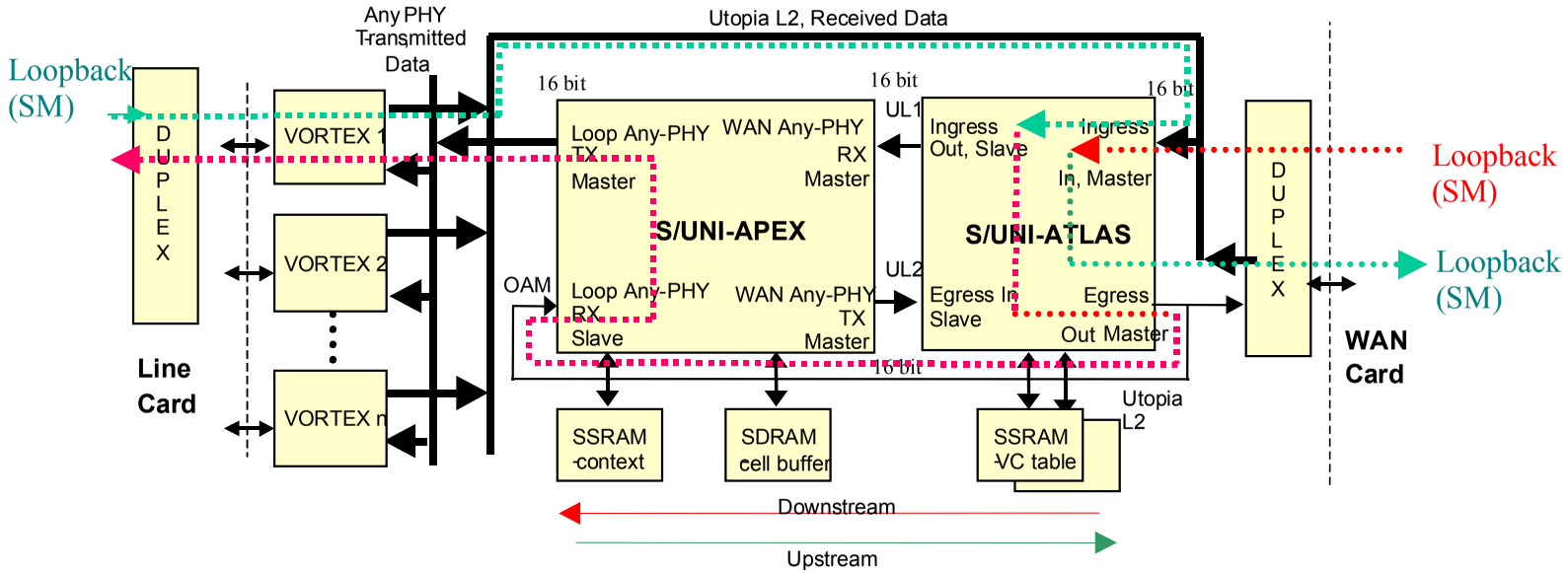
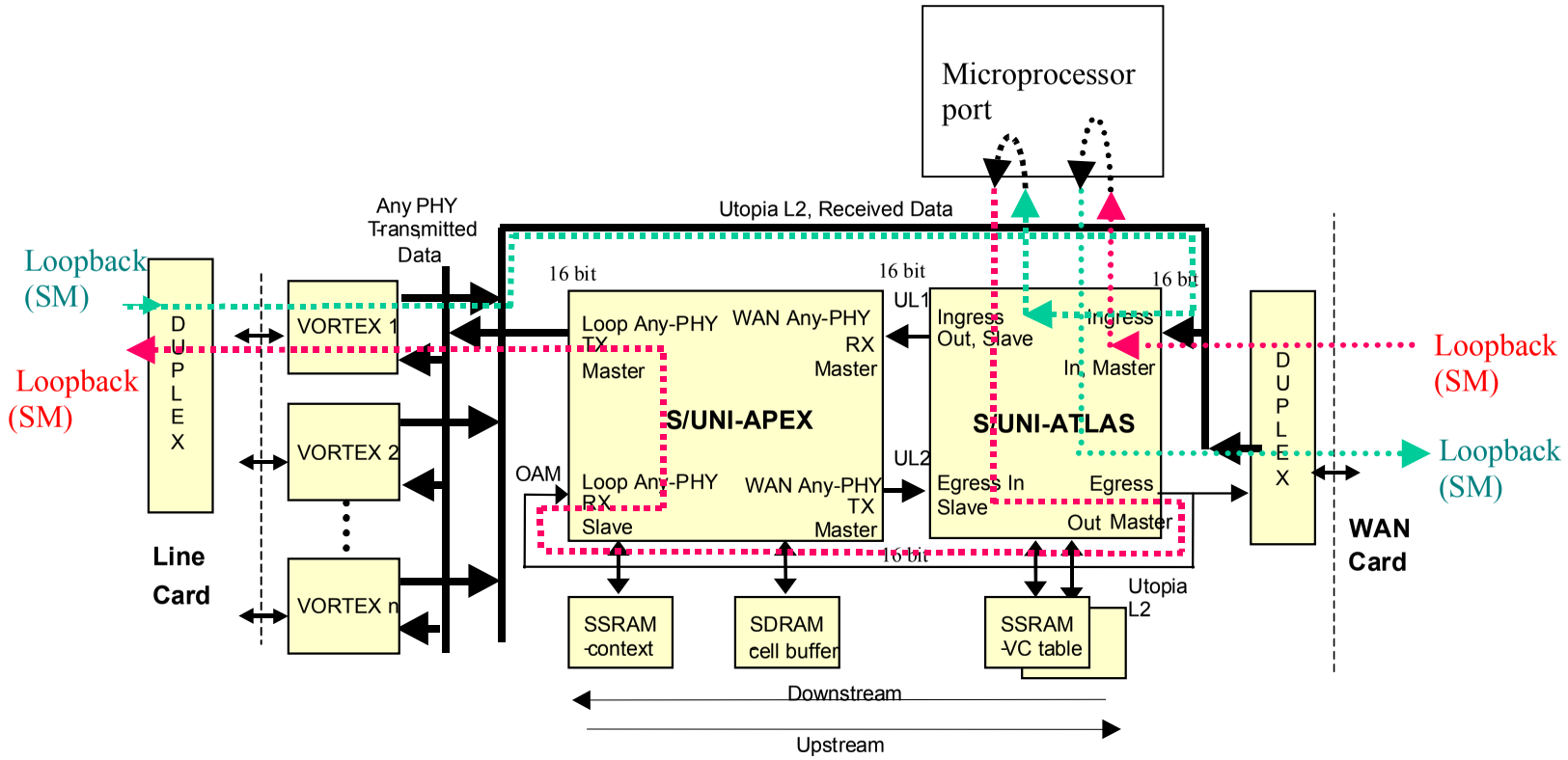


Figure 26: Microprocessor OAM support



14.1 List of Terms

AIS: Alarm Indication Signal.

API (Application Programming Interface): Describes the connection between this MODULE and the USER's Application.

CAC: Connection Admission Control.

CC: Continuity Check.

CDB (Chipset Data Block): Structure that holds the Configuration Data for each Chipset.

DEVICE: One VORTEX chipset Integrated Circuit, which can be APEX, ATLAS, VORTEX, or DUPLEX.

DEVICE DRIVER: A device level software module to control and service an individual type of VORTEX chipset devices.

CHIPSET: A VORTEX chipset consists of APEX, ATLAS , DUPLEX and VORTEX chips.

CHIPSET DRIVER: A board-level software module, which integrates the underlying device drivers, and provides a synchronized access and control over all VORTEX chipset devices on CORE CARDs to achieve one or more system-level functionality.

CHIPSET CARD: A circuit card containing VORTEX chipset devices for traffic management. It at least consists of one APEX and one ATLAS chip, but may contain several VORTEX and DUPLEX chips as well. There can be more than one card, all served by this one Chipset Driver MODULE.

CIV (Chipset Initialization Vector): Structure passed from the API to the Chipset driver during initialization; it contains parameters that identify the specific modes and arrangements of the physical CORE CARD being initialized.

CORE CARD: same as CHIPSET CARD.

Issue date: September 1999

DPR (Deferred Processing Routine): This function is installed as a task by each device driver, 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.

DSLAMs: Digital Subscriber Line Access Multiplexer.

FM: Fault management, one of OAM cell types. It includes AIS, RDI, CC.

HSS: High Speed Serial links.

ISR (Interrupt Service Routine): A common function in each Device Driver 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.

LB: LoopBack

LINE CARD: A circuit card containing S/UNI-DUPLEX device and other Loop-side interface devices. The line card is usually connected to the core card via a HSS link.

GDD (Global Driver Database): Structure that holds the Configuration Data for this MODULE.

MIV (MODULE Initialization Vector): Structure passed from the API to the MODULE during initialization, it contains parameters that identify the specific characteristics of the Chipset driver MODULE being initialized.

MODULE: All of the code that is part of this chipset driver, there is only ONE instance of this MODULE connected to ONE OR MORE VORTEX chipset cards.

OAM: Operation And Maintenance.

OAM flow: Information flow transferred through the network by the means of a dedicated channel supported by specific octets of the transmission systems for the physical layer and by specific ATM cells referred to as OAM cells for the ATM layer.

PM: Performance Management, one of OAM cell types.

Issue date: September 1999

RDI: Remote Defect Indication.

RTOS (Real Time Operating System): The host for this Chipset driver.

VC: Virtual Circuit connection, including VCC and VPC.

VCC: Virtual Channel or F5 Connection.

VPC: Virtual Path or F4 Connection.

WAN: Wide Area Network.

WAN CARD: A circuit card containing S/UNI-DUPLEX device and other WAN-side interface devices. The WAN card is usually connected to the core card via a HSS link.

Issue date: September 1999