PM7351

S/UNI™ - VORTEX

S/UNI-VORTEX

OCTAL SERIAL LINK MULTIPLEXER

DRIVER MANUAL

RELEASED

ISSUE 2: JULY 2000

REVISION HISTORY

Issue No.	Issue Date	Originator	Details of Change
Issue 1	July 1999	James Lamothe	Document created from *S/UNI-VORTEX Device Driver Design Specification* (PMC-981181 Issue 2)
Issue 2	July 2000	Kevin Murray	Added documentation for three API functions: - vortexHssSetState - vortexHssSetLogChnlAddrMap - vortexSetCtrlChnlBaseAddr

ABOUT THIS MANUAL

This manual describes the S/UNI-VORTEX device driver. It describes the driver's functions, data structures, and architecture. This manual focuses on the driver's interfaces to your application, real-time operating system, and to the S/UNI-VORTEX device. It also describes in general terms how to modify and port the driver to your software and hardware platform.

Audience

This manual was written for people who need to:

- Evaluate and test the S/UNI-VORTEX device
- Modify and add to the S/UNI-VORTEX driver's functions
- Port the S/UNI-VORTEX driver to a particular platform.

References

For more information about the S/UNI-VORTEX driver, see the release notes. For more information about the S/UNI-VORTEX device, see the following documents:

- *S/UNI-VORTEX (Octal Serial Link Multiplexer) Datasheet*: PMC-980582
- *S/UNI-VORTEX (Octal Serial Link Multiplexer) Short Form Datasheet*: PMC-990148)
- *S/UNI-VORTEX and S/UNI-VORTEX Technical Overview*: PMC-98102

Note: Ensure that you use the document that was issued for your version of the device and driver.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 DRIVER PORTING QUICK START

This section summarizes how to port the S/UNI-VORTEX device driver to your hardware and operating system (OS) platform.

Note: Because each platform and application is unique, this manual can only offer guidelines for porting the S/UNI-VORTEX driver.

The code for the S/UNI-VORTEX driver is organized into C source files. You may need to modify the code or develop additional code. The code is in the form of constants, macros, and functions. For the ease of porting, the code is grouped into source files (src) and include files (`inc`). The `src` files contain the functions and the `inc` files contain the constants and macros.

To port the S/UNI-VORTEX driver to your platform:

1. Port the driver's OS extensions (page 84):
   - Data types
   - OS-specific services
   - Utilities and interrupt services that use OS-specific services
2. Port the driver to your hardware platform (page 86):
   - Port the device detection function.
   - Port low-level device read-and-write macros.
   - Define hardware system-configuration constants.
3. Port the driver's application-specific elements (page 88):
   - Define the task-related constants.
   - Code the callback functions.
4. Build the driver (page 89).

For more information about porting the S/UNI-VORTEX driver, see section 7

2 DRIVER FUNCTIONS AND FEATURES

The following table lists the main functions and features offered by the S/UNI-VORTEX driver. You can alter these functions by modifying or adding to the driver's code.

Table 1: Driver Functions and Features

Functions	Description
Device Addition and Deletion (page 37)	These functions perform the following tasks: • Reset new devices • Allocate and initialize memory that will store context information for new devices • De-allocate device context memory during device shutdown
Device Initialization (page 41)	These functions initialize the S/UNI-VORTEX device and its associated context structures.
Device Diagnostics (page 44)	These functions write values to registers and read them back to verify the microprocessor's input and output interface with the device. They enable and disable internal and external loopback for the S/UNI-VORTEX device's high-speed serial (HSS) links. They also monitor the device's clocks.
HSS Link Configuration (page 47)	These functions configure the HSS links of the S/UNI-VORTEX device by programming the HSS link registers according to the parameters specified.
Cell Insertion and Extraction (page 53)	These functions insert cells into, and extract cells from, the S/UNI-VORTEX device control channels by manipulating the insert and extract FIFO control and status registers.
BOC Transmission and Reception (page 61)	These functions transmit and receive BOC on the HSS links. Writing to the transmit BOC registers transmits BOC. BOC is received by monitoring the RECEIVE BOC status-registers.

Statistics Collection (page 63)	These functions retrieve the device counts (including cells received, cells transmitted, errored cells received) for accumulation by the application.
Interrupt Servicing (page 19)	These functions clear the interrupts raised by the S/UNI-VORTEX device. Then they store the interrupt status for later processing by a 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 Interrupt Servicing Routine (ISR). In polling mode, the DPR process periodically services the interrupt status.
Indication Callbacks (page 67)	The DPR uses indication callback functions to notify the application of events in the S/UNI-VORTEX device and driver. These events include the reception of cells in the microprocessor extract cell FIFOs and the reception of valid BOC.

2.1 Driver Architecture

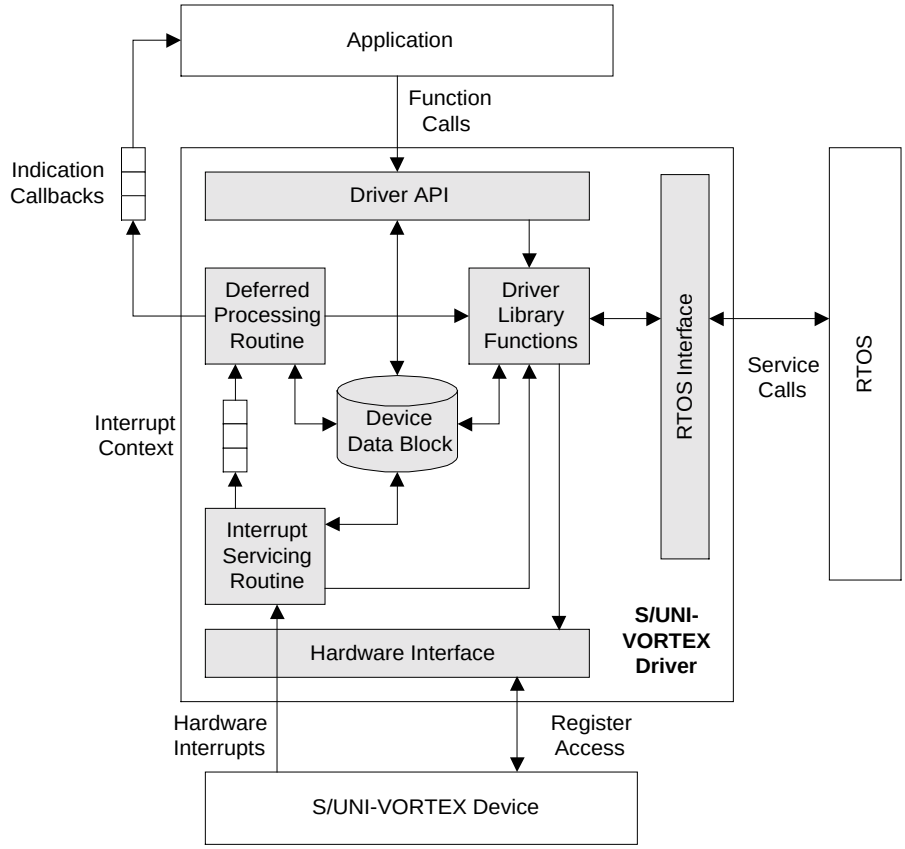
The driver includes seven main modules:

- Driver API module
- Real-time-OS interface module
- Hardware interface module
- Driver library module
- Device data-block module
- Interrupt-service routine module
- Deferred-processing routine module

For more information about these modules, see the following sections.

Figure 1 illustrates the architectural modules of the S/UNI-VORTEX driver.

Figure 1: Driver Architecture



2.1.1 Driver API Module

The driver's API is a collection of high level functions that can be called by application programmers to configure, control, and monitor the S/UNI-VORTEX device, such as:

- Initializing the device
- Validating device configuration
- Retrieving device status and statistics information.
- Diagnosing the device

The driver API functions use the driver library functions as building blocks to provide this system level functionality to the application programmer (see below).

The driver API also consists of callback functions that notify the application of significant events that take place within the device and driver, including cell and BOC reception.

2.1.2 Driver Real-Time-OS Interface Module

The driver's RTOS interface module provides functions that let the driver use RTOS services. The S/UNI-VORTEX driver requires the memory, interrupt, and preemption services from the RTOS. The RTOS interface functions perform the following tasks for the S/UNI-VORTEX device and driver:

- Allocate and de-allocate memory
- Manage buffers for the DPR
- Pause task execution
- Manage semaphores

Note: You must modify this code to suit your RTOS.

2.1.3 Driver Hardware-Interface Module

The S/UNI-VORTEX hardware interface provides functions that read from and write to S/UNI-VORTEX device-registers. The hardware interface also provides a template for an ISR that the driver calls when the device raises a hardware interrupt. You must modify this function based on the interrupt configuration of your system.

2.1.4 Driver Library Module

The driver library module is a collection of low-level utility functions that manipulate the device registers and the contents of the driver's DDB. The driver library functions serve as building blocks for higher level functions that constitute the driver API module. Application software does not normally call the driver library functions.

2.1.5 Device Data-Block Module

The DDB stores context information about the S/UNI-VORTEX device, such as:

- Device state

- Control information
- Initialization vector
- Callback function pointers
- Statistical counts

The driver allocates context memory for the DDB when the driver registers a new device.

2.1.6 Interrupt-Service Routine Module

The S/UNI-VORTEX driver provides an ISR called `vortexISR` that checks if there are any valid interrupt conditions present for the device. This function can be used by a system-specific interrupt-handler function to service interrupts raised by the device.

The low-level interrupt-handler function that traps the hardware interrupt and calls `vortexISR` is system and RTOS dependent. Therefore, it is outside the scope of the driver. An example implementation of such an interrupt handler (see page 81) as well as installation and removal functions (see page 80 and page 80) is provided as a reference. You can customize these example implementations to suit your specific needs.

See page 19 for a detailed explanation of the ISR and interrupt-servicing model.

2.1.7 Deferred-Processing Routine Module

The DPR provided by the S/UNI-VORTEX driver (`vortexDPR`) clears and processes interrupt conditions for the device. Typically, a system specific function, which runs as a separate task within the RTOS, executes the DPR.

See page 19 for a detailed explanation of the DPR and interrupt-servicing model.

2.2 Driver Software States

Figure 2 shows the software state diagram for the S/UNI-VORTEX driver. State transitions occur on the successful execution of the corresponding transition functions shown. State information helps maintain the integrity of the driver's DDB by controlling the set of device operations allowed in each state. Table 2 describes the software states for the S/UNI-VORTEX device as maintained by the driver.

Figure 2: Driver Software States

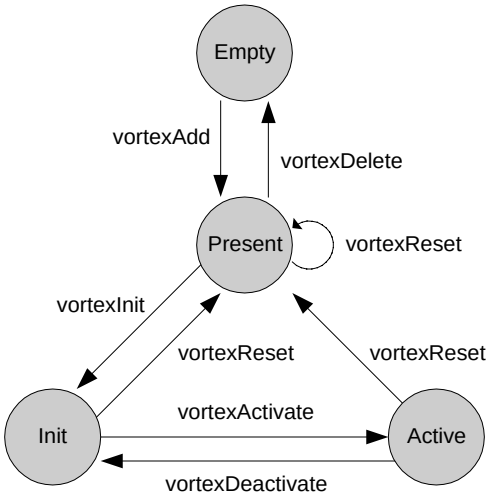


Table 2: Driver Software States

State	Description
Empty	The S/UNI-VORTEX device is not registered. This is the initial state.
Present	The driver has detected the S/UNI-VORTEX device and the drive has passed power-on self-tests. The driver has allocated memory to store context information about this device.
Init	An initialization vector passed by the application has successfully initialized the S/UNI-VORTEX device. The initialization parameters have been validated and the device has been configured by writing appropriate bits in the control registers of the device.
Active	The S/UNI-VORTEX device has been activated. This means that the device interrupts have been enabled and the device is ready for normal operation.

2.3 Processing Flows

This section describes some of the main processing flows of the S/UNI-VORTEX driver:

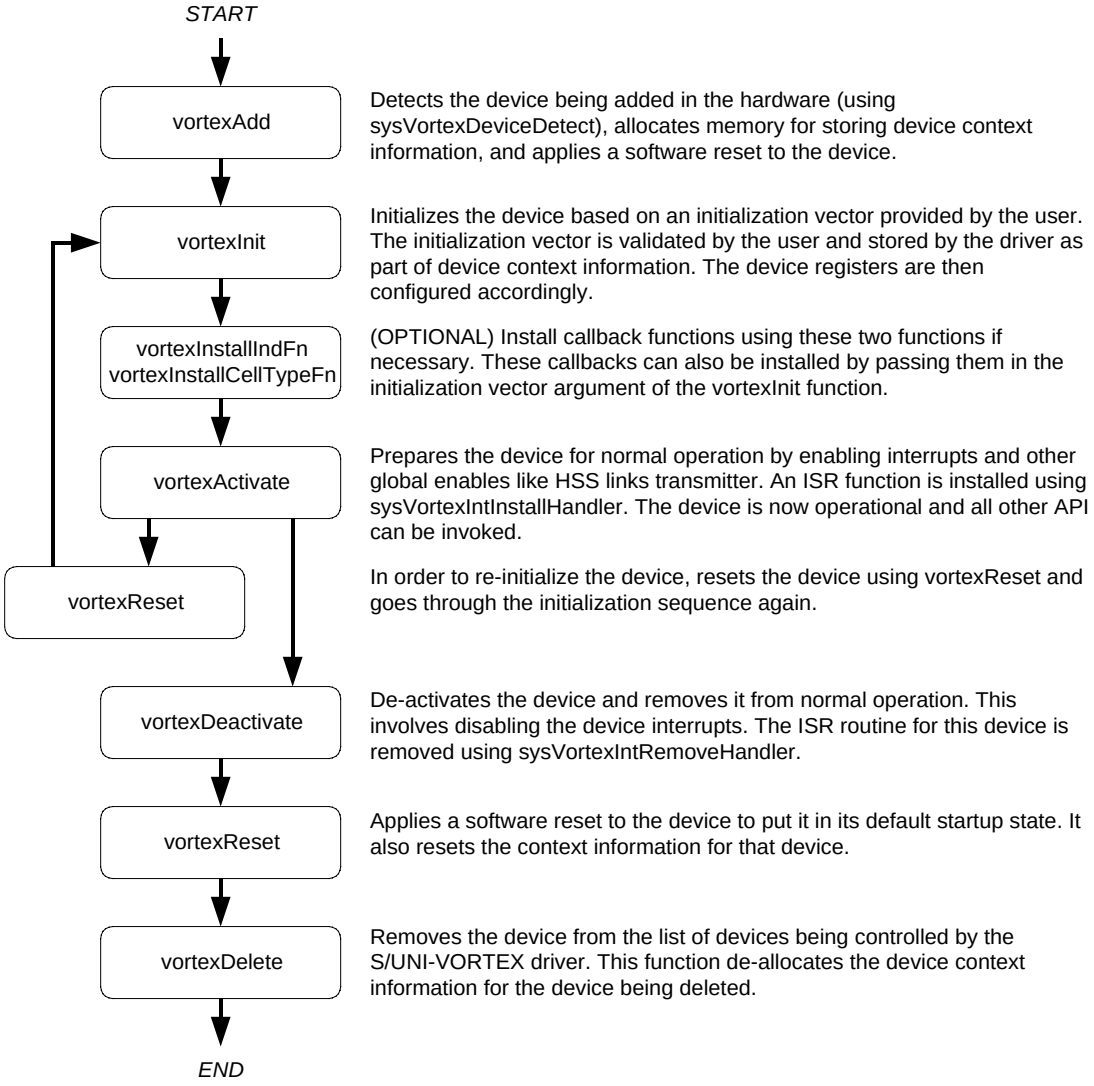
- Device initialization, re-initialization, and shutdown
- Cell extraction
- Interrupt servicing
- Polling servicing

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

2.3.1 Device Initialization, Re-initialization, and Shutdown

The following figure shows the functions and process that the driver uses to initialize, re-initialize, and shutdown the S/UNI-VORTEX device.

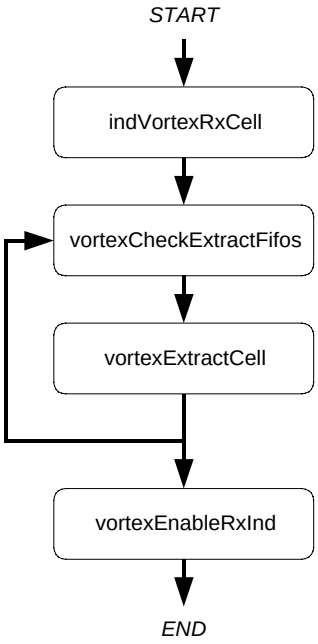
Figure 3: Device Initialization, Re-initialization, and Shutdown



2.3.2 Cell Extraction

The following figure shows the functions and process that the driver uses to extract cells from the S/UNI-VORTEX device.

Figure 4: Cell Extraction



The deferred processing routine invokes this indication callback function to inform the user of a cell reception. The indVortexRxCell function is typically implemented as a message queuing function that sends a message to another task (referred to henceforth as the cell reception task) that is dedicated to process received cells. The deferred processing routine also disables further RX indications.

The cell reception task now checks the status of the Extract FIFOs of the S/UNI-VORTEX device. This function determines which extract FIFOs have cells to be dequeued.

Cells are now dequeued by repeatedly invoking vortexExtractCell till the Extract FIFOs are empty. The message completion is detected by an End of Message bit in a cell type flag output from pCellTypeFn function. The funcion is installed by the user as a callback function. The Extract FIFOs are again checked to see if there are any more cells to be extracted.

After extracting all the cells from the Extract FIFOs of the S/UNI-VORTEX device, the cell reception task re-enables the RX indication for the device.

2.3.3 Interrupt Servicing

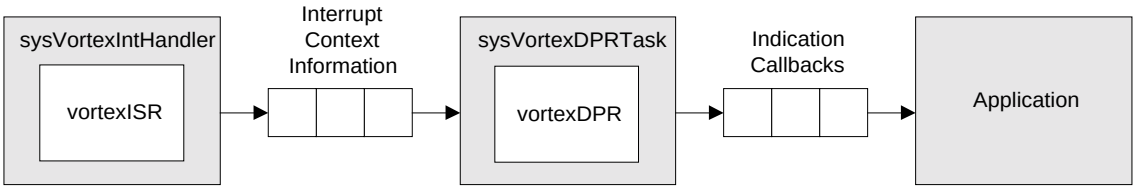
The S/UNI-VORTEX driver services device interrupts using an interrupt service routine (ISR) that traps interrupts and a deferred processing routine (DPR) that actually processes the interrupt conditions and clears them. This lets the ISR execute quickly and exit. Most of the time-consuming processing of the interrupt conditions is deferred to the DPR by queuing the necessary interrupt-context information to the DPR task. The DPR function runs in the context of a separate task within the RTOS.

Note: Since the DPR task processes potentially serious interrupt conditions, you should set the DPR task's priority higher than the application task interacting with the S/UNI-VORTEX driver.

The driver provides system-independent functions, `vortexISR` and `vortexDPR`. You must fill in the corresponding system-specific functions, `sysVortexISR` and `sysVortexDPR`. The system-specific functions isolate the system-specific communication mechanism (between the ISR and DPR) from the system-independent functions, `vortexISR` and `vortexDPR`.

Figure 5 illustrates the interrupt service model used in the S/UNI-VORTEX driver design.

Figure 5: Interrupt Service Model



Note: Instead of using an interrupt service model, you can use a polling service model in the S/UNI-VORTEX driver to process the device's event-indication registers (see page 22).

Calling vortexISR

An interrupt handler function, which is system dependent, must call `vortexISR`. But first, the low-level interrupt-handler function must trap the device interrupts. You must implement this function for your system. As a reference, an example implementation of the interrupt handler (`sysVortexIntHandler`) appears on page 81. You can customize this example implementation to suit your needs.

The interrupt handler that you implement (`sysVortexIntHandler`) is installed in the interrupt vector table of the system processor. Then it is called when one or more S/UNI-VORTEX devices interrupt the processor. The interrupt handler then calls `vortexISR` for each device in the active state. `vortexISR` reads from the HSS interrupt-status register and the miscellaneous interrupt-status register of the S/UNI-VORTEX.

Then `vortexISR` returns with the status information if a valid status bit is set. If a valid status bit is set, `vortexISR` also disables that device's interrupts. The `sysVortexIntHandler` then sends a message to the DPR task that consists of the device handles of all the S/UNI-VORTEX devices that had valid interrupt conditions.

Note: Normally you should save the status information for deferred processing by implementing a message queue. The interrupt handler uses `sysVortexIntHandler` to send the status information to the queue.

Calling vortexDPR

sysVortexDPRTask is a system specific function that runs as a separate task within the RTOS. You should set the DPR task's priority higher than the application task(s) interacting with the S/UNI-VORTEX driver. In the message-queue implementation model, this task has an associated message queue. The task waits for messages from the ISR on this message queue. When a message arrives, sysVortexDPRTask calls the DPR (vortexDPR). Then vortexDPR processes the status information and takes appropriate action based on the specific interrupt condition detected. The nature of this processing can differ from system to system. Therefore, vortexDPR calls different indication callbacks for different interrupt conditions.

Typically, you should implement these callback functions as simple message posting functions that post messages to an application task. However, you can implement the indication callback to perform processing within the DPR task context and return without sending any messages. In this case, ensure that the indication function does not call any API functions that change the driver's state, such as vortexDelete. Also, ensure that the indication function is non-blocking because the DPR task executes while S/UNI-VORTEX interrupts are disabled. You can customize these callbacks to suit your system. See page 67 for a description of the callback functions.

Note: Since the vortexISR and vortexDPR routines themselves do not specify a communication mechanism, you have full flexibility in choosing a communication mechanism between the two. A convenient way to implement this communication mechanism is to use a message queue, which is a service that most RTOSs provide.

You must implement the two system specific routines, sysVortexIntHandler and sysVortexDPRTask. When sysVortexIntInstallHandler is called for the first time, sysVortexIntHandler is installed in the interrupt vector table of the processor. The sysVortexDPRTask routine is also spawned as a task during this first time invocation of sysVortexIntInstallHandler. sysVortexIntInstallHandler also creates the communication channel between sysVortexIntHandler and sysVortexDPRTask. This communication channel is most commonly a message queue associated with sysVortexDPRTask.

Similarly, during removal of interrupts, the sysVortexIntHandler function is removed from the microprocessor's interrupt vector table and the task associated with sysVortexDPRTask is deleted.

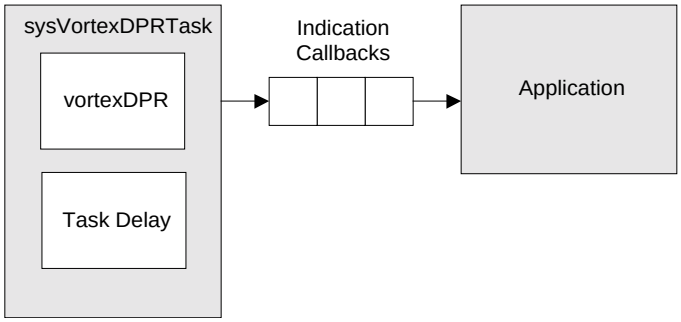
As a reference, this manual provides example implementations of the interrupt installation and removal functions. For more information about the interrupt removal function and prototype, see page 80. You can customize these prototypes to suit your specific needs.

2.3.4 Polling Servicing

Instead of using an interrupt service model, you can use a polling service model in the S/UNI-VORTEX driver to process the device's event-indication registers.

Figure 6 illustrates the polling service model used in the S/UNI-VORTEX driver design.

Figure 6: Polling Service Model



The polling service code includes some system specific code (prefixed by "`sysVortex`"), which typically you must implement for your application. The polling service code also includes some system independent code (prefixed by "`vortex`") provided by the driver that does not change from system to system.

In polling mode, `sysVortexIntHandler` and `vortexISR` are not used. Instead, a `sysVortexDPRTask` routine is spawned as a task processor when `sysVortexIntInstallHandler` is called for the first time.

In `sysVortexDPRTask`, the driver-supplied DPR (`vortexDPR`) is periodically called for each device in the active state. The `vortexDPR` reads from the HSS interrupt-status and miscellaneous interrupt-status registers of the S/UNI-VORTEX. If some valid status bits are set, it processes the status information and takes appropriate action based on the specific interrupt condition detected.

The nature of this processing can differ from system to system. Therefore, the DPR calls different indication callbacks for different interrupt conditions. You can customize these callbacks to fit your application's specific requirements. See page 67 for a description of these callback functions.

Similarly, during removal of polling service, the task associated with `sysVortexDPRTask` is deleted if none of S/UNI-VORTEX devices is activated.

3 DRIVER DATA STRUCTURES

The S/UNI-VORTEX driver uses several data structures. These structures help to:

- Control and store cell header information
- Configure the S/UNI-VORTEX device
- Identify the device's context
- Support interrupt processing
- Store indication callbacks

3.1 Cell Data Structures

This section describes the data structures that the driver uses to help control cell insertion and extraction. These structures serve as templates for received and transmitted cells.

3.1.1 Cell-Header Data Structure

The following structure stores cell header data.

Table 3: sVTX_CELL_HDR: Cell Header Structure

Member Name	Type	Description
u1UsrPrpnd[2]	UINT1	2 prepend bytes that you specify
u1Hdr[5]	UINT1	H1-H5 cell header bytes
u1UDF	UINT1	A field you define

3.1.2 Cell-Control Data Structure

The following structure controls cell insertion and extraction operations.

Table 4: sVTX_CELL_CTRL: Cell Control Structure

Member Name	Type	Description
u4Crc32Prev	UINT4	The CRC-32 value in the insert and extract CRC-32 accumulator registers after the previous cell was inserted or extracted. Used to preset the accumulator registers before inserting or extracting the next cell.
u4Crc32	UINT4	The CRC-32 value in the insert and extract accumulator registers after the current cell is inserted or extracted.

u1CellType	UINT1	A flag used by the driver to indicate that the cell extracted is the last cell or first cell of a message, and is CRC protected or not.
		• Bit 0:
		  • If 1, then CRC-32 on
		  • If 0, then CRC-32 off
		• Bit 1:
		  • If 1, then first cell
		  • If 0, then not first cell
		• Bit 2:
		  • If 1, then last cell
		  • If 0, then not last cell

3.2 Device-Configuration Data Structures

This section describes the data structures that the driver uses to initialize and configure the S/UNI-VORTEX device.

3.2.1 Initialization Data Structure

The device initialization function initializes the S/UNI-VORTEX device and its associated context structures. This involves reading an initialization vector. The driver validates this vector and then configures the S/UNI-VORTEX device accordingly.

The application sets the initialization vector before initializing a S/UNI-VORTEX device. The initialization vector contains configuration parameters that the driver uses to program the S/UNI-VORTEX device control-registers.

Note: The application must free the initialization vector memory.

Table 5: sVTX_INIT_VECTOR: Initialization Vector

Member Name	Type	Description
sRegInfo	sVTX_REGS	Contains the values that the driver will write to the control registers of the S/UNI-VORTEX device
indNotify	VTX_IND_CB_FN	Indication callback function called by the DPR when a significant event occurs in the driver software
indRxBOC	VTX_IND_CB_FN	Indication callback function called by the DPR to forward a received valid BOC to the application
indRxCell	VTX_IND_CB_FN	Indication callback function called by the DPR when the driver must read cells from the Extract FIFOs
pCellTypeFn	VTX_CELLTYPE_FN	A cell-type detector function that is used by the driver to determine if a cell extracted is the last or first of a particular message, and/or if it is CRC-32 protected
u4Reserved	UINT4	Placeholder for future use

3.2.2 Register Data Structure

The register data structure contains the initial values that the driver will write to the S/UNI-VORTEX device control-registers.

Table 6: sVTX_REGS: Device Registers

Member Name	Type	Description
u1MasterCfg	UINT1	Master configuration register
u1CtrlChnlBaseAddr[2]	UINT1	Control channel base address [2 bytes (LSB, MSB)]
u1DnstrmCellIntfCfg	UINT1	Downstream cell interface configuration
u1UpstrmCellIntfCfg	UINT1	Upstream cell interface configuration
sHssRegs	sVTX_HSS_REGS	HSS link control registers
sIntEnRegs	sVTX_INT_ENBLS	Interrupt enable registers

Table 7: sVTX_HSS_REGS: Device Registers

Member Name	Type	Description
u1RxHssCfg	UINT1	Receive HSS configuration
u1RxHssCellFilterCfgStat	UINT1	Receive-HSS cell-filtering configuration and status
u1UpstrmRRWt	UINT1	Upstream round-robin weight
u1LogChnlBaseAddrLsb	UINT1	Logical-channel base address
u1LogChnlAddrRngBaseAddrMsb	UINT1	Logical-channel address-range and logical-channel base-address MSB
u1DnstrmLogChnlFifoRdyLvl	UINT1	Downstream logical-channel FIFO-ready level

Member Name	Type	Description
u1TxHssCfg	UINT1	Transmit HSS configuration

3.3 Device-Context Data Structures

This section describes the data structures that the driver uses to store data about the S/UNI-VORTEX device and related devices.

3.3.1 Global Driver-Database Structure

The Global Driver Database (GDD) stores module level data, such as the number of devices that the driver controls and an array of pointers to the individual device context structures (DDBs).

Table 8: sVTX_GDD: Global Driver Database

Member Name	Type	Description
u1NumDevs	UINT1	Number of devices added
pDdb[VTX_MAX_NUM_DEVS]	sVTX_DDB*	Array of pointers to the individual DDBs
u4Reserved	UINT4	Reserved for future use

3.3.2 Device Data-Block Structure

The DDB contains device context data, such as:

- Device state
- Control data
- Initialization vector
- Callback function pointers

The driver allocates the DDB memory when the driver registers a new device. The memory is de-allocated when an existing device is deleted.

Table 9: sVTX_DDB: Device Data Block

Member Name	Type	Description
usrCtxt	VTX_USR_CTXT	This variable stores the device's role in the context of your system. The driver passes it as an input parameter when the driver calls an application callback.
pSysInfo	VOID *	Pointer to system-specific device information. For example, in PCI bus environments, the bus, device, function numbers, IRQ assignment etc.
u4BaseAddr	UINT4	Base address of the device
eDevState	eVTX_STATE	Device state, which can be one of the following enumerated type values: • VTX_EMPTY • VTX_PRESENT • VTX_INIT • VTX_ACTIVE
u1IntrProcEn	UINT1	1: Interrupt processing enabled 0: Interrupt processing disabled

Member Name	Type	Description
sInitVector	sVTX_INIT_VECTOR	Device configuration information passed by the application to the driver. The driver writes the appropriate S/UNI-VORTEX device registers based on the contents of this vector.
sIntEnbls	sVTX_INT_ENBLS	Maintains a snapshot of the current interrupt-enables registers for the S/UNI-VORTEX device
indNotify	VTX_IND_CB_FN	Indication callback function called by the DPR when a significant event occurs in the driver software
indRxBOC	VTX_IND_CB_FN	Indication callback function called by the DPR to forward a received valid BOC to the application
indRxCell	VTX_IND_CB_FN	Indication callback function called by the DPR when the driver must read cells from the Extract FIFOs
pCellTypeFn	VTX_CELLTYPE_FN	Indication callback function called by the driver when extracting a cell
sLogChnlAddrRng	sVTX_CHNL_ADDR_RNG	An array of VTX_NUM_HSS_LNKS elements. Each element contains the logical channel base address and range for a particular serial link.
sStatCounts	sVTX_STAT_COUNTS	Interrupt status counts per event

Member Name	Type	Description
lockId	VTX_SEM_ID	Semaphore for mutually exclusive access to sStatCounts
u4Reserved	UINT4	Placeholder for future use

3.4 Interrupt Data Structures

This section describes the data structures that the S/UNI-VORTEX driver uses to store interrupt context data for interrupt-enable bit-setting data.

3.4.1 Interrupt-Enable Data Structure

The interrupt-enable bit-setting data is stored in the following structure.

Table 10: sVTX_INT_ENBLS: Interrupt Enables

Member Name	Type	Description
u1MasterEn	UINT1	Master interrupt enable
u1ROOLEn	UINT1	ROOLE bit: Tracks changes in ROOLV bit. It is located in the clock monitor register.
u1DnstrmCellIntfEn	UINT1	Downstream-cell interface interrupt-enable
u1UpstrmCellIntfIntEn	UINT1	Upstream-cell interface interrupt-enable (CELLXFERRE bit)
u1MicroCellBufCtrl	UINT1	Microprocessor cell-buffer interrupt control
u1RxHssIntEn[8]	UINT1	Receive-HSS interrupt-enables (8 instances)

u1RxHssFifoOvr[8]	UINT1	Receive-HSS FIFO-overflow register (8 instances)
u1RxHssBocIntEn[8]	UINT1	Receive-HSS BOC interrupt-enables (8 instances)

3.4.2 Interrupt-Context Data Structure

The following structure passes interrupt context data from the interrupt servicing routine to the DPR.

Table 11: sVTX_INT_CTXT: Interrupt Context

Member Name	Type	Description
u1NumDevs	UINT1	Number of devices for which interrupts have to be processed
pu4DevHandles	UINT4 *	Array of size VTX_MAX_NUM_DEVS. The first u1NumDevs elements of this array contain the device handles for the devices for which interrupts have to be processed.

3.5 Statistical Count Structure

This section describes the data structure that the S/UNI-VORTEX driver uses to store statistical counts.

Table 12: sVTX_STAT_COUNTS: Statistical Counts

Member Name	Type	Description
CntPllErr	UINT4	Register 0x07, bit 3
CntBufFifoOvrRn	UINT4	Register 0x10, bit 5
CntBufFifoCrc32Err	UINT4	Register 0x10, bit 7

Member Name	Type	Description
CntUpStrmCellIfXferErr	UINT4	Register 0x0C, bit 7
CntDwnStrmCellIfParityErr	UINT4	Register 0x0B, bit 1
CntDwnStrmCellIfTxStCellErr	UINT4	Register 0x0B, bit 2
CntTxCellCntOvrnInd[8]	UINT4	Register 0x91, 0xB1, 0xD1, 0xF1, 0x111, 0x131, 0x151, 0x171 bit 5
CntTxCellCntUpdInd[8]	UINT4	Register 0x91, 0xB1, 0xD1, 0xF1, 0x111, 0x131, 0x151, 0x171 bit 6
CntTxFifoOvrRn[8]	UINT4	Register 0x8D, 0xAD, 0xCD, 0xED, 0x10D, 0x12D, 0x14D, 0x16D bit 0
CntRxTransFrmLos[8]	UINT4	Register 0x83, 0xA3, 0xC3, 0xE3, 0x103, 0x123, 0x143, 0x163 bit 0
CntRxTransFrmLcd[8]	UINT4	Register 0x83, 0xA3, 0xC3, 0xE3, 0x103, 0x123, 0x143, 0x163 bit 1
CntRxTransOfActv[8]	UINT4	Register 0x83, 0xA3, 0xC3, 0xE3, 0x103, 0x123, 0x143, 0x163 bit 2
CntRxNonZeroCrc[8]	UINT4	Register 0x83, 0xA3, 0xC3, 0xE3, 0x103, 0x123, 0x143, 0x163 bit 3
CntRxCellDelinXSync[8]	UINT4	Register 0x83, 0xA3, 0xC3, 0xE3, 0x103, 0x123, 0x143, 0x163 bit 4
CntRxCellHcsErrDetect[8]	UINT4	Register 0x83, 0xA3, 0xC3, 0xE3, 0x103, 0x123, 0x143, 0x163 bit 5

Member Name	Type	Description
CntRxCellCntsUpd[8]	UINT4	Register 0x83, 0xA3, 0xC3, 0xE3, 0x103, 0x123, 0x143, 0x163 bit 6
CntRxHldCntOvr[8]	UINT4	Register 0x83, 0xA3, 0xC3, 0xE3, 0x103, 0x123, 0x143, 0x163 bit 7
CntRxCellDatLstFifoOvrFlw[8]	UINT4	Register 0x88, 0xA8, 0xC8, 0xE8, 0x108, 0x128, 0x148, 0x168 bit 4
CntRxCellCtlLstFifoOvrFlw[8]	UINT4	Register 0x88, 0xA8, 0xC8, 0xE8, 0x108, 0x128, 0x148, 0x168 bit 5
CntRxBocValid[8]	UINT4	Register 0x99, 0xB9, 0xD9, 0xF9, 0x119, 0x139, 0x159, 0x179 bit 6
CntRxBocIdle[8]	UINT4	Register 0x99, 0xB9, 0xD9, 0xF9, 0x119, 0x139, 0x159, 0x179 bit 7
CountInterrupts	UINT4	Number of interrupts

4 APPLICATION INTERFACE FUNCTIONS

The driver's API is a collection of high level functions that application programmers can call to configure, control, and monitor S/UNI-VORTEX devices.

Note: These functions are not re-entrant. This means that two application tasks cannot invoke the same API at the same time. However, the driver protects it's data structures from concurrent accesses by the application and the DPR task.

The application interface also consists of callback functions. These callback functions notify the application of significant events that take place within the device and driver, such as:

- Occurrence of critical errors
- Reception of cells
- Reception of valid BOCs

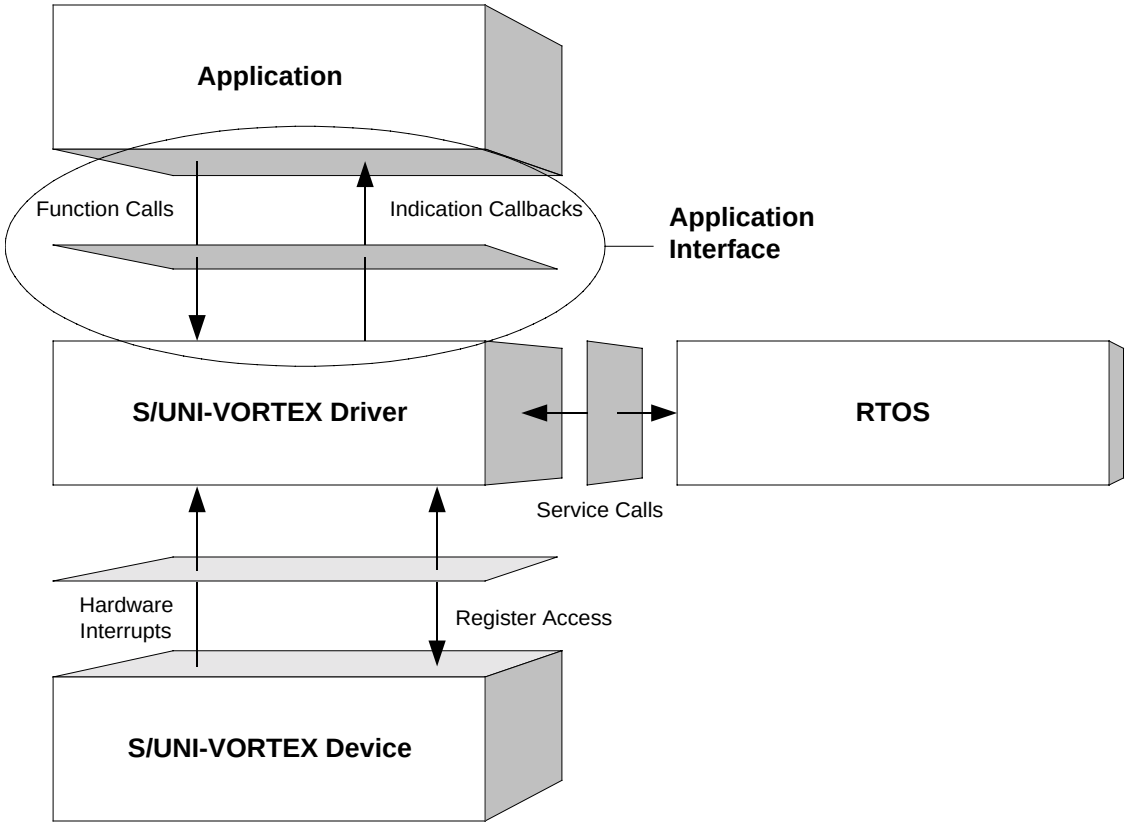
The vortexDPR routine invokes the indication callback functions. These execute in the context of the DPR task. Typically, these callback routines are implemented as simple message posting routines that post messages to an application task. However, the user can choose to implement the indication callback to perform processing within the DPR task context and return without sending any messages. In this case, ensure that the indication routine does not call any API function that changes the driver's state, such as vortexDelete.

The indication routine should be non-blocking because the DPR task executes while interrupts are disabled. The DPR task is also responsible for re-enabling device interrupts once the deferred processing is complete.

Many API functions change the device's state. For information about device states, see page 15.

Figure 7 illustrates the external interfaces defined for the S/UNI-VORTEX driver.

Figure 7: Application Interface



4.1 Driver Initialization and Shutdown

This section describes the API functions used to initialize and shutdown the driver's modules.

4.1.1 vortexModuleInit: Initializing Driver Modules

This function performs module level initialization of the device driver. This involves allocating memory for the GDD and initializing the data structure.

Valid States Not applicable

Side Effects None

Prototype `INT4 vortexModuleInit(VOID)`

Inputs None

Outputs None

Return Codes VTX_SUCCESS

VTX_ERR_MEM_ALLOC (memory allocation failure)

VTX_ERR_MODULE_ALREADY_INIT

4.1.2 vortexModuleShutdown: Shutting Down Driver Modules

This function performs module level shutdown of the driver. This involves deleting all devices controlled by the driver and de-allocating the GDD.

Valid States All states

Side Effects None

Prototype VOID vortexModuleShutdown(VOID)

Inputs None

Outputs None

Return Codes None

4.2 Device Addition, Reset, and Deletion

When you add a new S/UNI-VORTEX device, the driver's device-addition functions allocate memory to store context information for new devices. The driver also applies a software reset to the device. The device deletion function de-allocates device context memory during device shutdown.

4.2.1 vortexAdd: Adding Devices

This function detects the new device in the hardware and allocates memory for the DDB. Then it stores the device's role (within your system's context) and returns the pointer to the DDB as a handle back to your system. You should use the device handle to identify the device on which the driver will perform the operation.

Valid States VTX_EMPTY

Side Effects This function puts the device in the VTX_PRESENT state. The function applies a software reset to the device.

Prototype `INT4 vortexAdd(VTX_USR_CTXT usrCtxt, VORTEX *pVortex)`

Inputs `usrCtxt`: Pointer to context information (maintained by your system) for the device being added

Outputs `pVortex`: Pointer to the S/UNI-VORTEX device handle that contains context information maintained by the driver. The variable type, VORTEX, is actually the following type, which you define:

- `#define VORTEX (void *)`

This prevents the application from accessing the DDB directly.

Return Codes VTX_SUCCESS

VTX_ERR_INVALID_STATE (invalid device state)

VTX_ERR_DEV_NOT_DETECTED (device was not detected)

VTX_ERR_MEM_ALLOC (memory allocation failure)

VTX_ERR_BAD_REVISION (revision not supported)

4.2.2 vortexReset: Resetting Devices

This function applies a software reset to the S/UNI-VORTEX device. It also resets all of the device's context information in the DDB (except for the initialization vector, which it leaves unmodified). Typically, the driver calls this function during device shutdown, or before re-initializing the device with an initialization vector.

Valid States All states except VTX_EMPTY

Side Effects This function puts the device in the VTX_PRESENT state. Therefore, the driver must initialize the device after a reset.

Prototype `INT4 vortexReset(VORTEX vortex)`

Inputs `vortex`: Pointer to DDB that contains device context information maintained by the driver.

Outputs None

Return Codes `VTX_SUCCESS`

`VTX_ERR_INVALID_DEVICE` (invalid device handle)

4.2.3 vortexDelete: Deleting Devices

This function removes the specified device from the list of devices controlled by the S/UNI-VORTEX driver. Deleting a device involves de-allocating the DDB for that device.

Valid States `VTX_PRESENT`

Side Effects This function changes the device state to `VTX_EMPTY`.

Prototype `INT4 vortexDelete(VORTEX vortex)`

Inputs `vortex`: Pointer to device context information maintained by the driver

Outputs None

Return Codes `VTX_SUCCESS`

`VTX_ERR_INVALID_DEVICE` (invalid device handle)

`VTX_ERR_INVALID_STATE` (invalid device state)

4.3 Reading from and Writing to Devices

This section describes the API functions used to read from and write to S/UNI-VORTEX devices. Their tasks include reading from and writing to the registers of a device.

4.3.1 vortexRead: Reading from Device Registers

This function can read from a register of a specific S/UNI-VORTEX device by providing the register identifier. This function derives the actual address location based on the device handle and register identifier inputs. It then reads the contents of this address location using the system specific macro, `sysVortexRawRead`.

Prototype `INT4 vortexRead(VORTEX vortex, UINT2 u2RegId, UINT1 *pu1Val)`

Inputs `vortex`: Pointer to device context information

`u2RegId`: Register identifier

Outputs `pu1Val`: Register value

Return Codes `VTX_SUCCESS`

`VTX_ERR_INVALID_DEVICE` (invalid device handle)

`VTX_ERR_REG_RANGE` (invalid register identifier)

4.3.2 vortexWrite: Writing to Device Registers

This function can write to a register of a specific S/UNI-VORTEX device by providing the register identifier. This function derives the actual address location based on the device handle and register identifier inputs. It then writes the contents of this address location using the system specific macro, `sysVortexRawWrite`.

Prototype `INT4 vortexWrite(VORTEX vortex, UINT2 u2RegId, UINT1 u1Val)`

Inputs `vortex`: Pointer to device context information

`u2RegId`: Register identifier

`u1Val`: Value to be written

Outputs None

Return Codes VTX_SUCCESS

VTX_ERR_INVALID_DEVICE (invalid device handle)

VTX_ERR_REG_RANGE (invalid register identifier)

4.4 Device Initialization

This section describes the API functions used to initialize S/UNI-VORTEX devices. Their tasks include initializing the device based on the initialization vector passed by the application. They also install and remove the indication callback functions that vortexDPR calls.

4.4.1 vortexInit: Initializing Devices

This function initializes the device based on the initialization vector passed by the application. The driver validates this initialization vector and then stores it in the device's DDB. The driver then configures the device registers accordingly.

Valid States VTX_PRESENT

Side Effects This function puts the device in the VTX_INIT state.

Prototype
```
INT4 vortexInit(VORTEX vortex,
sVTX_INIT_VECT, sInitVector)
```

Inputs vortex: Pointer to DDB that contains device context information maintained by the driver

sInitVector: Initialization vector that the driver uses to program the device registers

Outputs None

Return Codes VTX_SUCCESS

VTX_ERR_INVALID_DEVICE (invalid device handle)

VTX_ERR_INVALID_STATE (invalid device state)

VTX_ERR_INVALID_INIT_VECTOR (invalid initialization vector)

4.4.2 vortexInstallIndFn: Installing Indication Callback Functions

This function installs the indication callback functions (which you define) that vortexDPR calls. The function pointer is stored in the device context structure (the DDB).

Valid States VTX_INIT

Side Effects None

Prototype
```
INT4 vortexInstallIndFn(VORTEX vortex,
eVTX_CB_TYPE eCbType, VTX_IND_CB_FN pCbFn)
```

Inputs vortex: Pointer to DDB that contains device context information maintained by the driver

eCbType: Identifies the callback being installed, which can be one of the following:

- VTX_CB_NOTIFY
- VTX_CB_RX_BOC
- VTX_CB_RX_CELL

pCbFn: Callback function that the driver is installing

Outputs None

Return Codes VTX_SUCCESS

VTX_ERR_INVALID_DEVICE (invalid device handle)

VTX_ERR_INVALID_CB_TYPE (invalid callback function type)

4.4.3 vortexRemoveIndFn: Removing Indication Callback Functions

This function removes the indication callback functions (which you define) that vortexDPR calls.

Valid States VTX_INIT

Side Effects The driver will no longer report events to the application.

Prototype `INT4 vortexRemoveIndFn(VORTEX vortex, eVTX_CB_TYPE eCbType)`

Inputs `vortex`: Pointer to DDB that contains device context information maintained by the driver

`eCbType`: Identifies the callback being installed, which can be one of the following:

- `VTX_CB_NOTIFY`
- `VTX_CB_RX_BOC`
- `VTX_CB_RX_CELL`

Outputs None

Return Codes `VTX_SUCCESS`

`VTX_ERR_INVALID_DEVICE` (invalid device handle)

`VTX_ERR_INVALID_CB_TYPE` (invalid callback function type)

4.5 Device Activation and Deactivation

This section describes the API functions used to activate and deactivate S/UNI-VORTEX devices. These functions set the device interrupts and other global enables.

4.5.1 vortexActivate: Activating Devices

This function activates the S/UNI-VORTEX device by preparing it for normal operation. This involves enabling device interrupts and other global enables (for example, the HSS link transmitter).

Valid States `VTX_INIT`

Side Effects Puts the device in `VTX_ACTIVE` state.

Prototype `INT4 vortexActivate(VORTEX vortex)`

Inputs `vortex`: Pointer to DDB that contains device context information maintained by the driver

Outputs None

Return Codes VTX_SUCCESS

VTX_ERR_INVALID_DEVICE (invalid device handle)

VTX_ERR_INVALID_STATE (invalid device state)

4.5.2 vortexDeactivate: Deactivating Devices

This function de-activates the S/UNI-VORTEX device and removes it from normal operation. This involves disabling device interrupts and other global disables (for example, the HSS link transmitter).

Valid States VTX_ACTIVE

Side Effects Puts the device in VTX_INIT state.

Prototype `INT4 vortexDeactivate(VORTEX vortex)`

Inputs vortex: Pointer to DDB that contains device context information maintained by the driver

Outputs None

Return Codes VTX_SUCCESS

VTX_ERR_INVALID_DEVICE (invalid device handle)

VTX_ERR_INVALID_STATE (invalid device state)

4.6 Device Diagnostics

This section describes the API functions used to diagnose the S/UNI-VORTEX device. Their tasks include:

- Verifying the correctness of the microprocessor's access to the device registers
- Enabling or disabling a diagnostic or line loopback on the specified HSS link
- Monitoring the activity of the device's clocks

4.6.1 vortexRegisterTest: Verifying Device Register Access

This function verifies the correctness of the microprocessor's access to the device registers by writing values to the writable registers and reading them back.

Valid States VTX_PRESENT

Side Effects Puts the device in the VTX_PRESENT state after the test. Therefore, the device should be re-initialized after calling this function.

Prototype
```
INT4 vortexRegisterTest(VORTEX vortex)
```

Inputs vortex: Pointer to DDB that contains device context information maintained by the driver

Outputs None

Return Codes VTX_SUCCESS

VTX_ERR_INVALID_DEVICE (invalid device handle)

VTX_FAILURE (test failed)

4.6.2 vortexLoopback: Enabling/Disabling Diagnostic or Line Loopback

This function enables or disables a diagnostic or line loopback on the specified HSS link.

Valid States All states except VTX_EMPTY

Side Effects None

Prototype
```
INT4 vortexLoopback(VORTEX vortex, UINT1
u1HssId, UINT1 u1LpbkType, UINT1 u1Enable)
```

Inputs vortex: Pointer to DDB that contains device context information maintained by the driver

u1HssLnkId: Serial link identifier. Valid identifiers are 0 through (VTX_NUM_HSS_LNKS – 1).

u1LpbkType: Type of loopback. It can be VTX_DIAG_LPBK or VTX_LINE_LPBK.

u1Enable: Loopback operation requested. It can be VTX_LPBK_SET or VTX_LPBK_RESET.

Outputs None

Return Codes VTX_SUCCESS

VTX_ERR_INVALID_DEVICE (invalid device handle)

VTX_ERR_INVALID_LPBK_TYPE (invalid loopback type)

VTX_ERR_INVALID_HSS_ID (invalid serial link identifier)

VTX_ERR_INVALID_FLAG (invalid loopback flag)

4.6.3 vortexGetClockStatus: Monitoring Device Clocks

This function monitors the activity of the S/UNI-VORTEX device clocks. It reads the contents of the clock monitor register and provides the status of each clock in a bit vector format. Call this function periodically to check if the clock signals are making low to high transitions.

Valid States All states except VTX_EMPTY

Side Effects None

Prototype
```
INT4 vortexGetClockStatus(VORTEX vortex,
UINT1 *pu1ClkStat)
```

Inputs vortex: Pointer to DDB that contains device context information maintained by the driver

Outputs `pu1ClkStat`: Contains the following bit vector that indicates the active/inactive status of the S/UNI-VORTEX device clocks. A one in the bit position indicates that the clock is active. A zero indicates that the clock is inactive.

- Bit 0: Transmit FIFO clock input (`TCLK`)
- Bit 1: Receive FIFO clock input (`RCLK`)
- Bit 2: Reference clock input (`REFCLK`)

Return Codes `VTX_SUCCESS`

`VTX_ERR_INVALID_DEVICE` (invalid device handle)

4.7 HSS Link Configuration

This section describes the API functions used to configure HSS links. Their tasks include:

- Retrieving the contents of the specified serial-link's configuration registers
- Configuring or modifying the contents of the specified serial-link's configuration registers
- Getting a snapshot of the state of the eight serial links for the specified device
- Retrieving the logical-channel address information for all serial links of the specified device

4.7.1 vortexHssGetConfig: Getting HSS-Link Configuration Information

This function retrieves the contents of the specified serial link's configuration registers. With one call, this function can retrieve the value of individual configuration registers as well as the entire configuration register set.

Valid States `VTX_INIT, VTX_ACTIVE`

Side Effects None

Prototype

```
INT4 vortexHssGetConfig(VORTEX vortex, UINT1
u1HssLnkId, eVTX_HSS_REG eHssRegId,
sVTX_HSS_REGS *psHssRegs)
```

Inputs

vortex: Pointer to DDB that contains device context information maintained by the driver

u1HssLnkId: Serial link identifier. Valid identifiers are 0 through (VTX_NUM_HSS_LNKS – 1)

eHssRegId: Specifies the register holding the value the driver will retrieve. It can be one of the following:

- VTX_RX_HSS_CFG
- VTX_RX_HSS_CELL_FILTER_CFGSTAT
- VTX_UPSTRM_RR_WT
- VTX_LOG_CHNL_BASE_ADDR_RANGE
- VTX_DNSTRM_LOG_CHNL_FIFO_RDY_LVL
- VTX_TX_HSS_CFG
- VTX_ALL_REGS

Note: The logical channel base address and address range are retrieved together. In addition, the driver can retrieve all configuration registers at once using VTX_ALL_REGS.

Outputs

psHssRegs: Contents of the specified HSS link control register(s) output by this function. These contents are valid only if the function returns VTX_SUCCESS. Further, only those fields of this structure are valid that have been requested using the input parameter, eHssRegId.

Return Codes

VTX_SUCCESS

VTX_ERR_INVALID_DEVICE (invalid device handle)

VTX_ERR_INVALID_STATE (invalid device state)

VTX_ERR_INVALID_HSS_ID (invalid serial link identifier)

VTX_ERR_INVALID_REG_ID (invalid register ID)

4.7.2 vortexHssSetConfig: Modifying HSS-Link Configuration Information

This function sets up or modifies the contents of the specified serial link's configuration registers. With one call, this function can set the value of individual configuration registers as well as the entire configuration register set.

Valid States VTX_INIT, VTX_ACTIVE

Side Effects None

Prototype

```
INT4 vortexHssSetConfig(VORTEX vortex, UINT1
u1HssLnkId, eVTX_HSS_REG eHssRegId,
sVTX_HSS_REGS *psHssRegs)
```

Inputs

vortex: Pointer to DDB that contains device context information maintained by the driver

u1HssLnkId: Serial link identifier. Valid identifiers are 0 through (VTX_NUM_HSS_LNKS – 1)

eHssRegId: Specifies the register with the value the driver will write. It can be one of the following:

- VTX_RX_HSS_CFG
- VTX_RX_HSS_CELL_FILTER_CFGSTAT
- VTX_UPSTRM_RR_WT
- VTX_LOG_CHNL_BASE_ADDR_RANGE
- VTX_DNSTRM_LOG_CHNL_FIFO_RDY_LVL
- VTX_TX_HSS_CFG
- VTX_ALL_REGS

Note: The logical channel base address and address range have to be set together. In addition, the driver can set all configuration registers at once using VTX_ALL_REGS.

psHssRegs: Contents of the specified HSS link control register(s) to be set. The only fields in this structure that will be set are those that the driver has requested using eHssRegId.

Outputs None

Return Codes VTX_SUCCESS

VTX_ERR_INVALID_DEVICE (invalid device handle)

VTX_ERR_INVALID_STATE (invalid device state)

VTX_ERR_INVALID_HSS_ID (invalid serial link identifier)

VTX_ERR_INVALID_REG_ID (invalid register ID)

VTX_ERR_INVALID_CONTENTS (values to be written are invalid)

4.7.3 vortexHssSetState: Setting Vortex HSS Link State

This function can be used to configure the VORTEX HSS link in a specified state.

Valid States VTX_INIT, VTX_ACTIVE

Side Effects None

Prototype

```
INT4 vortexHssSetState( VORTEX vortex, UINT1
u1HssLnkId, UINT1 u1State)
```

Inputs

vortex : pointer to DDB that contains device context information maintained by the driver.

u1HssLnkId : HSS link ID (0-7 valid)

u1State : State for HSS links:

```
VTX_DISABLE
VTX_ENABLE_INACTIVE
VTX_ENABLE_ACTIVE
```

Outputs None

Return Codes VTX_SUCCESS

VTX_ERR_INVALID_DEVICE (invalid device handle)

VTX_ERR_INVALID_STATE (invalid device state)

VTX_ERR_INVALID_HSS_ID (invalid serial link identifier)

VTX_ERR_INVALID_FLAG (invalid HSS link state)

4.7.4 vortexHssGetLinkInfo: Getting the State of HSS Links

This function gets a snapshot of the state (unconfigured, configured, disabled) of the eight serial links for the specified S/UNI-VORTEX device.

Valid States VTX_INIT, VTX_ACTIVE

Side Effects None

Prototype

```
INT4 vortexHssGetLinkInfo(VORTEX vortex,
eVTX_LNK_CFG_STATE *peLnkCfgState)
```

Inputs vortex: Pointer to DDB that contains device context information maintained by the driver

Outputs peLnkCfgState: Pointer to an array of VTX_NUM_HSS_LNKS elements. Each element contains the status of a serial link. The status value can be one of the following:

- VTX_LNK_LOOPBACK
- VTX_LNK_DISABLED
- VTX_LNK_ENABLED

Note: It is the responsibility of the calling function to allocate and free the array to which peLnkState points. The contents of the array are only valid if this function returns with VTX_SUCCESS.

Return Codes VTX_SUCCESS

VTX_ERR_INVALID_DEVICE (invalid device handle)

VTX_ERR_INVALID_STATE (invalid device state)

4.7.5 vortexHssGetLogChnlAddrMap: Getting Logical Channel Addresses

This function can retrieve the logical-channel address information (base address and address range) for all serial links of the specified device.

Valid States VTX_INIT, VTX_ACTIVE

Side Effects None

Prototype
```
INT4 vortexHssGetLogChnlAddrMap(VORTEX
vortex, sVTX_CHNL_ADDR_RNG *psAddrRng)
```

Inputs vortex: Pointer to DDB that contains device context information maintained by the driver

Outputs psAddrRng: Pointer to an array of VTX_NUM_HSS_LNKS elements. Each element contains the logical-channel base address and range for a particular serial link.

Note: It is the responsibility of the calling function to allocate and free the structure to which psAddrRng points. The contents of this structure are only valid if this function returns with VTX_SUCCESS.

Return Codes VTX_SUCCESS

VTX_ERR_INVALID_DEVICE (invalid device handle)

VTX_ERR_INVALID_STATE (invalid device state)

4.7.6 vortexHssSetLogChnlAddrMap: Setting Logical Channel Addresses

This function can be used to set the logical channel address information (base address and address range) for all serial links of the specified device.

Valid States VTX_INIT, VTX_ACTIVE

Side Effects None

Prototype
```
INT4 vortexHssSetLogChnlAddrMap(VORTEX vortex,sVTX_CHNL_ADDR_RNG *psAddrRng)
```

Inputs

vortex : pointer to DDB that contains device context information maintained by the driver.

psAddrRng : pointer to an array of VTX_NUM_HSS_LNKS elements; each element contains the logical channel base address and range for a particular serial link.

Note: It is the responsibility of the calling routine to allocate, assign, and free the structure psAddrRng

Outputs None

Return Codes VTX_SUCCESS

VTX_ERR_INVALID_DEVICE (invalid device handle)

VTX_ERR_INVALID_STATE (invalid device state)

4.7.7 vortexSetCtrlChnlBaseAddr: Setting Control Channel Base Addresses

This function can be used to set the control channel base address for the specified device.

Valid States VTX_INIT, VTX_ACTIVE

Side Effects None

Prototype `INT4 vortexSetCtrlChnlBaseAddr(VORTEX vortex, UINT2 u2BaseAddr)`

Inputs

`vortex` : pointer to DDB that contains device context information maintained by the driver.

`u2BaseAddr` : base address of the control channel.

bits 0-2: must be 0 or will be ignored

bits 3-11: programmed into the VORTEX register

bits 12-15: must be 0 or will be ignored

Outputs None

Return Codes `VTX_SUCCESS`

`VTX_ERR_INVALID_DEVICE` (invalid device handle)

`VTX_ERR_INVALID_STATE` (invalid device state)

4.8 Cell Insertion and Extraction

This section describes the API functions used to insert and extract cells. Their tasks include:

- Transmitting a cell on a specified HSS link 's control channel
- Extracting a cell received on a specified HSS link 's control channel
- Returning the contents of the microprocessor extract FIFO ready register
- Enabling the interrupt indication for a cell's reception
- Installing a callback function that determines the type of cell being extracted

4.8.1 vortexInsertCell: Inserting Cells into HSS Links

This function transmits a cell on a specified HSS link 's control channel. This function can send messages, which you define, over the HSS links. If the message is longer than the length of a cell's payload, then the application should segment the message into 48 byte cells. Call this function repeatedly until all the cells that constitute the message have been transmitted.

Optionally, a 32-bit CRC can protect messages. The CRC accumulates each time a cell belonging to the message is sent. For the last cell of the message (indicated by the application), the CRC is inserted into the last four bytes of the cell's payload.

Message interleaving (over different control channels and different circuits on same control channel) is allowed. For CRC-32 protected messages, message interleaving requires the application to save the intermediate CRC-32 value output by this function, if a cell has to be sent out on another control channel or another circuit on the same control channel.

Valid States VTX_ACTIVE

Side Effects You should give cell reception higher priority than cell transmission to prevent extract FIFO overflow. In other words, all cells of a received message should be extracted before switching context.

Prototype

```
INT4 vortexInsertCell(VORTEX vortex, UINT1
u1HssLnkId, sVTX_CELL_HDR *psCellHdr, UINT1
*pu1CellPyld, sVTX_CELL_CTRL *psCtrl)
```

Inputs

`vortex`: Pointer to DDB that contains device context information maintained by the driver

`u1HssLnkId`: Serial link identifier. Valid identifiers are 0 through (`VTX_NUM_HSS_LNKS` – 1).

`psCellHdr`: Pointer to the cell header structure that contains the two prepend bytes that you define (optional), H1-H4 bytes, and the H5 (optional) and UDF (optional) bytes. The driver uses the optional bytes based on the transmit HSS configuration register contents.

`pu1CellPyld`: Pointer to first byte of cell payload (48 contiguous bytes)

`psCtrl->u1CellType`: Contains three control-flag bits:

- Bit 0:
  - 0, no CRC protection required
  - 1, CRC protected
- Bit 1:
  - 0, non-first cell
  - 1, first cell
- Bit 2:
  - 0, non-last cell
  - 1, last cell
- For bits (2,1):
  - 01b, first cell of message
  - 10b, last cell of message
  - 11b, single cell message
  - 00b, intermediate cell

`psCtrl->u4Crc32Prev`: Used to restore previously saved CRC-32 value output by this function. Only applicable if bit 0 of `psCtrl->u1CrcFlg` is set.

Outputs

`psCtrl->u4Crc32`: Used to output CRC-32 value after writing a cell. The driver then passes this value back as an input parameter (`psCtrl->u4Crc32Prev`) for the next cell to be transmitted on the same control channel connection.

Return Codes VTX_SUCCESS

VTX_ERR_INVALID_DEVICE (invalid device handle)

VTX_ERR_INVALID_STATE (invalid device state)

VTX_ERR_INVALID_HSS_ID (invalid serial link identifier)

VTX_ERR_CELL_TX_BUSY (cell transmission failed)

4.8.2 vortexExtractCell: Extracting Cells from HSS Links

This function extracts a cell received on a specified HSS link 's control channel. This function also receives messages, which you define, that can span multiple cells. The application must call this function once for each cell that constitutes the message.

If the incoming message contains a CRC-32 field at the end, then the driver can perform a CRC check over the body of the message. The function also provides the header information of the cell to the calling function.

Valid States VTX_ACTIVE

Side Effects You should give cell reception a higher priority than cell transmission to prevent extract FIFO overflow. In other words, all cells of a received message should be extracted before switching context.

Prototype

```
INT4 vortexExtractCell(VORTEX vortex, UINT1
u1HssLnkId, sVTX_CELL_HDR *psCellHdr, UINT1
*pu1CellPyld, sVTX_CELL_CTRL *psCtrl)
```

Inputs vortex: Pointer to DDB that contains device context information maintained by the driver

u1HssLnkId: Serial link identifier. Valid identifiers are 0 through (VTX_NUM_HSS_LNKS – 1).

Outputs

`psCellHdr`: Pointer to the cell header-data received.

`pu1CellPyld`: Pointer to first byte of cell payload 48 contiguous bytes)

`psCtrl->u4Crc32`: Used to output CRC-32 value after reading a cell. The driver then passes this value back as an input parameter (`psCtrl->u4Crc32Prev`) for the next cell to be extracted on the same control channel connection.

`psCtrl->u1CrcFlg`: This is a control flag. Contains the following bit vector:

- Bit 0: CRC protection flag
- Bit 1: Flag for first cell of a CRC protected message
- Bit 2: Flag for last cell of a CRC protected message

Return Codes

`VTX_SUCCESS`

`VTX_ERR_INVALID_DEVICE` (invalid device handle)

`VTX_ERR_INVALID_STATE` (invalid device state)

`VTX_ERR_INVALID_HSS_ID` (invalid serial link identifier)

`VTX_ERR_CB_FN_NOT_INSTALLED` (callback function is not installed yet)

`VTX_ERR_CELL_DISCARDED` (cell reception failed)

`VTX_ERR_CELL_RX_CRC` (cell CRC error)

4.8.3 vortexCheckExtractFifos: Getting Contents of the Extract-FIFO-Ready Register

This function returns the contents of the microprocessor extract-FIFO-ready register. This function can check if there are any cells to extract from the extract FIFOs.

Valid States

`VTX_ACTIVE`

Side Effects

None

Prototype `UINT4 vortexCheckExtractFifos(VORTEX vortex, UINT1 *pu1FifoStat)`

Inputs `vortex`: Pointer to DDB that contains device context information maintained by the driver

Outputs `pu1FifoStat`: A bit vector of the status of the eight extract FIFOs. Each bit corresponds to a link ID. The following bit vector represents the state of each Extract FIFO.

For bit # (where # is 0 to 7):

- If value = 1, then HSS link # has at least one cell ready for extraction
- If value = 0, then no cells present at HSS link #

Return Codes `VTX_SUCCESS`

`VTX_ERR_INVALID_DEVICE` (invalid device handle)

`VTX_ERR_INVALID_STATE` (invalid device state)

4.8.4 vortexEnableRxCellInd: Enabling the Received Cell Indicator

This function enables the interrupt indication in the device for the reception of a cell. The application calls this function after it has responded to a previous indication by extracting all received cells (using multiple `vortexExtractCell` calls). The application task can now re-enable this indication and wait for the arrival of more cells.

Valid States `VTX_ACTIVE`

Side Effects None

Prototype `INT4 vortexEnableRxCellInd(VORTEX vortex)`

Inputs `vortex`: Pointer to DDB that contains device context information maintained by the driver

Outputs None

Return Codes VTX_SUCCESS

VTX_ERR_INVALID_DEVICE (invalid device handle)

VTX_ERR_INVALID_STATE (invalid device state)

4.8.5 vortexInstallCellTypeFn: Installing Callback Functions

This function can install a callback function (which you define) that the driver uses to determine the type of cell it is extracting. The detector function takes a cell header as the input argument and returns a cell type bit. It also returns the accumulated CRC value for the previous cells received for the same message.

Valid States VTX_INIT, VTX_ACTIVE

Side Effects None

Prototype
```
INT4 vortexInstallCellTypeFn(VORTEX vortex,
VTX_CELLTYPE_FN pCellTypeFn)
```

Inputs vortex: Pointer to DDB that contains device context information maintained by the driver

pCellTypeFn: pointer to the EOM detector function. The prototype of this function is:

- `UINT1 pCellTypeFn(UINT1 *pu1Hdr, UINT4 *pu4Crc32Prev)`

pu1Hdr is the pointer to the first byte of the cell header's eight bytes. pu4Crc32Prev is the accumulated CRC for the previous cells received for the same message.

Outputs None

Return Codes VTX_SUCCESS

VTX_ERR_INVALID_DEVICE (invalid device handle)

VTX_ERR_INVALID_STATE (invalid device state)

4.9 BOC Transmission and Reception

This section describes the API functions used to transmit and receive bit-oriented code (BOC). Their tasks include transmitting the specified BOC on the specified HSS link, and reading the BOC received on a serial link

4.9.1 vortexTxBOC: Transmitting BOC

This function transmits the specified BOC on the specified HSS link.

Valid States VTX_ACTIVE

Side Effects a "u1Code" of 000001b (Loopback activate) is a special case. When transmitting a loopback activate code. The RDIDIS bit in the Serial Link Maintenance register is set to logic 1 to prevent a premature loopback due to a preemptive remote defect indication (RDI) code being sent when a loss-of-signal or loss-of-cell-delineation event occurs.

Prototype

```
INT4 vortexTxBOC(VORTEX vortex, UINT1
u1HssLnkId, UINT1 u1Code)
```

Inputs vortex: Pointer to DDB that contains device context information maintained by the driver

u1HssLnkId: Serial link identifier. Valid identifiers are 0 through (VTX_NUM_HSS_LNKS – 1).

u1Code: BOC to be transmitted. Valid BOCs are:

- 000000b (RDI)
- 000001b (Loopback activate)
- 000010b (Loopback deactivate)
- 000011b (Remote reset activate)
- 000100b (Remote reset deactivate)
- 010001b to 111110b (defined by you)
- 111111b (idle code)

Outputs None

Return Codes VTX_SUCCESS

VTX_ERR_INVALID_DEVICE (invalid device handle)

VTX_ERR_INVALID_STATE (invalid device state)

VTX_ERR_INVALID_HSS_ID (invalid serial link identifier)

VTX_ERR_INVALID_BOC (invalid BOC)

4.9.2 vortexRxBOC: Reading Received BOC

This function can read BOC received on a serial link.

Valid States VTX_ACTIVE

Side Effects This function reads from the receive BOC status register. This function clears the status bits (IDLEI and BOCI) bits in the BOC status register.

Prototype

```
INT4 vortexRxBOC(VORTEX vortex, UINT1
u1HssLnkId, UINT1 *pu1Code)
```

Inputs vortex: Pointer to DDB that contains device context information maintained by the driver

u1HssLnkId: Serial link identifier. Valid identifiers are 0 through (VTX_NUM_HSS_LNKS – 1).

Outputs pu1Code: Pointer to BOC to be received. Valid BOCs are:

- 000000b (RDI)
- 000001b (Loopback activate)
- 000010b (Loopback deactivate)
- 000011b (Remote reset activate)
- 000100b (Remote reset deactivate)
- 010001b to 111110b (defined by you)
- 111111b (idle code)

Return Codes VTX_SUCCESS

VTX_ERR_INVALID_DEVICE (invalid device handle)

VTX_ERR_INVALID_STATE (invalid device state)

VTX_ERR_INVALID_HSS_ID (invalid serial link identifier)

VTX_ERR_INVALID_BOC (invalid BOC)

4.10 Statistics Collection

This section describes the API functions used to collect statistics about the device's HSS links. Their tasks include:

- Accumulating the received-cell count and header-check sequence (HCS) cell-error count for a specified HSS link
- Accumulating the transmitted-cell count for a specified HSS link
- Reading all the cell counts (transmit and receive) for all the serial links of the specified device
- Retrieving and resetting the statistical counts maintained by the driver

4.10.1 vortexGetHssLnkRxCounts: Accumulating Counts for Received Cells

This function accumulates the received cell and HCS cell errors counts for a specified HSS link. This function triggers an update of the receive HSS cell-counter registers and the receive-HSS HCS error-count register. It then reads the contents of these registers and returns the values read to the application.

To maintain a steady count of received cells and HCS cell errors, and to avoid overflow, the application should call this function at least every 30 seconds.

Valid States VTX_ACTIVE

Side Effects You should not use this function at the same time (in periodic polling fashion) as vortexGetAllHssLnkCounts because both functions trigger updates to the receive counters.

Prototype `INT4 vortexGetHssLnkRxCounts(VORTEX vortex, UINT1 u1HssLnkId, UINT4 *pu4RxCells, UINT4 *pu4HcsErrs)`

Inputs `vortex`: Pointer to DDB that contains device context information maintained by the driver

`u1HssLnkId`: Serial link identifier. Valid identifiers are 0 through (`VTX_NUM_HSS_LNKS` – 1).

Outputs `pu4RxCells`: Count of cells received

`pu4HcsErrs`: Count of HCS errored cells received

Return Codes `VTX_SUCCESS`

`VTX_ERR_INVALID_DEVICE` (invalid device handle)

`VTX_ERR_INVALID_STATE` (invalid device state)

`VTX_ERR_INVALID_HSS_ID` (invalid serial link identifier)

4.10.2 vortexGetHssLnkTxCounts: Accumulating Counts for Transmitted Cells

This function is accumulates the transmitted cell count for a specified HSS link. This function triggers an update of the transmit HSS cell-counter registers. It then reads the contents of these registers and returns the values read to the application.

To maintain a steady count of transmitted cells and to avoid overflow, the application should call this function at least every 30 seconds.

Valid States `VTX_ACTIVE`

Side Effects You should not use this function at the same time (in periodic polling fashion) as `vortexGetAllHssLnksCounts` because both functions trigger updates to the transmit counter.

Prototype `INT4 vortexGetHssLnkTxCounts(VORTEX vortex, UINT1 u1HssLnkId, UINT4 *pu4TxCells)`

Inputs vortex: Pointer to DDB that contains device context information maintained by the driver

`u1HssLnkId`: Serial link identifier. Valid identifiers are 0 through (`VTX_NUM_HSS_LNKS` – 1).

Outputs `pu4TxCells`: Count of cells transmitted

Return Codes `VTX_SUCCESS`

`VTX_ERR_INVALID_DEVICE` (invalid device handle)

`VTX_ERR_INVALID_STATE` (invalid device state)

`VTX_ERR_INVALID_HSS_ID` (invalid serial link identifier)

4.10.3 vortexGetAllHssLnkCounts: Accumulating Counts for All Cells

This function reads all the cell counts (transmit and receive) for all the serial links of the specified S/UNI-VORTEX device. This function triggers an update to all the counters of all the HSS links by writing a dummy value to the load performance meters register. It then reads the counters of all the serial links and returns the contents to the calling function.

To maintain a steady count of cells received, cells transmitted, and HCS errored cells on a per-link basis for all the serial links, and to avoid overflow, the application should call this function at least every 30 seconds.

Valid States `VTX_ACTIVE`

Side Effects You should not use this function at the same time (in periodic polling fashion) as `vortexGetHssLnkRxCounts` and `vortexGetHssLnkTxCounts` because both functions trigger updates to the same counters.

Prototype

```
INT4 vortexGetAllHssLnkCounts(VORTEX vortex,
UINT4 *pu4TxCellsArray UINT4
*pu4RxCellsArray UINT4 *pu4HcsErrsArray)
```

Inputs `vortex`: Pointer to DDB that contains device context information maintained by the driver

Outputs

`pu4TxCells`: Pointer to first element of an array of transmitted cell counts

`pu4RxCells`: Pointer to first element of an array of received cell counts

pu4HcsErrs: Pointer to first element of an array of HCS errored cell counts

Notes:

- Each array has `VTX_NUM_HSS_LNKS` elements.
- It is the responsibility of the calling function to allocate and free memory for each of these arrays.
- If a link is not configured, the driver will not read its counts and the value of the counts returned will be 0xffffffff.

Return Codes

`VTX_SUCCESS`

`VTX_ERR_INVALID_DEVICE` (invalid device handle)

`VTX_ERR_INVALID_STATE` (invalid device state)

`VTX_ERR_INVALID_HSS_ID` (invalid serial link identifier)

4.10.4 vortexGetStatisticCounts: Retrieving Driver Statistical Counts

This function retrieves the statistical counts maintained by the driver. It contains the counts for events and interrupts of the S/UNI-VORTEX device since the last call to reset statistic counts.

Valid States All states except `VTX_EMPTY`

Side Effects None

Prototype

```
INT4 vortexGetStatisticCounts(VORTEX vortex,
sVTX_STAT_COUNTS *psStatCounts)
```

Inputs `vortex`: Pointer to DDB that contains the count information maintained by the driver

Outputs `psStatCounts`: Contains statistical counts of events and interrupts

Return Codes `VTX_SUCCESS`

`VTX_ERR_INVALID_DEVICE` (invalid device handle)

4.10.5 vortexResetStatisticCounts: Resetting Driver Statistical Counts

This function resets the statistical counts maintained by the driver.

Valid States All states except `VTX_EMPTY`

Side Effects None

Prototype `INT4 vortexResetStatisticCounts(VORTEX vortex)`

Inputs `vortex`: Pointer to DDB that contains the count information maintained by the driver

Outputs None

Return Codes `VTX_SUCCESS`

`VTX_ERR_INVALID_DEVICE` (invalid device handle)

4.11 Indication Callbacks

The DPR uses indication callback functions to notify the application of events in the S/UNI-VORTEX device and driver. You must implement these functions to work within the inter-task communication and scheduling capabilities of your RTOS. Typically, the callback functions will run in the context of the DPR, not in the context of the application. Therefore, these functions must be non-blocking. They should use RTOS-based inter-task notification to pass callback information safely from the DPR to the application task.

4.11.1 indVortexNotify: Notifying the Application of Significant Events

This indication function notifies the application about the occurrence of a significant event in the hardware or the driver software. The `vortexDPR` function calls this function. This function should be non-blocking. Typically, the indication function sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements.

Prototype

```
VOID indVortexNotify(USR_CTXT usrCtxt,
sVTX_IND_BUF *pIndBuf)
```

Inputs

`usrCtxt`: Context information (maintained by your system) for the device

`pIndBuf`: Information regarding the indication. It consists of an event identifier that identifies the reported event. Uniquely supplemental information about the event. The application should use `vortexReturnIndBuf` to free the indication context structure.

Outputs None

Return Codes None

4.11.2 indVortexRxBOC: Notifying the Application of Received BOC

This indication function notifies the application about the reception of a valid BOC. The vortexDPR function calls this function. This function should be non-blocking.

Prototype

```
VOID indVortexRxBOC(USR_CTXT usrCtxt,
sVTX_IND_BUF *pIndBuf)
```

Inputs `usrCtxt`: Context information (maintained by your system) for the device

`pIndBuf`: Information regarding the indication. It consists of:

- `u1HssLnkId`: Serial link that received the BOC
- `u1BOC`: BOC received. It can be one of the following:
  - `000000b` (RDI)
  - `000001b` (loopback activate)
  - `000010b` (loopback deactivate)
  - `000011b` (remote reset activate)
  - `000100b` (remote reset deactivate)
  - `010001b` to `111110b` (defined by you)
  - `111111b` (idle code)

The application should use `vortexReturnIndBuf` to free the indication context structure.

Outputs None

Return Codes None

4.11.3 indVortexRxCell: Notifying the Application of Ready Extract-Cell-FIFOs

This indication function notifies the application of the reception of cells in the microprocessor extract cell FIFOs. The `vortexDPR` function calls this function. This function should be non-blocking. Typically, the indication function sends a message to another task with the event identifier and other context information. The task that receives this message can then extract the received cells using `vortexCheckExtractFifos` and `vortexExtractCell`.

Prototype
```
VOID indVortexRxCell(USR_CTXT usrCtxt,
sVTX_IND_BUF *pIndBuf)
```

Inputs `usrCtxt`: Context information (maintained by your system) for the device

`pIndBuf`: Information regarding the indication. Currently, the driver does not use it, so the driver passes a null pointer for now.

Outputs None

Return Codes None

5 REAL-TIME-OS INTERFACE FUNCTIONS

The driver's RTOS interface module provides functions and macros that let the driver use RTOS services. The S/UNI-VORTEX driver requires the following RTOS services:

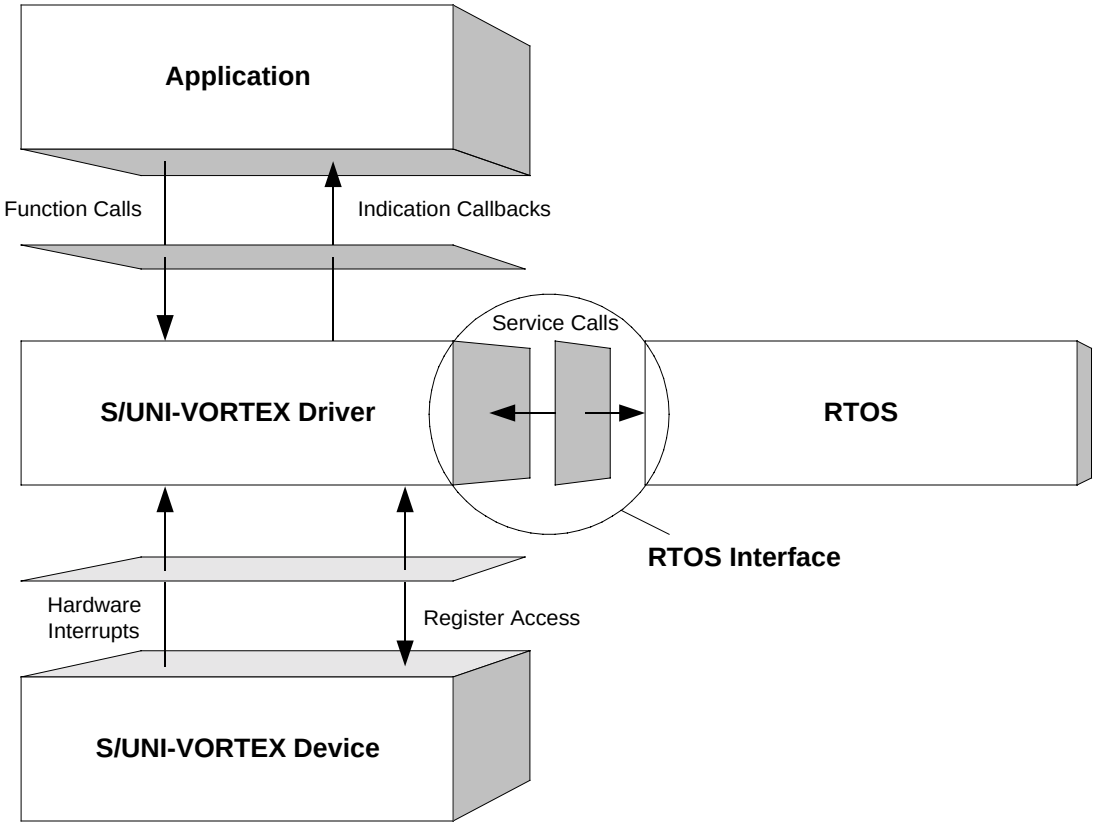
- Memory: Allocate and de-allocate
- Interrupts: Install and remove
- Preemption: Enable and disable

The driver may also require the following additional RTOS services depending on how you customize the code (for example, the ISR, the DPR, and so on). These services are:

- Timers: Create, delete, start and abort
- Tasks: Spawn and delete
- Message queues: Create and destroy queues, send and receive messages

Figure 8 illustrates the external interfaces defined for the S/UNI-VORTEX driver.

Figure 8: Real-Time OS Interface



5.1 Memory Allocation and De-allocation

This section describes the RTOS interface functions used to allocate and de-allocate memory.

5.1.1 sysVortexMemAlloc: Allocating Memory

This macro allocates a specified number of bytes.

Prototype `#define sysVortexMemAlloc(nbytes) malloc(nbytes)`

Inputs `nbytes`: Number of bytes to be allocated

Outputs None

Return Codes Pointer to first byte of allocated memory

NULL pointer (memory allocation failed)

5.1.2 sysVortexMemFree: De-allocating Memory

This macro de-allocates memory allocated by `sysVortexMemAlloc`.

Prototype `#define sysVortexMemFree(pu1First) free(pu1First)`

Inputs `pu1First`: Pointer to first byte of the memory region being de-allocated

Outputs None

Return Codes None

5.2 Buffer Management

This section describes the RTOS interface functions used to manage buffers for the DPR. Their tasks include getting a buffer for saving the context information for the indication callbacks, and returning the buffer after the application has received the context information.

5.2.1 vortexGetIndBuf: Getting DPR Buffers

This function gets a buffer that saves the context information for the indication callbacks called by the DPR.

Prototype `sVTX_IND_BUF *vortexGetIndBuf(VOID)`

Inputs None

Outputs None

Return Codes Pointer to indication context buffer

NULL pointer (buffer unavailable)

5.2.2 vortexReturnIndBuf: Returning DPR Buffers

This function returns the indication context buffer after the DPR has received the context information.

Prototype `VOID vortexReturnIndBuf(sVTX_IND_BUF *pBuf)`

Inputs `pBuf`: Pointer to indication context structure

Outputs None

Return Codes None

5.3 Timer Operations

This section describes the RTOS interface function used to suspend a task for a specified period.

5.3.1 sysVortexDelayFn: Delaying Functions

This function suspends execution of the calling function's task for a specified number of milliseconds.

Prototype `VOID sysVortexDelayFn(UINT4 u4Msecs)`

Inputs `u4Msecs`: Delay (in milliseconds)

Outputs None

Return Codes None

5.4 Semaphore Operations

This section describes the RTOS interface macros used to manage semaphores. Their tasks include:

- Creating a new mutual-exclusion semaphore
- Deleting a specified semaphore
- Taking and giving semaphores

5.4.1 sysVortexSemCreate: Creating Semaphores

This macro creates a new mutual-exclusion semaphore.

Prototype `#define sysVortexSemCreate() semMCreate()`

Inputs None

Outputs None

Return Codes semaphore ID

5.4.2 sysVortexSemDelete: Deleting Semaphores

This macro deletes a specified semaphore.

Prototype
```
#define sysVortexSemDelete(semId)
semDelete(semId)
```

Inputs semaphore ID

Outputs None

Return Codes None

5.4.3 sysVortexSemTake: Taking Semaphores

This macro takes a semaphore.

Prototype
```
#define sysVortexSemTake(semId)
semTake(semId)
```

Inputs semaphore ID

Outputs None

Return Codes None

5.4.4 sysVortexSemGive: Giving Semaphores

This macro gives a semaphore.

Prototype `#define sysVortexSemGive(semId) semGive(semId)`

Inputs semaphore ID

Outputs None

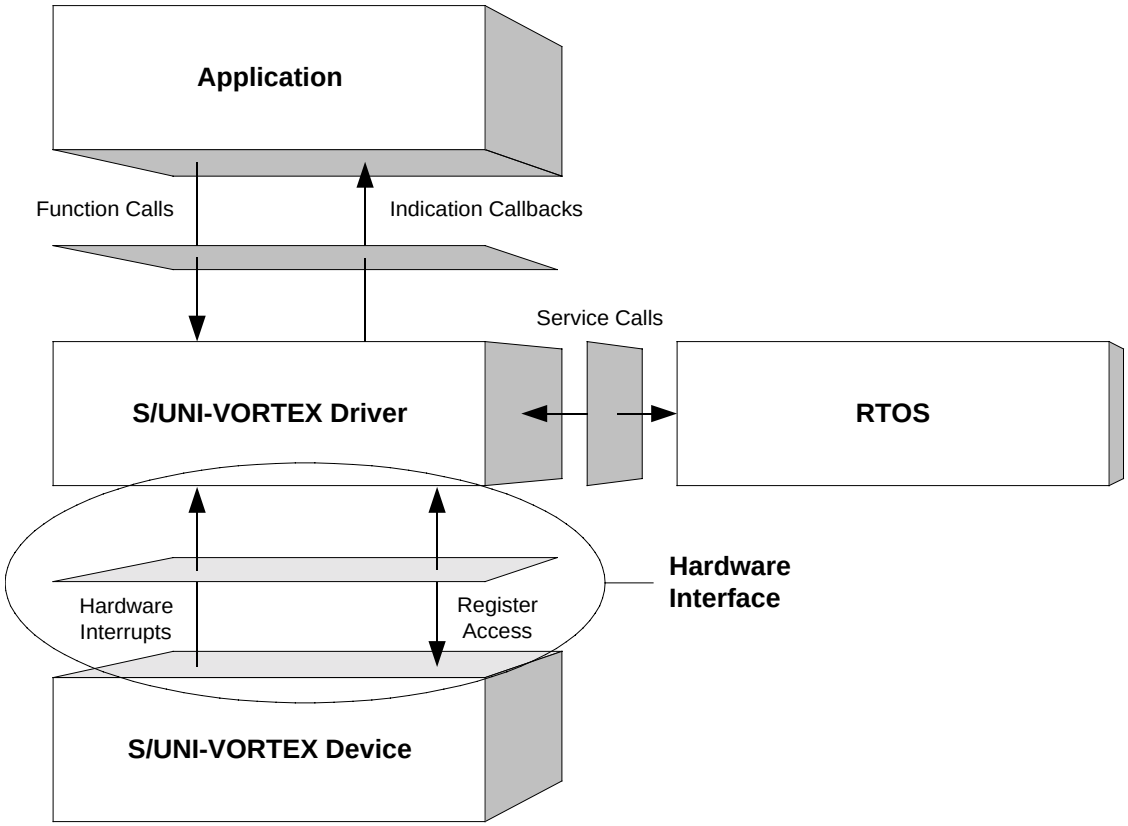
Return Codes None

6 HARDWARE INTERFACE FUNCTIONS

The S/UNI-VORTEX hardware interface provides functions and macros that read from and write to S/UNI-VORTEX device-registers. The hardware interface also provides a template for an ISR that the driver calls when the device raises a hardware interrupt. You must modify this function based on the interrupt configuration of your system.

Figure 9 illustrates the external interfaces defined for the S/UNI-VORTEX driver.

Figure 9: Hardware Interface



6.1 Device Register Access

This section describes the hardware interface functions used to read from and write to S/UNI-VORTEX device registers. Their tasks include reading and writing the contents of a specific address. It also includes getting the base address of the new device so that the driver can access the device register map to control it.

6.1.1 sysVortexRawRead: Reading from Register Address Locations

This low-level system-specific macro reads the contents of a specific register-address location. You should define this to reflect your system's addressing logic.

Prototype `#define sysVortexRawRead(addr, pval)`

Inputs `addr`: Address location to be read

`pval`: Value read

Outputs None

Return Codes Value read from the address location

6.1.2 sysVortexRawWrite: Writing to Register Address Locations

This low-level system-specific macro writes the contents to a specific register-address location. You should define this macro to reflect your system's addressing logic.

Prototype `#define sysVortexRawWrite(addr, val)`

Inputs `addr`: Address location to write

`val`: Value to be written

Outputs None

6.1.3 sysVortexDeviceDetect: Getting Device Base Addresses

This function gets the base address of the new device so that the driver can access it. The `vortexAdd` API function calls it.

Prototype
```
INT4 sysVortexDeviceDetect(VTX_USR_CTXT
usrCtxt, VOID **ppSysInfo, UINT4 *pu4BaseAddr)
```

Inputs `usrCtxt`: Context information (maintained by your system) for the device

Outputs `pu4BaseAddr`: Base address of device

`ppsysinfo`: Pointer to the sysinfo structure

Return Codes `VTX_SUCCESS`

`VTX_DEVICE_NOT_DETECTED`

6.2 Interrupt Servicing

This section describes the hardware interface functions used to provide hardware interrupt servicing. They install and remove the interrupt handlers and DPRs for the S/UNI-VORTEX devices. These functions depend on whether you implement the driver in interrupt mode or polling mode. In interrupt mode, their tasks include:

- Installing and removing the system-dependent interrupt-handler function (`sysVortexIntHandler`) and the DPR function (`sysVortexDPRTask`), creating a communication channel between the two, and adding the device to a list of devices for which interrupts will be serviced
- Removing the specified device from the list of devices for which interrupt processing will be done
- Calling `vortexISR` for each device for which interrupt processing is enabled
- Retrieving interrupt status information saved for it by the `sysVortexIntHandler` function, and calling the `vortexDPR` function for the appropriate device

In polling mode, these functions' tasks include:

- Spawning and removing the `sysVortexDPRTask` function
- Adding the device to a list of devices that need polling
- Polling the S/UNI-VORTEX device for interrupt status information and processing the interrupt status

The S/UNI-VORTEX driver provides a function called `vortexISR` that checks if there are any valid interrupt conditions present for a specified device. This function can be used by a system-specific interrupt-handler function to service interrupts raised by S/UNI-VORTEX devices.

The low-level interrupt handler function that traps the hardware interrupt and calls `vortexISR` is system and RTOS dependent. Therefore, it is outside the scope of the driver. As a reference, this manual provides an example implementation of such an interrupt handler (see `sysVortexIntHandler`) as well as installation and removal functions (see `sysVortexIntInstallHandler` and `sysVortexIntRemoveHandler`). You can customize these example implementations as per your specific requirements.

6.2.1 sysVortexIntInstallHandler: Installing Interrupt Service Functions

In interrupt mode, this function installs `sysVortexIntHandler` in the processor vector table, spawns the `sysVortexDPRTask` function as a task, and creates a communication channel (for example, a message queue) between the two. In addition, it adds the S/UNI-VORTEX device to a list of devices that need interrupt servicing.

In polling mode, this function spawns the `sysVortexDPRTask` function. This function periodically polls the device for interrupts and services the interrupts. It also adds the S/UNI-VORTEX device to a list of devices that need polling services.

Prototype `INT4 sysVortexIntInstallHandler(VORTEX vortex)`

Inputs `vortex`: Pointer to device context information

Outputs None

Return Codes `VTX_SUCCESS`

`VTX_ERR_INT_ALREADY`

`VTX_ERR_INT_INSTALL`

6.2.2 sysVortexIntRemoveHandler: Removing Interrupt Service Functions

In interrupt mode, this function removes the specified device from the list of devices that need interrupt processing. If this is the last active device, the function deletes the `sysVortexDPRTask` function and the associated message queue. It also removes the `sysVortexIntHandler` function from the processor's interrupt-vector table.

In polling mode, this function removes the specified device from the list of devices that need polling services. If this is the last active device, this function deletes `sysVortexDPRTask`.

Prototype	`VOID sysVortexIntRemoveHandler(VORTEX vortex)`
Inputs	`vortex`: Pointer to device context information
Outputs	None
Return Codes	None

6.2.3 sysVortexIntHandler: Calling vortexISR

In interrupt mode, this function calls `vortexISR` for each device with interrupt processing enabled. The driver calls this function when one or more S/UNI-VORTEX devices interrupt the microprocessor. If `vortexISR` detects at least one valid pending interrupt condition, then this function queues the interrupt context information for later processing by `sysVortexDPRTask`.

In polling mode, this function is not used.

Prototype	`VOID sysVortexIntHandler(UINT4 Irq)`
Inputs	`u4IntCtxt`: IRQ number of interrupt
Outputs	None
Return Codes	None

6.2.4 sysVortexDPRTask: Calling vortexDPR

In interrupt mode, the driver spawns this function as a separate task within the RTOS. It retrieves interrupt status information saved for it by the `sysVortexIntHandler` function and calls the `vortexDPR` function for the appropriate device.

In polling mode, the driver spawns this function as a separate task within the RTOS. It periodically calls the `vortexDPR` function for each active device.

Prototype `VOID sysVortexDPRTask(VOID)`

Inputs None

Outputs None

Return Codes None

7 PORTING THE DRIVER

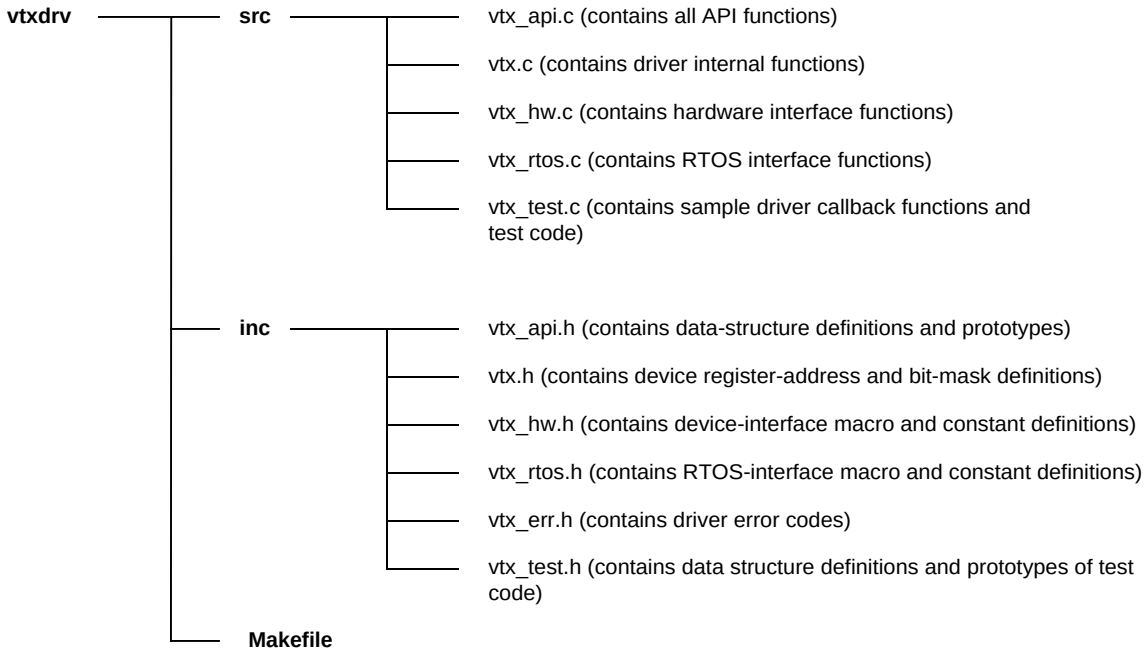
This section outlines how to port the S/UNI-VORTEX device driver to your hardware and OS platform.

Note: Because each platform and application is unique, this manual can only offer guidelines for porting the S/UNI-VORTEX driver.

7.1 Driver Source Files

The C source files listed in Figure 7-1 contain the code for the S/UNI-VORTEX driver. You may need to modify the code or develop additional code. The code is in the form of constants, macros, and functions. For the ease of porting, the code is grouped into source files (`src`) and include files (`inc`). The `src` files contain the functions and the `inc` files contain the constants and macros.

Figure 10: Driver Source Files



7.2 Driver Porting Procedures

The following steps summarize how to port the S/UNI-VORTEX driver to your platform. The following sections describe these steps in more detail.

Note: Because each platform and application is unique, this manual can only offer guidelines for porting the S/UNI-VORTEX driver.

To port the S/UNI-VORTEX driver to your platform:

1. Port the driver's OS extensions (page 84):
   - Data types
   - OS specific services
   - Utilities and interrupt services that use OS specific services
2. Port the driver to your hardware platform (page 86):
   - Port the device detection function.
   - Port low-level device read-and-write macros.
   - Define hardware system-configuration constants.
3. Port the driver's application-specific elements (page 88):
   - Define the task-related constants.
   - Code the callback functions.
4. Build the driver (page 89).

7.2.1 Porting the Driver's OS Extensions

The OS extensions encapsulate all OS specific services and data types used by the driver. The `vtx_rtos.h` file contains data types and compiler-specific data-type definitions. It also contains macros for OS specific services used by the OS extensions. These OS extensions include:

- Task management
- Message queues

- Timers
- Events
- Semaphores
- Memory Management

In addition, you may need to modify functions that use OS specific services, such as utility and interrupt-event handling functions. The `vtx_rtos.c` file contains the utility and interrupt-event handler functions that use OS specific services.

To port the driver's OS extensions:

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

2. Modify the OS specific services in `vtx_rtos.h`. Redefine the following macros to the corresponding system calls that your target system supports:

Service Type	Macro Name	Description
Memory	`sysVortexMemAlloc`	Allocates the memory block
	`sysVortexMemFree`	Frees the memory block
	`sysVortexMemCopy`	Copies the memory block from `src` to `dest`
Semaphore	`sysVortexSemCreate`	Creates the mutually exclusive semaphore
	`sysVortexSemDelete`	Frees the mutually exclusive semaphore
	`sysVortexSemGive`	Relinquishes the mutually exclusive semaphore
	`sysVortexSemTake`	Gets the mutually exclusive semaphore

3. Modify the utilities and interrupt services that use OS specific services in the `vtx_rtos.c`. The `vtx_rtos.c` file contains the utility and interrupt-event handler functions that use OS specific services. Refer to the function headers in this file for a detailed description of each of the functions listed below:

Service Type	Function Name	Description
Memory	`sysVortexMemSet`	Sets each character in the memory buffer
	`vortexGetIndBuf`	Gets a block of memory for the indication buffer
	`vortexReturnIndBuf`	Frees the indication buffer
Timer	`sysVortexDelayFn`	Sets the task execution delay in milliseconds
Interrupt	`sysVortexIntInstallHandler`	Installs the interrupt handler for the OS
	`sysVortexIntRemoveHandler`	Removes the interrupt handler from the OS
	`sysVortexIntHandler`	Interrupt handler for the S/UNI-VORTEX device
	`sysVortexDPRTask`	Deferred process routine for interrupts

7.2.2 Porting the Driver to a Hardware Platform

This section describes how to modify the S/UNI-VORTEX driver for your hardware platform.

Before you build the driver, ensure that you port the driver's OS extensions (page 84).

To port the driver to your hardware platform:

1. Modify the device detection function in the `vtx_hw.c` file. The function `sysVortexDeviceDetect` is implemented for a PCI platform. Modify it to reflect your specific hardware interface. Its purpose is to detect a S/UNI-VORTEX device based on a `UsrContext` input parameter. It returns two output parameters:

   - The base address of the S/UNI-VORTEX device
   - A pointer to the system-specific configuration information

2. Modify the low-level device read/write macros in the `vtx_hw.h` file. You may need to modify the raw read/write access macros (`sysVortexRawRead` and `sysVortexRawWrite`) to reflect your system's addressing logic.

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

#define	Description	Default
VTX_MEM_ADDR_RANGE	The assigned address memory range for each S/UNI-VORTEX device. Your system's memory map determines it.	0x800
VTX_ADAPTER_MAX_UNITS	The maximum number of S/UNI-VORTEX cards allowed in the system Note: The DSLAM architecture allows up to 16 S/UNI-VORTEX cards.	7
VTX_ADAPTER_MAX_DEVS	The maximum number of S/UNI-VORTEX devices on each card	2

7.2.3 Porting the Driver's Application-Specific Elements

Application specific elements are configuration constants used by the API for developing an application. This section describes how to modify the application specific elements in the S/UNI-VORTEX driver.

Before you port the driver's application-specific elements, ensure that you:

1. Port the driver's OS extensions (page 84).
2. Port the driver to your hardware platform (page 86).

To port the driver's application-specific elements:

1. Define the following driver task-related constants for your OS-specific services in file `vtx_rtos.h`:

#define	Description	Default
VTX_DPR_TASK_PRIORITY	Deferred Task (DPR) task priority	85
VTX_DPR_TASK_STACK_SZ	DPR task stack size, in bytes	4096
VTX_POLLING_DELAY	Constant used in polling task mode, this constant defines the interval time in millisecond between each polling action	10
VTX_TASK_SHUTDOWN_DELAY	Delay time in millisecond. When clearing the DPR loop active flag in the DPR task, this delay is used to gracefully shutdown the DPR task before deleting it.	10
VTX_MAX_DPR_MSGS	The queue message depth of the queue used for pass interrupt context between the ISR task and DPR task	10
VTX_MAX_IND_BUFSZ	Maximum indication buffer size in bytes	53

VTX_MAX_NUM_DEVS	The maximum number of S/UNI-VORTEX devices in the system	14

2. Code the callback functions according to your application. There are four sample callback functions in the `vtx_test.c` file. You can use these callback functions or you can customize them before using the driver. The driver will call these callback functions when an event occurs on the device. These functions must conform to the following prototypes:

   - `VOID indVortexNotify(VTX_USR_CTXT usrCtxt, sVTX_IND_BUF *psIndCtxt)`
   - `VOID indVortexRxBOC(VTX_USR_CTXT usrCtxt, sVTX_IND_BUF *psIndCtxt)`
   - `VOID indVortexCell(VTX_USR_CTXT usrCtxt, sVTX_IND_BUF *psIndCtxt)`
   - `UINT1 pCellTypeFn(UINT1 *pu1Hdr, UINT4 *pu4Crc32Prev)`

7.2.4 Building the Driver

This section describes how to build the S/UNI-VORTEX driver.

Before you build the driver, ensure that you:

1. Port the driver's OS extensions (page 84).
2. Port the driver to your hardware platform (page 86).
3. Port the driver's application-specific elements (page 88).

To build the driver:

1. Modify the makefile's compile-switch flag `VTX_CSW_INTERRUPT_MODE`. Set it to `1` for interrupt mode or `0` for polling mode.
2. Set the makefile's compile-switch flag `CSW_PV_FLAG` to `0`. This disables the test code specific to product verification.

3. Ensure that the directory variable names in the makefile reflect your actual driver and directory names.

4. Compile the source files and build the S/UNI-VORTEX API driver library using your make utility.

5. Link the S/UNI-VORTEX API driver library to your application code.

APPENDIX: CODING CONVENTIONS

This section describes the coding and naming conventions used in the implementation of the driver software. This section also describes the variable types.

Definition of Variable Types

The following table describes the variable types used by the S/UNI-VORTEX driver.

Table 13: Definition of Variable Types

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
VTX_USR_CTXT	Void *, pointer to user maintained device context
VORTEX	Void *, pointer to driver maintained device context

Naming Conventions

The names for variables, functions, and macros (but not constants) include prefixes that indicate their type. Variable, function, and macro names that contain multiple words have the first letter of each word capitalized.

Variables

The following table describes the prefixes used for the driver's variables.

Table 14: Variable Naming Conventions

Variable Type	Prefix	Example
UINT1	u1	u1Flag
UINT2	u2	u2Code
UINT4	u4	u4Val
INT1	i1	i1Flag
INT2	i2	i2Code
INT4	i4	i4Val
Structure variable	s	sCellHdr
Enumerated type	e	eHssRegId
Pointers	p	pu1Flag pi4Val psCellHdr peHssRegId
Pointer to a pointer	pp	ppu1Flag ppi4Val ppsCellHdr ppeHssRegId

Functions and Macros

The following table describes the prefixes used for the driver's functions and macros.

Table 15: Function and Macro Naming Conventions

Function Type	Prefix	Example Name
API functions	`vortex`	`vortexAdd`
Indication functions	`indVortex`	`indVortexRxCell`
System-specific functions and macros	`sysVortex`	`sysVortexIntHandler`

Definable Constants

You can define some constants using the "`#define`" command. These constants have names that are composed of all uppercase letters with underscores separating multiple words. An example is `VTX_NUM_HSS_LNKS`.

ACRONYMS

API: Application programming interface

DDB: Device data block

BOC: Bit oriented code

DPR: Deferred processing routine

GDD: Global driver database

HCS: Header check sequence

HSS link: High-speed serial link

ISR: Interrupt service routine

RTOS: Real-time operating system

INDEX

CONTACTING PMC-SIERRA, INC.

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

Tel: (604) 415-6000
Fax: (604) 415-6200

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