PM7326

S/UNI-APEX

ATM/PACKET TRAFFIC MANAGER AND SWITCH

DRIVER MANUAL

DOCUMENT ISSUE 2
ISSUED MAY 2000

INTRODUCTION TO THIS MANUAL

This manual describes the S/UNI-APEX 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-APEX device. It also describes in general terms how to modify and port the driver to your software and hardware platform.

Audience

This manual will help people who need to:

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

References

For more information about the S/UNI-APEX driver, see the driver release notes. For more information about the S/UNI-APEX device, see the documents listed in Table 1.

Table 1: Related Documents

Device	Document Name	Document Number
PM7326	ATM/Packet Traffic Manager and Switch Data Sheet	PMC-981224
	S/UNI-APEX Device Errata	PMC-990882
	S/UNI-APEX ATM/PACKET Traffic Manager and Switch Short Form Data Sheet	PMC-990146

Note: Ensure that you use the document that PMC-Sierra issued for your version of the device.

REVISION HISTORY

Issue No.	Issue Date	Details of Change
Issue 1	December 1999	Document created
Issue 2	April 2000	Added API functions to update congestion thresholds and scheduling parameters for direction, port, class and connection. Added API functions to install and reset multicasting callback function Added multicasting callback function to section on application callbacks Modified section on SAR Assist to include support for multicasting.

Legal Issues

PMC-991727 (P1), ref PMC-990236 (P2)

PMC-Sierra, Inc. has patents pending on the following S/UNI-APEX device and driver technologies:

- Loop port scheduler

- HSS protocol
- DSLAM architecture

Contacting PMC-Sierra

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
Technical Support: apps@pmc-sierra.com
Web Site: http: //www.pmc-sierra.com

TABLE OF CONTENTS

FIGURES

TABLES

1 DRIVER PORTING QUICK START

This section summarizes how to port the S/UNI-APEX device driver to your hardware and operating system (OS) platform. For more information about porting the S/UNI-APEX driver, see page 129.

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

The code for the S/UNI-APEX 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 source files contain the functions and the include files contain the constants and macros.

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

1. Port the driver’s hardware interface (page 130):
   - Data types
   - Port the device detection function.
   - Port low-level device read-and-write macros.
   - Define hardware system-configuration constants.
   - Port the busy-bit polling function.
   - Port the error tracing function (Optional).
2. Port the driver’s RTOS interface (page 132):
   - OS-specific services
   - Utilities and interrupt services that use OS-specific services
3. Port the driver’s application-specific elements (page 134):
   - Define the base value for the driver’s return codes.
   - Code the indication callback functions.
4. Build the driver (page 135).

2 DRIVER FUNCTIONS AND ARCHITECTURE

This section describes the functions and software architecture of the S/UNI-APEX device driver. It includes a discussion of the driver's external interfaces and its main components.

2.1 Driver Functions

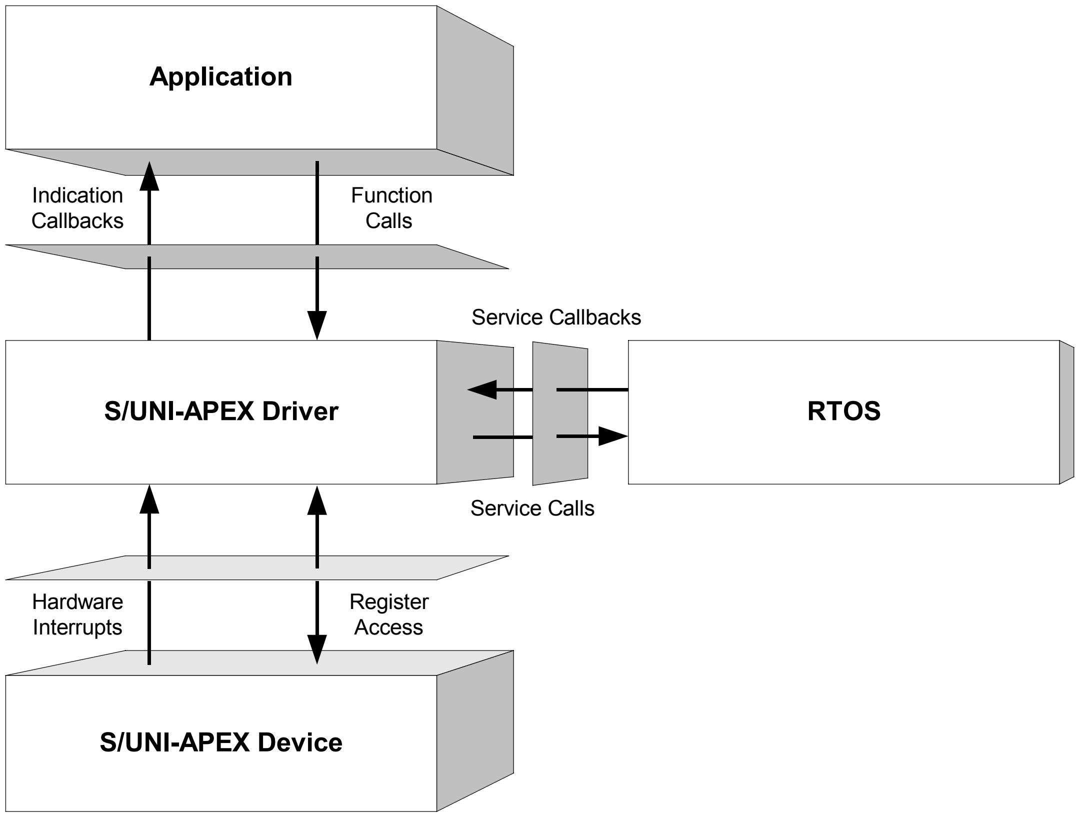
The S/UNI-APEX driver supports the following functions:

- Driver initialization and shutdown (see page 54)
- Profile management (see page 55)
- Device addition and removal (see page 61)
- Device register access (see page 62)
- Device diagnostics (see page 63)
- Device reset and initialization (see page 68)
- Device activation and deactivation (see page 70)
- Queue engine operations (see page 71)
- SAR-assist operations (see pages 38, 89 and 126)
- Loop port scheduler configuration (see pages 32 and 92)
- WAN port scheduler operation (see page 96)
- Statistic functions (see page 97)
- Interrupt service operations (see pages 26, 102 and 117)

2.2 Driver Interfaces

The driver's main function is to serve as an interface between the device and your application and operating system. Thus, the driver itself interfaces with the device, the application, and the operating system. Figure 1 illustrates the external interfaces defined for the S/UNI-APEX device driver.

Figure 1: Driver Interfaces



Application Programming Interface

The driver's API is a collection of high-level functions that application programmers can call to perform the following tasks (and many others):

- Initialize the device
- Validate the device's configuration
- Retrieve device status and statistics information
- Diagnose 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, such as cell and frame transmission/reception and error events.

Real-Time OS Interface

The driver's RTOS interface module consists of functions that the driver calls so that the driver can use RTOS services. These services include

- Memory allocation and de-allocation
- Semaphore operations
- Timer operations

The RTOS interface also includes service callback functions. The driver installs these service callbacks using RTOS service calls that install interrupt handler routines. The RTOS invokes these service callbacks when an interrupt occurs or a timer expires.

Note: You must modify the RTOS interface code according to your RTOS environment.

Hardware Interface

The S/UNI-APEX hardware interface module consists of functions/macros that read from and write to the S/UNI-APEX device-registers. It also consists of some system-specific constants that you will need to define. (For example, the maximum number of S/UNI-APEX devices to be controlled by the driver). The hardware interface also provides a template for an ISR that the driver calls when the device raises a device interrupt. You must modify this template based on the interrupt configuration of the application.

2.3 Main Driver Components

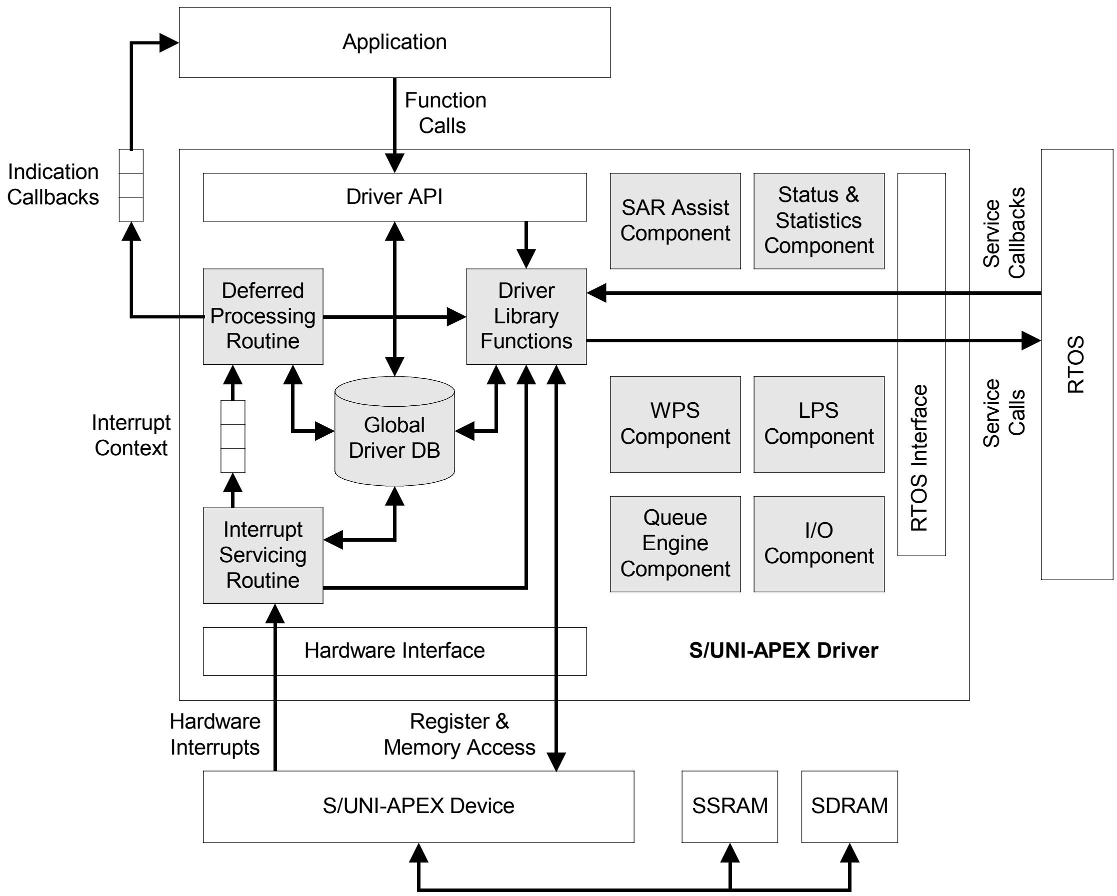
Figure 2 illustrates the top-level architectural components of the S/UNI-APEX device driver. This applies in both polled and interrupt-driven operation. In polled operation, the driver calls the ISR periodically. In interrupt operation, the interrupt directly triggers the ISR.

The driver includes ten main components:

- Global driver database
- Interrupt service routine
- Deferred processing routine
- Driver library
- Queue engine
- Loop port scheduler
- WAN port scheduler
- Segmentation and re-assembly assist component
- Input/output component

- Status and statistics component

Figure 2: Driver Architecture



Global Driver Database

The Global Driver Database (GDD) is the main data structure employed by the S/UNI-APEX device driver. It serves as a central repository for driver data. The driver allocates the GDD during driver initialization. One of the main components of the GDD is an array of pointers to per-device context structures called Device Data Blocks (DDBs).

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

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

Interrupt Service and Deferred Processing Routines

The device driver provides an interrupt service routine (ISR) for each of the device interrupt outputs. When the device interrupts the microprocessor, these ISRs store the interrupt context information and clear the interrupt conditions.

The ISR routines provided by the driver simply retrieve context information. This allows the routines to be compact and efficient. The interrupt context retrieved by these routines is saved for deferred processing. This processing occurs in the context of separate tasks within the RTOS.

The driver provides a deferred processing routine (DPR) that can run as a separate task. The DPR processes the interrupt context information and invokes callbacks, which you define, to inform the application when specific interrupt events have occurred. The driver supports two modes for servicing interrupts:

- Asynchronous interrupt servicing
- Synchronous polling

For more information about the DPR and interrupt-servicing model, see page 26.

Driver Library

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

Queue Engine

The queue engine controls the device's queue engine functions. These functions include:

- Setting up and tearing down ports
- Setting up and tearing down classes (loop, WAN and uP)
- Setting up and tearing down connections
- Setting up and tearing down shapers

For more information about the queue engine, see pages 29 and 71.

WAN Port Scheduler

The WAN port scheduler (WPS) schedules packet transmissions to the four WAN ports. To fairly and efficiently service these ports, the WPS uses the port weight table; this resides in the WPS internal-context memory. When you configure WAN ports, you must assign weights to them so that your application services the high-bandwidth ports more often than low-bandwidth ports. The WPS provides functions that set and retrieve the port weights assigned to the WAN ports.

Loop Port Scheduler

The loop port scheduler (LPS) controls the S/UNI-APEX loop port scheduler. This component manipulates the loop port scheduler's internal context memory (polling sequence and polling weight tables) so that the driver services the S/UNI-APEX device's loop ports fairly and efficiently.

For more information about the LPS, see pages 32 and 92.

Segmentation and Re-assembly Assist Component

The segmentation and re-assembly (SAR) assist component performs the insertion/extraction of cells and AAL5 frames from the microprocessor interface. This component uses the SAR assist features of the S/UNI-APEX device to perform these functions. The SAR transmit task injects cells or frames into the device. The SAR receive task extracts cells or frames from the device. They both typically run as separate tasks within the RTOS.

Note: The SAR assist component is not a full-fledged AAL5 SAR implementation. It does not perform automatic retransmission or error correction.

The SAR Assist module also provides support for multicasting cells or frames. Multicasting is defined as forwarding cells or frames, received on an incoming connection, to multiple outgoing connections.

For more information about the SAR assist component, see pages 38 and 89.

Input/Output Component

The input/output component provides low-level access to the device registers and the context memories. It uses the memory port interface to provide context-memory access. This component provides routines to perform read, write, and mask write operation on the context memory apertures.

The input/output component also maintains an image of the context memory in its host memory. This image only mirrors the configuration and control fields in the context memory. This image minimizes the number of indirect accesses through the memory port (which affects the device and overall system performance).

The context memory image is optional. You can compile the driver so that it does not use the context memory image. You may choose to use this option when memory resources in the system are limited.

Statistics Component

The statistics component consists of functions that retrieve statistical and congestion counts accumulated by the device.

2.4 Software States

Figure 3 shows the software state diagram for the S/UNI-APEX 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-APEX device as maintained by the driver.

Figure 3: Driver Software States

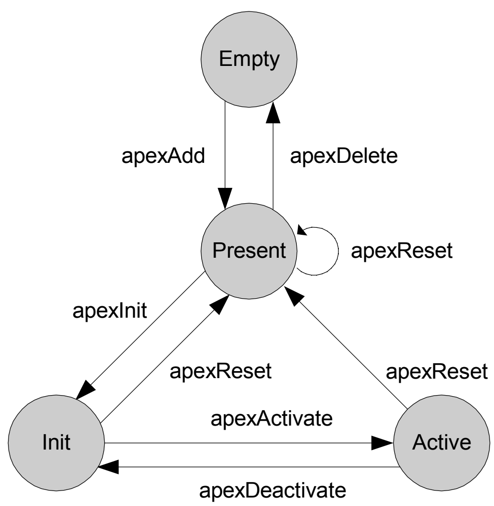


Table 2: Driver Software States

State	Description
APEX_EMPTY	The S/UNI-APEX device is not registered. This is the initial state.

State	Description
APEX_PRESENT	The driver has detected the S/UNI-APEX device and the device has passed power-on self-tests. A software reset has been applied to the device. The driver has allocated memory to store context information about this device.
APEX_INIT	An initialization vector passed by the application has successfully initialized the S/UNI-APEX device. The driver has validated the initialization parameters, and it has configured the device by writing appropriate bits in the control registers of the device.
APEX_ACTIVE	The driver has activated the S/UNI-APEX device. This means that the driver has enabled the device interrupts and SAR processing. The device is ready for normal operation.

2.5 Process Flows

This section describes two of the main processing flows of the S/UNI-APEX driver:

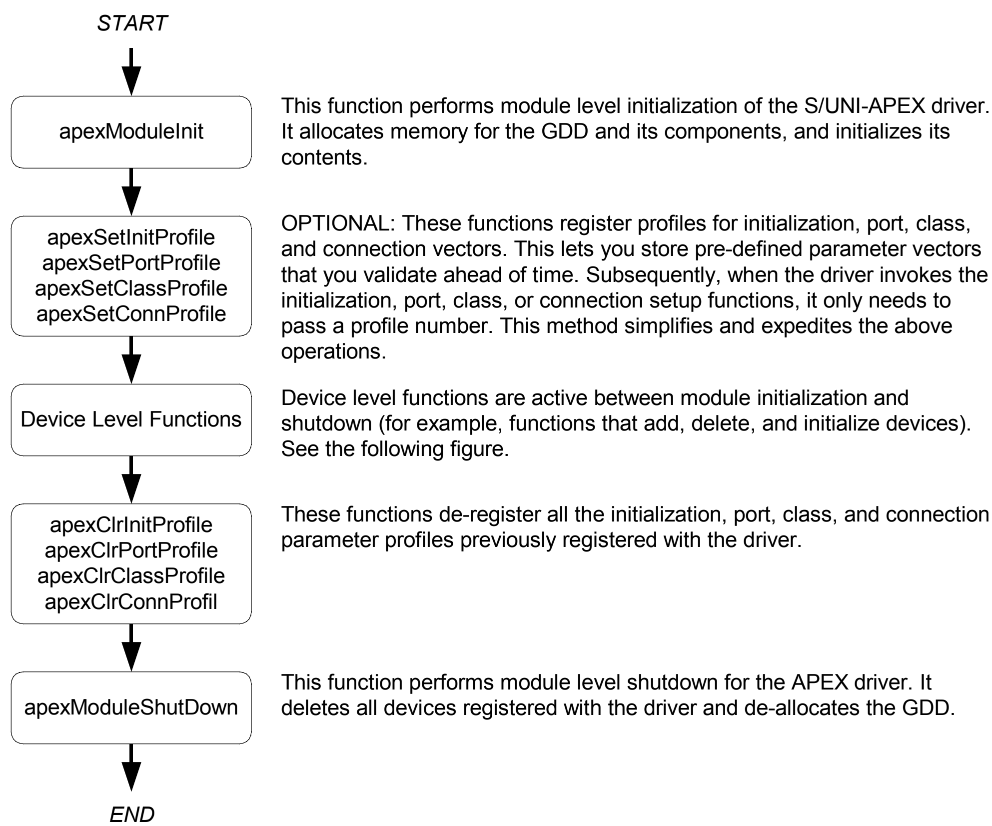
- Driver initialization and shutdown
- Device addition and deletion

The following flow diagrams 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 application must invoke the driver API.

Driver Initialization and Shutdown

The following figure shows the functions and processes that the driver uses to initialize and shutdown the S/UNI-APEX driver components.

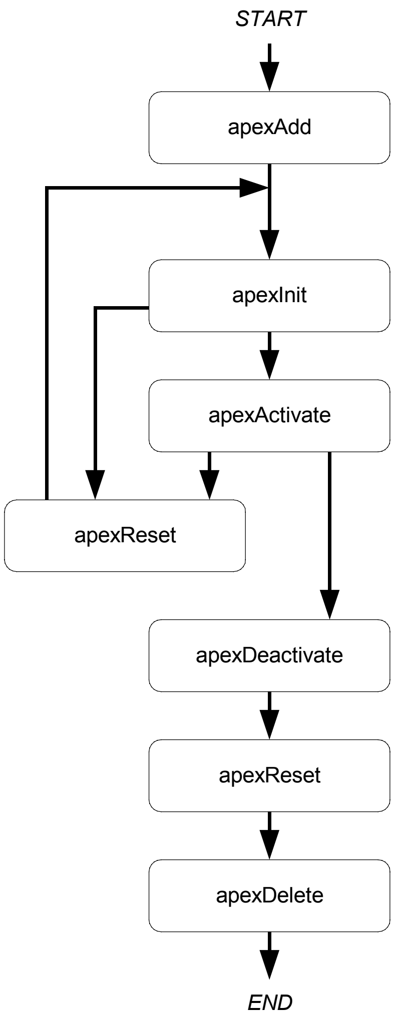
Figure 4: Driver Initialization and Shutdown



Device Addition, Initialization, and Deletion

Figure 5 illustrates the typical function call sequences that occur when adding, initializing, re-initializing and deleting devices.

Figure 5: Device Addition, Initialization, and Deletion



This function detects the device being added to the hardware (using sysApexDeviceDetect). Then it performs a register readback test, assigns a device handle for storing device information, and applies a software reset to the device.

This function initializes the device based on an initialization vector or initialization vector profile that you provide. Your application validates the initialization vector, then the driver stores it as part of device context information. The device registers are then configured accordingly.

This function installs and enables interrupts, enables the transmission and reception of cells and frames from the microprocessor port, and enables the queue engine's external interfaces. The device is now operational and all other APIs can be invoked.

This function de-activates the device and removes it from normal operation. It disables device interrupts, disables the transmission and reception of cells and frames from the microprocessor port, and disables the queue engine's external interfaces.

This function performs a software reset on the device. It also resets the device context information in the DDB contents, except for the initialization vector. This function can be invoked from any device state.

This function removes the device from the list of devices being controlled by the driver. This function also clears the device context information for the device being deleted and frees the device handle assigned for this device.

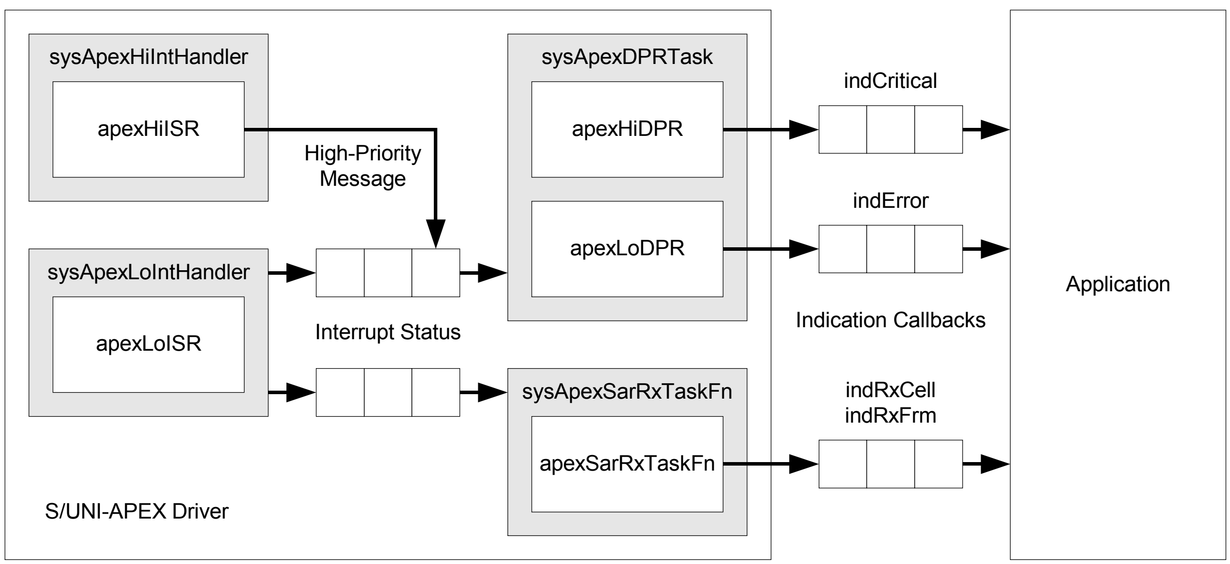
3 INTERRUPT SERVICING

The S/UNI-APEX driver services device interrupts by using an interrupt service routine (ISR) and a deferred processing routine (DPR). The ISR traps the interrupts and saves the interrupt context information. The DPR performs the actual processing of the saved interrupt context information. 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-APEX driver.

Figure 6 illustrates the interrupt service model used in the S/UNI-APEX driver design.

Figure 6: Interrupt Service Model



The interrupt service code includes some system-specific code that you provide (routines prefixed by `sysApex`); it also includes some application-independent code that comes with this driver and does not change from application to application (prefixed by `apex`).

You must implement the following system-specific interrupt-handler routines and install them in the interrupt vector table of the system processor: `sysApexHiIntHandler` and `sysApexLoIntHandler`. They correspond to the high and low priority interrupt pins of the S/UNI-APEX device. The microprocessor invokes these routines when one or more S/UNI-APEX devices interrupt the processor.

3.1 High-Priority Interrupt Servicing

When a high priority interrupt occurs, `sysApexHiIntHandler` invokes a driver provided routine, `apexHiISR`, for each device that has high-priority interrupt processing enabled. The `apexHiISR` function reads the high-priority interrupt status-register of the device and returns with the status information if a valid status bit is set. Then `sysApexHiIntHandler` sends this status information to the DPR task via a high-priority messaging function to the DPR task.

The DPR task processes this information using the driver provided routine, `apexHiDPR`. This function updates the interrupt counters for the interrupt events causing the interrupt. For each event that crosses its threshold, it invokes an indication callback, `indCritical`. The input arguments passed to this indication function include your context for the device, the event identifier, and any applicable event information.

After processing all interrupt events, the DPR exits after enabling the high-priority interrupt processing.

3.2 Low-Priority Interrupt Servicing

When a low priority interrupt occurs, `sysApexLoIntHandler` invokes a driver provided routine, `apexLoISR`, for each device that has low-priority interrupt processing enabled. The `apexLoISR` function reads the low-priority interrupt error-register and low-priority interrupt status-register. After that, it returns the status information if valid error or status conditions are detected. The driver then selectively sends the status information to one of two tasks, depending on the nature of the condition(s) detected:

- The SAR receive task
- The DPR task

A system-specific routine, `sysApexSarRxTaskFn,` runs as a separate task (SAR receive task) within the RTOS. This task waits for messages, sent by `sysApexLoIntHandler`, to arrive at an associated message queue. These messages correspond to arrival of cell(s) in the SAR TX Data register(s)

When `sysApexSarRxTaskFn` receives a message, it invokes the driver-provided routine, `apexSarRxTaskFn`. The `apexSarRxTaskFn` routine takes the appropriate actions based on the status information received in the message. Actions include extracting cells/frames from the SAR TX registers and reporting frame re-assembly timeouts or length errors to the application via indication callback functions.

Another system-specific routine, `sysApexDPRtask`, runs as a separate task (DPR task) within the RTOS. This task also waits for messages, sent by `sysApexLoIntHandler`, to arrive at an associated message queue. These messages correspond to interrupt conditions that are not SAR-related. These events include the following:

- Port, class, and VC maximum threshold errors
- VC cell receive error, re-assembly length error, and re-assembly timeout error
- WAN and loop transmit-cell transfer error
- WAN and loop receive runt-cell error, parity error

When the driver receives a message, it invokes the driver-supplied function, `apexLoDPR`. This function updates the interrupt counters for the interrupt events that cause the interrupt. For each event that crosses its threshold, it invokes an indication callback, `indError`. The input arguments passed to this indication function include your context for the device, the event identifier, and any applicable event information.

After processing all interrupt events, the DPR exits after enabling the low-priority error interrupt processing.

Note: The driver-provided routines, `apexHiISR`, `apexLoISR`, `apexSarRxTaskFn`, `apexHiDPR`, and `apexLoDPR` do not specify a communication mechanism between the ISRs and tasks. Therefore, you have full flexibility in choosing a communication mechanism between the two. The most common way to implement this communication mechanism is to use a message queue, a service that most RTOSs provide.

3.3 Installation and Removal of Interrupt Handlers

You must implement the system-specific routines, `sysApexHiIntHandler`, `sysApexLoIntHandler` and `sysApexDPRtask`. Your interrupt installation routine, `sysApexIntInstallHandler`, installs the interrupt handlers (`sysApexHiIntHandler` and `sysApexLoIntHandler`) in the interrupt vector table of the processor.

The `sysApexDPRtask` is spawned as a task during the first time invocation of `sysApexIntInstallHandler`. In addition, `sysApexIntInstallHandler` also creates the communication channels between `sysApexLoIntHandler` and `sysApexDPRtask`. Programmers usually implement this communication channel as a message queue.

Similarly, during removal of interrupts, the driver removes `sysApexHiIntHandler` and `sysApexLoIntHandler` from the microprocessor's interrupt vector table and then deletes the `sysApexDPRtask` task. You must implement the function, `sysApexIntRemoveHandler`, that removes the interrupt handlers and the DPR task.

4 QUEUE ENGINE

The driver's queue engine controls and maintains the external and internal queue context information of the S/UNI-APEX devices. It also keeps track of the configured ports, classes, shapers, and connections and identifies how these entities are associated with each other.

4.1 Queue-Engine Data Structures

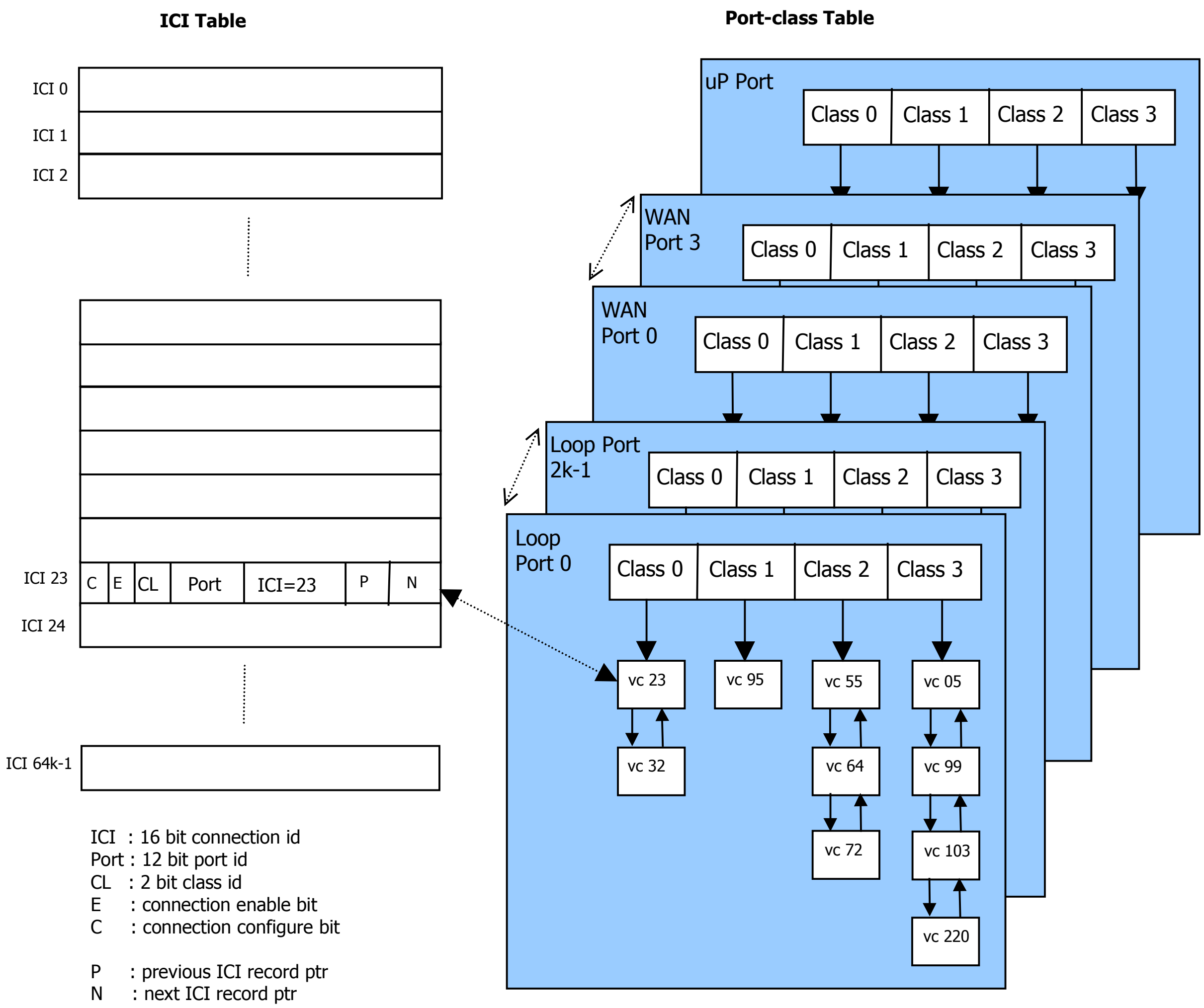
The queue engine module uses two main data structures, the port-class table and the ICI table. The port-class table is a data structure used to efficiently look up all the VCs that are associated with a port-class combination. The ICI array is used to efficiently lookup the port-class combination that a VC is associated with. Note that these two constructs are actually two different views of the same block of memory.

4.2 Port-Class and ICI Tables

Figure 7 is a diagrammatic representation of the queue control table and the ICI array. The ICI array is an array of 16k/64k elements. Each element has the following structure:

```
typedef struct _apx_ici_rec
{
        UINT4   conn;
        struct _apx_ici_rec *prev;
        struct _apx_ici_rec *next;
} sAPX_ICI_REC;
```

Figure 7: Port-Class Table Layout



The contents of the member, 'conn' , that belongs to this structure are used to look up the port-class combination that a connection is associated with. The 'prev' and 'next' pointers are used to form ordered linked lists of connections that are associated with a particular port-class combination. Each record of the table has the following structure:

```
typedef struct _apx_prt_class_rec
{
        UINT4           status;
        UINT2           numICIs[APX_NUM_CLASSES];
        sAPX_ICI_REC    *psIciLstHead[APX_NUM_CLASSES];
        sAPX_ICI_REC    *psIciLstTail[APX_NUM_CLASSES];
} sAPX_PRT_CLASS_REC;
```

The port-class table is comprised of the following arrays of port-class records:

```
typedef struct _apx_prt_class_tbl
{
        sAPX_PRT_CLASS_REC    *prec[APX_NUM_PORT_TYPES];
        sAPX_PRT_CLASS_REC    lp[APX_NUM_LOOP_PORTS];
        sAPX_PRT_CLASS_REC    wp[APX_NUM_WAN_PORTS];
        sAPX_PRT_CLASS_REC    up;
} sAPX_PRT_CLASS_TBL;
```

The table consists of ordered linked lists of connections associated with each port-class combination for loop, WAN, and microprocessor ports. The queue engine module uses this information to tear down a port or class gracefully. For example, to shutdown loop port 0, the queue engine module checks the table to figure out which connections and classes are associated with that port. In this case, connections 23, 32, 95, 55, 64, 72, 05, 99, 103 and 220 are torn down; then classes 0 through 3; and finally, port 0 is shutdown.

4.3 Queue Control Block

The queue control block contains all the bookkeeping information required by the queue engine module.

```
typedef struct _apx_qe_cb
{
        sAPX_ICI_REC          sIciTbl[APX_MAX_NUM_VCS];
        sAPX_PRT_CLASS_TBL    sPrtClTbl;
        UINT2                 u2WdgStartIci;
        UINT2                 u2WdgEndIci;
        UINT2                 u2PrtCfgCnt[APX_NUM_PORT_TYPES];
        UINT2                 u2ClCfgCnt[APX_NUM_PORT_TYPES];
        UINT4                 u4ConnCfgCnt[APX_NUM_PORT_TYPES];
} sAPX_QE_CB;
```

The sAPX_QE_CB structure contains the following information:

- The ICI and port-class tables previously described
- The watchdog patrol start and end ICI parameters (specified in the initialization vector)
- Counts for the number of configured ports, classes, and connections for each port type: uP, loop, and WAN

5 LOOP PORT SCHEDULER

The loop port scheduler (LPS) schedules packet transmissions to the 2048 loop ports that the device can handle. In order for all these ports to be serviced fairly and efficiently, the loop port scheduler uses the port weight and port sequence tables that reside in the LPS internal context memory.

When loop ports are configured, they have to be assigned weights and sequence numbers to achieve the following:

- High-bandwidth ports are serviced more often than ports with low-bandwidth requirements. The LPS module achieves this goal by assigning lower weights to high-bandwidth ports and higher weights to low-bandwidth ports. You must assign the weight values.
- The number of ports that need to be polled (to see if they can accept a packet for transmission) at any time is minimal. The port polling times should be "spread-out" and not "bunched-up". The APEX driver's LPS module achieves this by assigning sequence number to ports (that have the same weight), such that the number of ports associated with each sequence number is evenly distributed across the sequence numbers used for each weight.

5.1 Assigning Sequence Numbers

The sequence numbers assigned depend on the weight assigned to the node. For ports with weight 1, the sequence number assigned is either 0 or 1. For ports with weight 2, the sequence number is one of 0, 1, 2 or 3. Thus the number of possible sequence numbers increases with the weight assigned to a port. Subsequently, for ports assigned a weight of 7, the maximum possible range of sequence numbers is utilized, namely; 0 through 127.

Table 3: Port Poll-Sequence Numbers

Number of Port Added	Sequence Numbers Assigned for Each Weight							
	Wt 0	**Wt 1**	**Wt 2**	**Wt 3**	**Wt 4**	**Wt 5**	**Wt 6**	**Wt 7**
1	0	0	0	0	0	0	0	0
2	0	1	2	4	8	16	32	64
3	0	0	1	2	4	8	16	32
4	0	1	3	6	12	24	48	96
5	0	0	0	1	2	4	8	16
6	0	1	2	5	10	20	40	80
7	0	0	1	3	6	12	24	48

Number of Port Added	Sequence Numbers Assigned for Each Weight							
	Wt 0	Wt 1	Wt 2	Wt 3	Wt 4	Wt 5	Wt 6	Wt 7
8	0	1	3	7	14	28	56	112
-	-	-	-	-	-	-	-	-
2041	0	0	0	0	1	3	7	15
2042	0	1	2	4	9	19	39	79
2043	0	0	1	2	5	11	23	47
2044	0	1	3	6	13	27	55	111
2045	0	0	0	1	3	7	15	31
2046	0	1	2	5	11	23	47	95
2047	0	0	1	3	7	15	31	63
2048	0	1	3	7	15	31	63	127

Since sequence numbers should be assigned in a manner that the port polling times are “spread-out” and not “bunched-up,” the sequence number are not assigned in a linear order. Referring to Table 3, consider the sequence numbers for weight 3. The possible sequence numbers are 0, 1, 2, 3, 4, 5, 6 and 7. The first port of weight 3 is given the sequence number 0. The second port of weight 3 is assigned the sequence number 4(and not 1). This would “spread-out” the time interval between scheduling of these two ports with the same weight. The third port of weight 3 would be assigned the sequence number 2 and the fourth port is given the sequence number 6 and so on.

Note: Since the number of ports assigned to a particular weight can be greater than the number of sequence numbers available for that weight, the sequence numbers are repeated. For example, in the table the sequence numbers for weight 2 are repeated every 4 ports, whereas the sequence numbers for weight 3 are repeated every 8 ports and so on.

5.2 LPS Data Structures

The LPS module uses two main data structures: the poll sequence database and the port sequence table. The poll sequence database is a data structure used to efficiently assign sequence numbers to ports of different weights, such that the sequence numbers are distributed. The port sequence table is used to efficiently lookup the sequence number already assigned to a particular port.

5.3 Poll Sequence Database

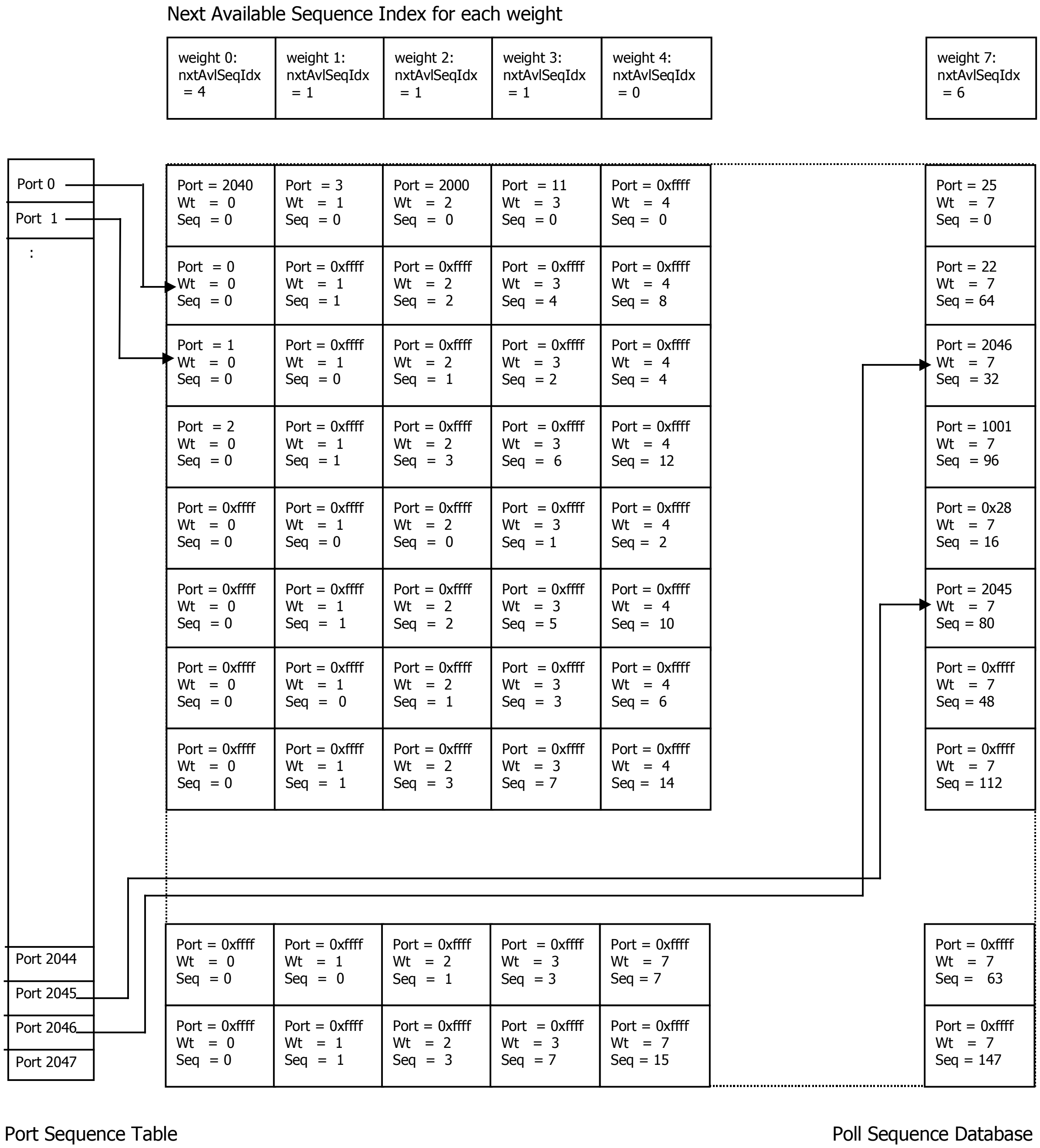
Figure 8 shows that the Poll Sequence Database is a two dimensional array. It consists of 8 columns, each column corresponding to a particular weight (0 – 7). The number of rows corresponds to the maximum number of loop ports. Each element of the array is a poll sequence record, which has the following structure:

```
typedef struct _apx_poll_seq_rec
{
        UINT1  u1PortWt;
        UINT1  u1PortSeq;
        UINT2  u2PortNum;
} sAPX_POLL_SEQ_REC;
```

When the poll sequence database is initialized, the port number for all the poll sequence records is set to 0xFFFF. This means that the sequence number associated with the node is unassigned. All the poll sequence records in the same column are assigned the same weight, which is the same as the column index of the array.

The sequence numbers in each column of the poll sequence database are initialized by following the same procedure used to assign sequence numbers for each column in Table 3.

Figure 8: LPS Module Data Structures



5.4 Port Sequence Table

The port sequence table is an array of pointers to poll sequence records. The purpose of this table is to efficiently lookup the sequence number assigned to a particular port. On initialization each entry of the port sequence table will be set to NULL since none of the ports have yet been assigned a sequence number. Each time a sequence number is assigned to a port, an entry in the port sequence table, indexed by the port number, is updated.

In Figure 8, the entry in the port sequence table for port 2046, points to the poll sequence record in column 7 and row 2; this has a sequence number of 32 associated with it.

5.5 Assigning Port Sequence Numbers

When a loop port is added, depending on the weight of the port, the driver routine goes to a particular column of the poll sequence database and, starting from the first row, it searches for the first unassigned poll sequence record. The sequence number in this record is the one assigned to the port. To expedite this process, a 'next available sequence index' array, of a dimension equal to the number of weights, is created. This array has an entry for each column of the poll sequence database and contains the index of the next unassigned poll sequence record in the corresponding column. At initialization all the index values will be zero. The next available sequence index for a column is updated each time a port is added or deleted from that column.

Consider an example where we have to assign a sequence number to loop port 10, which has a weight of 7. In the figure, the entry for weight 7 in the next available sequence index array is 6. Looking at the poll sequence database, the record at column 7 and row 6 is unassigned (since the port number entry is 0xFFFF). So the sequence number 48, which is associated with this record is assigned to loop port 10. To indicate that the sequence number has been assigned the port number for the poll sequence record is set to 10.

The entry for port 10 in the port sequence table is set to point to the poll sequence record at column 7 and row 6. The next available sequence number index for weight 7 is updated to 7.

5.6 Updating Port Sequence Numbers

On deleting a port, the driver routine gets the address of the poll sequence record from the port sequence table using the port number as the index. The poll sequence record is then freed. In this scenario, there is a free sequence number in the middle of a series of sequence numbers assigned to ports of the same weight. Note that deleting a few ports could potentially lead to a situation where the port polling times for ports of the same weight are “bunched up.” To avoid this situation, when a port of a particular weight is deleted, we reassign this sequence number to another port; specifically, a port that meets the following criteria: the port has the same weight, and the port is the last one in the series of sequence numbers assigned for this weight. By doing this, instead of having a free sequence number in the middle of a series of assigned sequence numbers for the same weight, we free the sequence number that is at the end of the series. Doing this guarantees that the sequence numbers remain “distributed.”

Referring to the figure, consider an example where we have to delete port number 2046. Using the port sequence table, we get a pointer to the poll sequence record at column 7 and row 2. Deleting the port frees up the sequence number 32. This sequence number needs to be reassigned to another port. The port that meets the criteria for reassignment is port 2045, since it has a weight of 7 and is the last one in the series of assigned sequence number for weight 7. So port 2045 is assigned sequence number of 32. The sequence number 80, previously assigned to port 2045 is freed. The next available poll sequence index entry for weight 7 is changed to 5. The entry for port 2045, in the port sequence table, now points to poll sequence node at column 7 and row 2.

6 SAR ASSIST

The SAR component provides the following functions:

- Insertion and extraction of cells
- Insertion and extraction of AAL5 frames
- Multicast forwarding of cells on multiple VCs

Figure 9: SAR Assist

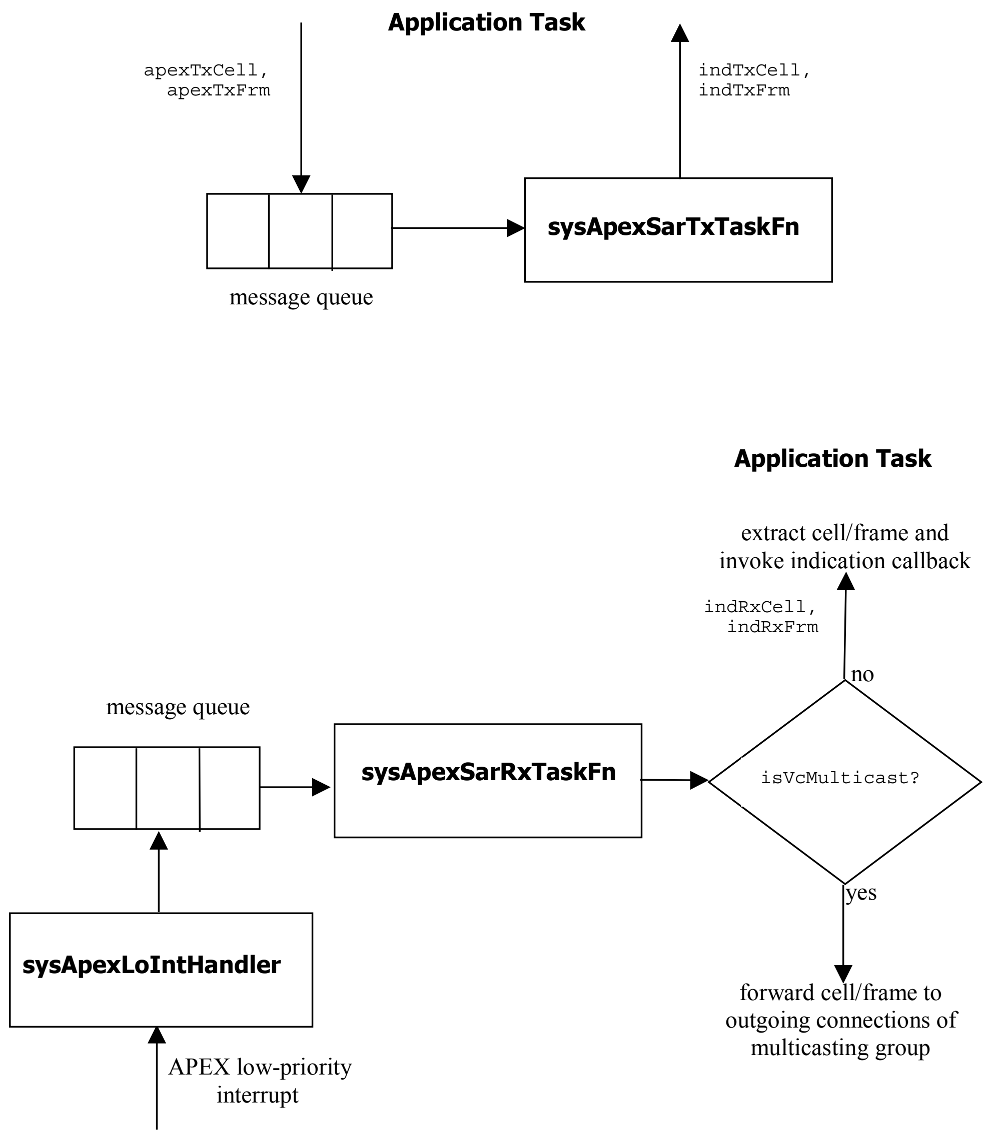


Figure 9 illustrates the SAR-Assist component's architecture. The SAR Assist component is implemented as a set of two tasks. One task is responsible for transmitting cells and frames, the other is responsible for receiving cells and frames. Both tasks are spawned when the first device is activated by invoking `apexActivate`. In addition to creating the tasks, `apexActivate` will also create a message queue for each task; this queue is used to communicate with the task.

Insertion of cells and frames

When the user invokes `apexTxCell` or `apexTxFrm` to transmit a cell or frame, the information is encapsulated into a message structure and is sent to the message queue of the SAR transmit task. The transmit task then dequeues the message and calls the appropriate routines to transmit the cell or frame. Once the transmission is complete, it invokes the indication callback functions `indTxCell` or `indTxFrm`, to inform the user about the status.

Extraction of cells and frames

When a low priority interrupt occurs and the low priority interrupt handler determines the cause of the interrupt to be the arrival of cells at the SAR module, it sends a message to the SAR receive task.

Once a message is received by the SAR receive task, it invokes the driver-provided routine, `apexSarRxTaskFn`. The `apexSarRxTaskFn` routine will scan through the four class queues, in order of priority as specified by the user in the initialization vector. If the multicasting support is not activated, the SAR receive task retrieves the cells/frames and saves them in a buffer. It then invokes the indication callback functions `indRxCell` or `indRxFrm` to inform the application about the receipt of the cell/frame and provides a pointer to the header and payload.

Multicast forwarding

The SAR module also provides support for multicasting cells and frames. Multicasting is defined as: forwarding a cell or frame received on an incoming connection to multiple outgoing connections. The incoming connection and the outgoing connections together comprise a multicast group. The multicast forwarding feature is enabled by installing the multicasting callback function, `isVcMulticast`, using the routine `apexInstallMulticastFn`. The multicasting support is disabled by invoking `apexResetMulticastFn`.

When the multicast forwarding feature is enabled, each time a cell or frame is received by the SAR receive task, the multicasting callback function is invoked with the connection ID of the received cell as an input. The multicasting callback function, which is provided by the user, determines whether the connection ID belongs to a multicasting group.

If the connection does belong to a multicast group, the callback function provides information about the number of outgoing connections and the connection id for each outgoing connection. The SAR receive task then forwards the cell or frame to these outgoing connection. It should be noted that in the event that the cell/frame is multicast, the cell/frame is not saved in a buffer and the indication callback functions `indRxCell` or `indRxFrm` are not invoked. The contents of the cell and frame are not checked for errors in payload, such as CRC errors, errors in frame length etc.

On the other hand, if the connection does not belong to a multicast group, the callback function returns the number of outgoing connections as 0. The SAR receive task then retrieves the cell or frame, saves it to a buffer, and invokes the callback functions `indRxCell` or `indRxFrm` (as in the case where multicasting support is not enabled).

7 DATA STRUCTURES

The following are the main data structures used by the S/UNI-APEX driver.

7.1 Global Driver Database

Table 4: Global Driver Database: sAPX_GDD

Member Name	Type	Description
u4Mode	UINT4	1: Interrupt mode 2: Polling mode
u4MemSz	UINT4	Total memory allocated by driver
u4ImgRd	UINT4	1: Read from driver's context memory image 0: Read from actual physical context memory
semApex	APX_SEM_ID	Semaphore to protect critical sections of driver
u2NumDevs	UINT2	Number of devices currently registered.
u2NumDevsActive	UINT2	Number of devices in active state
sMiv	sAPX_MIV	Module initialization vector
psDdb	sAPX_DDB *	Array of (u2MaxDevs) device data block (DDBs) pointers of the registered devices
psInitProfs	sAPX_INIT_VECT *	An array of pointers to different initialization vector profiles. A profile simply serves as a "canned configuration" that can be used to initialize a device without having to pass all the initialization parameters every time a device is configured. Instead, the application passes a profile number. The driver then indexes this array, obtains the initialization vector, and configures the device accordingly.

Member Name	Type	Description
psPortProfs	sAPX_PORT_VECT *	An array of pointers to port-parameter vector profiles. You can use these profile parameters to configure loop, WAN and uP ports easily without having to pass all the port parameters each time you add a port. This is useful when several ports have the same parameters.
psClassProfs	sAPX_CLASS_VECT *	An array of pointers to class vector profiles
psConnProfs	sAPX_CONN_VECT *	An array of pointers to connection-parameter vector profiles

7.2 Device Data Blocks

Each device data block (DDB) stores control information for a single S/UNI-APEX device. The driver allocates a DDB when the driver registers a new device. The driver de-allocates it when the driver deregisters the device.

Table 5: Device Data Block: sAPX_DDB

Member Name	Type	Description
u4Valid	UINT4	Indicates that this is a valid DDB if its value is APX_VALID
pSysInfo	void *	Pointer to system-specific device information. For example, in PCI bus environments, the bus, device, function numbers, IRQ assignment.
eDevState	eAPX_DEV_STATE	Device state, which can be one of the following: • APX_PRESENT • APX_INIT • APX_ACTIVE
u4BaseAddr	UINT4	Base address of the device

Member Name	Type	Description
usrCtxt	APX_USR_CTXT	Pointer to device context information, which the application maintains. Your application must pass this pointer while adding the device. The driver passes this information when it invokes the indication callbacks.
u4CbDiagMd	UINT4	Cell-buffer diagnostic access-mode: • CB_DIAG_DISABLED • CB_DIAG_READ • CB_DIAG_WRITE
u4MaxVCs	UINT4	Maximum number of VCs to be used by device
u4MaxCellBufs	UINT4	Maximum number of cell buffers (for queuing) to be used by device
u1LpTxECIPreEn	UINT1	Indicates if ECI prepend is expected on the loop transmit interface
u1LpTxHecDis	UINT1	Indicates if HEC/UDF field is expected on the loop transmit interface
u1WanTxECIPreEn	UINT1	Indicates if ECI prepend is expected on the WAN transmit interface
u1WanTxHecDis	UINT1	Indicates if HEC/UDF field is expected on the WAN transmit interface
u2LpTxSwPreEn	UINT2	Indicates if a switch tag prepend is expected on the loop transmit interface
u2WanTxSwPreEn	UINT2	Indicates if a switch tag prepend is expected on the WAN transmit interface
u4QLClsStartAddr	UINT4	Offset for the start of the loop-class context records in the external-queue context memory
u4ShprStartAddr	UINT4	Offset for the start of the shaper TxSlot context records in the external-queue context memory
u4CellStartAddr	UINT4	Offset for the start of the cell context records in the external-queue context memory
sInitVect	sAPX_INIT_VECT	Device configuration information that the application passes to the driver. The driver writes to the appropriate device registers, based on the contents of this vector.
sCtxt	sAPX_CTXT_IMG	Driver’s image of the context memory

Member Name	Type	Description
sIsmCb	sAPX_ISM_CB	Interrupt service control block
sQeCb	sAPX_QE_CB	Queue-engine control block
sSarCb	sAPX_SAR_CB	SAR-assist control block
sLpsCb	sAPX_LPS_CB	LPS control block

7.3 Configuration Vectors

Module Initialization Vector Structure: sAPX_MIV

The application allocates the module initialization vector before initializing an S/UNI-APEX device. The module initialization vector defines the number of profiles used by the driver.

Table 6: Module Initialization Vector Structure: sAPX_MIV

Member Name	Type	Description
u2MaxInitProfs	UINT2	Maximum number of initialization profiles supported by the driver.
u2MaxPortProfs	UINT2	Maximum number of port profiles supported by the driver.
u4MaxClassProfs	UINT4	Maximum number of class profiles supported by the driver.
u4MaxConnProfs	UINT4	Maximum number of connection profiles supported by the driver.

Device Initialization Vector Structure: sAPX_INIT_VECT

The application allocates the initialization vector before initializing an S/UNI-APEX device. The initialization vector contains various configuration parameters that the driver uses to program the device's control registers. It is the responsibility of the application to free the initialization vector memory.

Table 7: Device Initialization Vector Structure: sAPX_INIT_VECT

Member Name	Type	Description
u4Valid	UINT4	Indicates whether or not this vector's contents are valid: • APX_VALID • APX_INVALID Note: You should not set this field.
u1SarRxPri[4]	UINT1	Service priority for each of the four classes of the uP port
u4MaxVCs	UINT4	Maximum number of VCs
u4MaxCellBufs	UINT4	Maximum number of cell buffers available for queuing
sRegs	sAPX_REGS	Contains the values to be written to the control registers of the device
indCritical	APX_IND_INTR	Indication callback routine, invoked by the DPR, to notify the application of a high-priority interrupt event
indError	APX_IND_INTR	Indication callback routine, invoked by the DPR, to notify the application of a low-priority interrupt event
indTxCell	APX_IND_TX_CELL	Indication callback routine, invoked by the SAR transmit task, to confirm the success or failure of a cell transmission request by the application
indTxFrm	APX_IND_TX_FRM	Indication callback routine, invoked by the SAR transmit task, to confirm the success or failure of an AAL5 frame-transmission request by the application
indRxCell	APX_IND_RX_CELL	Indication callback routine, invoked by the SAR receive task, to notify the application of the reception of a cell
indRxFrm	APX_IND_RX_FRM	Indication callback routine, invoked by the SAR receive task, to notify the application of the reception of an AAL5 frame

Port Vector Structure: sAPX_PORT_VECT

The driver uses the port parameters vector to store port configuration profiles. They also pass port configuration parameters to the driver.

Table 8: Port Vector Structure: sAPX_PORT_VECT

Member Name	Type	Description
u1Valid	UINT1	Indicates whether or not this vector's contents are valid: • APX_VALID • APX_INVALID Note: You should not set this field.
u1Clp0Thrsh	UINT1	Maximum threshold for CLP0 cells
u1Clp1Thrsh	UINT1	Maximum threshold for CLP1 cells
u1MaxThrsh	UINT1	Maximum threshold for all cells
u4PollWt	UINT4	LPS (or WPS) poll weight
u4PollSeq	UINT4	LPS poll sequence
sCschd	sAPX_CS_VECT	Class scheduler parameters

Class Vector Structure: sAPX_CLASS_VECT

The driver uses the class parameters vector to store class configuration profiles. It also passes class configuration parameters to the driver.

Table 9: Class Vector Structure: sAPX_CLASS_VECT

Member Name	Type	Description
u4Valid	UINT4	Indicates whether or not this vector's contents are valid: • APX_VALID • APX_INVALID Note: You should not set this field.
u1ShpFlg	UINT1	1: This class is shaped 0: This class in not shaped
u1Clp0Thrsh	UINT1	Maximum threshold for CLP0 cells
u1Clp1Thrsh	UINT1	Maximum threshold for CLP1 cells

Member Name	Type	Description
u1MaxThrsh	UINT1	Maximum threshold for all cells

Connection Vector Structure: sAPX_CONN_VECT

The driver uses the connection parameters vector to store connection configuration profiles. It also passes connection configuration parameters to the driver.

Table 10: Connection Vector Structure: sAPX_CONN_VECT

Member Name	Type	Description
u4Valid	UINT4	Indicates whether or not this vector's contents are valid: • APX_VALID • APX_INVALID Note: You should not set this field.
u1EndSegOam	UINT1	00b: No redirection of OAM cells to uP 01b: Redirection of segment OAM cells to uP 10b: Redirection of end-end OAM cells to uP 11b: Redirection of both segment and end-end OAM cells to uP
u1VcVpc	UINT1	VC or VPC
u1Clp0MinThrsh	UINT1	Minimum number of CLP0 cells guaranteed to be allowed on a per-VC basis
u1Clp0Thrsh	UINT1	Maximum threshold for CLP0 cells
u1Clp1Thrsh	UINT1	Maximum threshold for CLP1 cells
u1MaxThrsh	UINT1	Maximum threshold for all cells
u1EfciMd	UINT1	EFCI marking mode
u1GfrMd	UINT1	GFR mode
u4RemapMd	UINT4	VC remapping mode (0-3)

Member Name	Type	Description
eQtype	eAPX_Q_TYPE	Queuing type, which can be one of the following: • WFQ • FCQ • SFQ
sRemap	sAPX_VC_REMAP	VC address remap information
uQinfo	uAPX_VC_Q_INFO	VC queuing parameters

Shaper Vector Structure: sAPX_SHPR_VECT

The shaper-parameters vector stores shaper configuration profiles and passes shaper configuration parameters to the driver.

Table 11: Shaper Vector Structure: sAPX_SHPR_VECT

Member Name	Type	Description
u1Valid	UINT1	Indicates whether or not this vector's contents are valid: • APX_VALID • APX_INVALID Note: You should not set this field.
u1Port	UINT1	WAN port to be shaped (0 to 3)
u1Class	UINT1	WAN port-class to be shaped
u1SlowDnEn	UINT1	Slow down enable used to provide fair shaping to high-speed VCs
u1ThrshEn	UINT1	Enables comparison of class queue-length and shaper threshold-value
u1ThrshVal	UINT1	Shaper threshold value (ignored if u1ThrshEn = 0)
u1MeasInt	UINT1	Congestion level measurement-interval (4-bit logarithmic value)
u1RedFact	UINT1	Encoded slow-down rate-reduction factor (0-3)
u4RtRate	UINT4	Real-time rate for shaper (9-bits)

7.4 Other API Data Structures

Port ID Structure: sAPX_PORT_ID

The port ID structure identifies the port type (loop, WAN, or uP) and port number.

Table 12: Port ID Structure: sAPX_PORT_ID

Member Name	Type	Description
u2Type	UINT2	Port type: • APX_LOOP_PORT • APX_WAN_PORT • APX_UP_PORT
u2Num	UINT2	Port number: • Loop (0 to 2047) • WAN (0 to 3) • uP (0)

Class ID Structure: sAPX_CLASS_ID

The class ID structure identifies a port-class by port type (loop, WAN, or uP), port number, and class number.

Table 13: Class ID Structure: sAPX_CLASS_ID

Member Name	Type	Description
u1Type	UINT1	Port type: • APX_LOOP_PORT • APX_WAN_PORT • APX_UP_PORT
u1Class	UINT1	Class number (0 to 3)

Member Name	Type	Description
u2Port	UINT2	Port number: • Loop (0 to 2047) • WAN (0 to 3) • uP (0)

Connection ID Structure: sAPX_CONN_ID

The connection ID structure identifies a connection and its destination port-class.

Table 14: Connection ID Structure: sAPX_CONN_ID

Member Name	Type	Description
u1Type	UINT1	Destination port type: • APX_LOOP_PORT • APX_WAN_PORT • APX_UP_PORT
u1Class	UINT1	Destination class number (0 to 3)
u2Port	UINT2	Destination port number: • Loop (0 to 2047) • WAN (0 to 3) • uP (0)
u4ICI	UINT4	ICI of the connection

Port Weight Structure: sAPX_PORT_WT

The port weight structure specifies the weight for a particular port.

Table 15: Port Weight Structure: sAPX_PORT_WT

Member Name	Type	Description
u2PortNum	UINT2	Port number
u1PortWt	UINT1	Port weight

Port Sequence Structure: sAPX_PORT_SEQ

The port sequence structure specifies the sequence number for a particular port.

Table 16: Port Sequence Structure: sAPX_SEQ_WT

Member Name	Type	Description
u2PortNum	UINT2	Port number
u1PortSeq	UINT1	Port sequence number

Queue-Module Information Structure: sAPX_QE_INFO

The queue-module information structure retrieves information from the queue module's control block.

Table 17: Queue-Module Information Structure: sAPX_QE_INFO

Member Name	Type	Description
u2WdgStartIci	UINT2	Start of ICI watchdog patrol range
u2WdgEndIci	UINT2	End of ICI watchdog patrol range
u2PrtCfgCnt[]	UINT2	Number of ports configured in loop, WAN, and uP directions
u2ClCfgCnt[]	UINT2	Number of classes configured in loop, WAN, and uP directions
u4ConnCfgCnt[]	UINT4	Number of connections configured in loop, WAN and uP directions

Module Information Structure: sAPX_MODULE_INFO

The module information structure retrieves select GDD parameters.

Table 18: Module Information Structure: sAPX_MODULE_INFO

Member Name	Type	Description
u2NumDevs	UINT2	Number of devices maintained by the driver (added)

Member Name	Type	Description
u2NumDevsActive	UINT2	Number of devices currently in APX_ACTIVE state
u4Mode	UINT4	1: Interrupt mode 2: Poll mode
u4MemSz	UINT4	Total memory allocated by the driver
u4ImgRd	UINT4	1: Perform context reads from driver image 0: Perform context reads from physical context memory

Device Information Structure: sAPX_DEV_INFO

The device information structure retrieves select DDB parameters.

Table 19: Device Information Structure: sAPX_DEV_INFO

Member Name	Type	Description
u4BaseAddr	UINT4	Base address of device
u4DevState	UINT4	Device state
usrCtxt	UINT4	Pointer to device context information, which the application maintains
u4CbDiagMd	UINT4	Cell-buffer diagnostic mode
u4LpClStartAddr	UINT4	Offset for the start of the loop-class context records in the external-queue context memory
u4ShprStartAddr	UINT4	Offset for the start of the shaper TxSlot context records in the external-queue context memory
u4CellStartAddr	UINT4	Offset for the start of the cell context records in the external-queue context memory
u4MaxVCs	UINT4	Maximum number of VCs
u4MaxCellBufs	UINT4	Maximum number of cell buffers available for queuing
u1LpTxECIPreEn	UINT1	Indicates if ECI prepend is expected on the loop transmit interface
u1LpTxHecDis	UINT1	Indicates if HEC/UDF field is expected on the loop transmit interface
u1WanTxECIPreEn	UINT1	Indicates if ECI prepend is expected on the WAN transmit interface

Member Name	Type	Description
u1WanTxHecDis	UINT1	Indicates if HEC/UDF field is expected on the WAN transmit interface
u2LpTxSwPreEn	UINT2	Indicates if a switch tag prepend is expected on the loop transmit interface
u2WanTxSwPreEn	UINT2	Indicates if a switch tag prepend is expected on the WAN transmit interface

SAR Transmit Context Structure: sAPX_TX_CTXT

The transmit context structure stores information about a transmit cell/frame for the SAR transmit task.

Table 20: SAR Transmit Context Structure: sAPX_TX_CTXT

Member Name	Type	Description
Apex	APEX	Apex device handle
txType	eAPX_SAR_TX_TYPE	Transmit type (either cell or frame)
txInfo	union (sAPX_CELL_INFO or sAPX_FRM_INFO)	Stores either cell information or frame information

8 APPLICATION PROGRAMMING INTERFACE

This section provides a detailed description of each function that is a member of the S/UNI-APEX driver API.

The API functions typically execute in the context of an application task.

Note: These functions are typically not re-entrant. Therefore, you should be careful not to execute the same functions in multiple tasks running concurrently. The driver does protect its data structures from simultaneous access by a single application task and all its internal tasks (i.e., the DPR and SAR tasks).

8.1 Driver Initialization and Shutdown Functions

This section describes the functions that initialize and shutdown the driver.

Initializing the Driver: apexModuleInit

This function initializes the device driver. Initialization involves allocating memory for the driver data structures (such as the GDD and DDB) and initializing these data structures.

Prototype `INT4 apexModuleInit(sAPX_MIV *psMiv)`

Inputs `psMiv`: Module initialization vector. The driver copies this vector into the GDD.

Outputs None

Returns `APX_SUCCESS`

`APX_ERR_MODULE_ALREADY_INIT`

`APX_ERR_MEM_ALLOC`

`APX_ERR_SEMAPHORE`

Shutting Down the Driver: apexModuleShutdown

This function shuts down the driver. Shutdown involves deleting all devices that the driver controls and de-allocating the GDD.

Prototype `void apexModuleShutdown(void)`

Inputs None

Outputs None

Returns None

8.2 Profile Management Functions

This section describes the functions that add, copy, and clear the following profiles:

- Initialization Profiles
- Port Profiles
- Class Profiles
- Connection Profiles

8.3 Initialization Profile Functions

This section describes the functions that add, copy, and clear initialization profiles.

Setting Initialization Profile Vectors: apexSetInitProfile

This function validates an initialization vector passed by the application and copies it into the GDD. Your application can now initialize a device by simply passing the initialization profile number. You should call this function only after `apexModuleInit`.

Prototype `INT4 apexSetInitProfile(sAPX_INIT_VECT *psProfile, UINT4 *pu4ProfileNum)`

Inputs `psProfile`: Profile that your application is setting

Outputs `pu4ProfileNum`: Profile number assigned by the driver

Returns APX_SUCCESS

APX_ERR_MODULE_NOT_INIT

APX_ERR_INVALID_INIT_VECTOR

APX_ERR_PROFILES_FULL

APX_ERR_MEM_ALLOC

Getting Initialization Profiles: apexGetInitProfile

This function copies the contents of the specified initialization vector stored in the GDD into the init-vector variable, which you provide. You should call this function only after calling `apexModuleInit`.

Prototype `INT4 apexGetInitProfile(UINT4 u4ProfileNum, sAPX_INIT_VECT *psProfile)`

Inputs `u4ProfileNum`: Profile number to display

Outputs `psProfile`: The driver copies the profile contents to this area, which you provide

Returns `APX_SUCCESS`

`APX_ERR_MODULE_NOT_INIT`

`APX_ERR_INVALID_PROFILE_NUM`

Clearing Initialization Profiles: apexClrInitProfile

Given the profile number, this function clears an initialization vector profile,

Prototype `INT4 apexClrInitProfile(UINT4 u4ProfileNum)`

Inputs `u4ProfileNum`: Initialization vector profile-number

Outputs None

Returns `APX_SUCCESS`

`APX_ERR_MODULE_NOT_INIT`

`APX_ERR_INVALID_PROFILE_NUM`

8.4 Port Profile Functions

This section describes the functions that add, copy, and clear port profiles.

Setting Port Profile Vectors: apexSetPortProfile

This function validates a port parameters vector, which you provide, and copies it into the GDD. Your application can now initialize a port by simply passing the initialization profile number. You should call this function only after `apexModuleInit`.

Prototype `INT4 apexSetPortProfile(sAPX_PORT_VECT *psProfile, UINT4 *pu4ProfileNum)`

Inputs `psProfile`: Profile that your application is setting

Outputs `pu4ProfileNum`: Profile number assigned by the driver

Returns
APX_SUCCESS
APX_ERR_MODULE_NOT_INIT
APX_ERR_INVALID_PORT_VECTOR
APX_ERR_PROFILES_FULL
APX_ERR_MEM_ALLOC

Getting Port Profiles: apexGetPortProfile

This function copies the contents of the specified port-parameters vector to the variable you provide.

Prototype `INT4 apexGetPortProfile(UINT4 u4ProfileNum, sAPX_PORT_VECT *psProfile)`

Inputs `u4ProfileNum`: Profile number to display

Outputs `psProfile`: Profile contents are copied here

Returns
APX_SUCCESS
APX_ERR_MODULE_NOT_INIT
APX_ERR_INVALID_PROFILE_NUM

Clearing Port Profiles: apexClrPortProfile

Given the profile number, this function clears a port vector profile.

Prototype `INT4 apexClrPortProfile(UINT4 u4ProfileNum)`

Inputs `u4ProfileNum`: Port-vector profile number

Outputs None

Returns
`APX_SUCCESS`
`APX_ERR_MODULE_NOT_INIT`
`APX_ERR_INVALID_PROFILE_NUM`

8.5 Class Profile Functions

This section describes the functions that add, copy, and clear class profiles.

Setting Class Profile Vectors: apexSetClassProfile

This function validates a class-parameters vector and copies it into the GDD. Your application can now initialize a class by simply passing the initialization profile number. You should call this function only after `apexModuleInit`.

Prototype
```
INT4 apexSetClassProfile(sAPX_CLASS_VECT *psProfile, UINT4
*pu4ProfileNum)
```

Inputs `psProfile`: The profile that your application is adding

Outputs `pu4ProfileNum`: Profile number assigned by the driver

Returns
`APX_SUCCESS`
`APX_ERR_MODULE_NOT_INIT`
`APX_ERR_INVALID_CLASS_VECTOR`
`APX_ERR_PROFILES_FULL`
`APX_ERR_MEM_ALLOC`

Getting Class Profiles: apexGetClassProfile

This function copies the contents of the specified class-parameters vector from the GDD to the variable you provide.

Prototype
```
INT4 apexGetClassProfile(UINT4 u4ProfileNum, sAPX_CLASS_VECT
*psProfile)
```

Inputs `u4ProfileNum`: Profile number to display

Outputs `psProfile`: Profile contents are copied in here

Returns APX_SUCCESS

APX_ERR_MODULE_NOT_INIT

APX_ERR_INVALID_PROFILE_NUM

Clearing Class Profiles: apexClrClassProfile

Given the profile number, this function clears a class vector profile.

Prototype `INT4 apexClrClassProfile(UINT4 u4ProfileNum)`

Inputs `u4ProfileNum`: Class-vector profile number

Outputs None

Returns APX_SUCCESS

APX_ERR_MODULE_NOT_INIT

APX_ERR_INVALID_PROFILE_NUM

8.6 Connection Profile Functions

This section describes the functions that add, copy, and clear connection profiles.

Setting Connection Profile Vectors: apexSetConnProfile

This function validates a connection-parameters vector and copies it into the GDD. Your application can now initialize a connection by simply passing the initialization profile number. The driver should call this function only after calling `apexModuleInit`.

Prototype `INT4 apexSetConnProfile(sAPX_CONN_VECT *psProfile, UINT4 *pu4ProfileNum)`

Inputs `psProfile`: Profile that your application is setting

Outputs `pu4ProfileNum`: Profile number assigned by the driver

Returns
APX_SUCCESS
APX_ERR_MODULE_NOT_INIT
APX_ERR_INVALID_CONN_VECTOR
APX_ERR_PROFILES_FULL
APX_ERR_MEM_ALLOC

Getting Connection Profiles: apexGetConnProfile

This function copies the contents of the specified connection-parameters vector from the GDD to the variable you provide.

Prototype

```
INT4 apexGetConnProfile(UINT4 u4ProfileNum, sAPX_CONN_VECT
*psProfile)
```

Inputs `u4ProfileNum`: Profile number to display

Outputs `psProfile`: Profile contents are filled in here

Returns
APX_SUCCESS
APX_ERR_MODULE_NOT_INIT
APX_ERR_INVALID_PROFILE_NUM

Clearing Connection Profiles: apexClrConnProfile

Given the profile number, this function clears a connection vector profile.

Prototype

```
INT4 apexClrConnProfile(UINT4 u4ProfileNum)
```

Inputs `u4ProfileNum`: Connection vector profile number

Outputs None

Returns
APX_SUCCESS
APX_ERR_MODULE_NOT_INIT
APX_ERR_INVALID_PROFILE_NUM

8.7 Device Addition and Removal Functions

This section describes the functions needed to add and removal devices.

Adding Devices: apexAdd

This function detects the new device in the hardware, assigns the device a device data block (DDB). Then it stores context information, which you maintain, for the device being added. Finally, it returns a device handle back to the application. You should use the device handle to identify the device on which the driver will perform the operation. Your application should call this function only after it calls `apexModuleInit`.

Prototype `INT4 apexAdd(APX_USR_CTXT usrCtxt, APEX *pApex)`

Inputs `usrCtxt`: Pointer to device context information, which the application maintains

Outputs `pApex`: Pointer to the S/UNI-APEX device handle that contains context information maintained by the driver.

Returns
`APX_SUCCESS`
`APX_ERR_MODULE_NOT_INIT`
`APX_ERR_DEVS_FULL`
`APX_ERR_DEV_NOT_DETECTED`
`APX_ERR_DEV_ALREADY_ADDED`
`APX_ERR_INVALID_TYPE_ID`
`APX_ERR_DLL_PHASE_LOCK`

Side Effects The device state changes to `APX_PRESENT`. The driver applies a software reset to the device.

Deleting Devices: apexDelete

This function removes the specified device from the list of devices that the driver controls. Deleting a device involves clearing the DDB for that device.

Prototype `INT4 apexDelete(APEX apex)`

Inputs `apex`: Device handle

Outputs None

Returns APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

Valid States APX_PRESENT

Side Effects The device handle, `apex`, is no longer valid.

8.8 Device Register Access Functions

Reading From Device Registers: apexReadReg

This function can be used to read the various registers of the APEX device.

Prototype `INT4 apexReadReg(APEX apex, UINT4 u4RegOff, UINT4 *pu4Val)`

Inputs `apex`: Device handle

`u4RegOff`: Register's offset from the base address (for example, 0x10, 0x14 etc)

Outputs `pu4Val`: Contents of the register

Returns APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_REG

Side Effects Will clear clear-on-read registers (e.g., interrupt status registers)

Writing To Device Registers: apexWriteReg

This function is used to write to the various registers of the APEX device.

Prototype `INT4 apexWriteReg(APEX apex, UINT4 u4RegOff, UINT4 u4Val)`

Inputs `apex`: Device handle

`u4RegOff`: Register's offset from the base address (for example, 0x10, 0x14 etc)

`u4Val:` Data to be written to the register

Outputs None

Returns APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_REG

Side Effects Writes to device registers after device initialization will overwrite initialization data and may cause incorrect operation of the device. Use with caution!

8.9 Device Diagnostic Functions

This section describes the functions that perform the following device tests:

Testing Register Access: apexRegisterTest

This function tests the microprocessor's access to the device registers by writing values to the registers and reading them back.

Prototype `INT4 apexRegisterTest(APEX apex)`

Inputs `apex`: Device handle

Outputs None

Returns APX_SUCCESS

APX_ERR_INVALID_DEV

APX_FAILURE

Side Effects The device is reset and put in the `APX_PRESENT` state

Testing Access to External Queue Context-Memory: apexExtQCtxtTest

This function tests the microprocessor's access to the external queue context-memory aperture.

Prototype `INT4 apexExtQCtxtTest(APEX apex, UINT1 u1TestType, UINT4 u4QuadStart, UINT4 u4QuadNum, sAPX_DATA34 *psPattern)`

Inputs

`apex`: Device handle

`u1TestType` (ZBT SSRAM tests):

- 0x00: WrRd.. pattern test
- 0x01: WrWr..RdRd.. pattern test
- 0x02: Address aliasing test

`u1TestType` (late write SSRAM tests):

- 0x10: WrRd.. pattern test
- 0x11: WrWr..RdRd.. pattern test
- 0x12: Address aliasing test

`u4QuadStart`: Starting quad-word for test

`u4QuadNum`: Number of quad-words to test

`psPattern`: Test pattern (only applicable for pattern tests)

Outputs None

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_TEST_PARAM

APX_FAILURE

APX_ERR_POLL_TIMEOUT

Side Effects The device is reset and put in the `APX_PRESENT` state

Testing Access to Internal Queue Context-Memory: apexIntQCtxtTest

This function tests the microprocessor's access to the internal queue context-memory aperture.

Prototype `INT4 apexIntQCtxtTest(APEX apex, UINT1 u1TestType, UINT4 u4QuadStart, UINT4 u4QuadNum, sAPX_DATA34 *psPattern)`

Inputs

`apex`: Device handle

`u1TestType`:

- 0: WrRd.. pattern test
- 1: WrWr..RdRd.. pattern test
- 2: address aliasing test

`u4QuadStart`: Starting quad-word for test (0..1559)

`u4QuadNum`: Number of quad-words to test (1..1559)

`psPattern`: Test pattern (only applicable for pattern tests). If test includes quad-words in the range 512-1023, then test pattern words should not be more than 16-bits wide

Outputs

None

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_TEST_PARAM

APX_ERR_INVALID_ADDR

APX_ERR_POLL_TIMEOUT

APX_FAILURE

Side Effects

The device is reset and put in the `APX_PRESENT` state

Testing Access to LPS Context-Memory: apexLpsCtxtTest

This function tests the microprocessor's access to the LPS context-memory aperture.

Prototype

```
INT4 apexLpsCtxtTest(APEX apex, UINT1 u1Ctxt, UINT1
u1TestType, UINT4 u4Pattern)
```

Inputs `apex`: Device handle

`u1Ctxt`:

- 0: LPS port poll-sequence context
- 1: LPS port weight context
- 2: LPS transmit-class status context

`u1TestType`:

- 0: WrRd.. pattern test
- 1: WrWr..RdRd.. pattern test

`u4Pattern`: Test pattern

Outputs None

Returns APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_TEST_PARAM

APX_ERR_POLL_TIMEOUT

APX_FAILURE

Side Effects The device is reset and put in the `APX_PRESENT` state

Testing Access to WPS Context-Memory: apexWpsCtxtTest

This function tests the microprocessor's access to the WPS context-memory aperture.

Prototype `INT4 apexWpsCtxtTest(APEX apex, UINT4 u4Pattern)`

Inputs `apex`: Device handle

`u4Pattern`: Test pattern

Outputs None

Returns

`APX_SUCCESS`

`APX_ERR_INVALID_DEV`

`APX_ERR_INVALID_TEST_PARAM`

`APX_ERR_POLL_TIMEOUT`

`APX_FAILURE`

Side Effects The device is reset and put in the `APX_PRESENT` state

Testing Access to the External SDRAM Cell-Buffers: apexCellBufTest

This function tests the microprocessor's access to the device's associated cell-buffer SDRAM. It does this by writing test-cell patterns to the SDRAM and reading them back.

Prototype

```
INT4 apexCellBufTest (APEX apex, UINT4 u4CellStartAddr, UINT4
u4NumCells, UINT1 u1TestType, UINT4 u4Pattern)
```

Inputs

`apex`: Device handle

`u4CellStartAddr:` Address in SDRAM to start the test from

`u4NumCells`: Number of cells

`u1TestType`:

- 0: WrWr..RdRd.. pattern test
- 1: Address aliasing test

`u4Pattern`: Test pattern (only applicable for pattern tests)

Outputs None

Returns

`APX_SUCCESS`

`APX_ERR_INVALID_DEV`

`APX_FAILURE`

`APX_INVALID_TEST_PARAM`

`APX_ERR_POLL_TIMEOUT`

Valid States

`APX_INIT`

`APX_ACTIVE`

Side Effects The device is reset and put into the `APX_PRESENT` state

Testing the Context Memory Image: apexCtxtMemCheck

This function tests the context memory image maintained by the driver. The driver compares the context memory image with the contents of the actual context memory.

Prototype `INT4 apexCtxtMemCheck(APEX apex)`

Inputs `apex`: Device handle

Outputs None

Returns
`APX_SUCCESS`
`APX_ERR_INVALID_DEV`
`APX_ERR_INVALID_STATE`
`APX_ERR_PORT_CTXT_CHK`
`APX_ERR_CLASS_CTXT_CHK`
`APX_ERR_CONN_CTXT_CHK`

Valid States
`APX_INIT`
`APX_ACTIVE`

Side Effects Can slow down device operations due to bottleneck at the memory port interface

8.10 Device Reset and Initialization Functions

This section describes the functions needed to reset and initialize S/UNI-APEX devices.

Resetting Devices: apexReset

This function applies a software reset to the S/UNI-APEX device. It also resets all the DDB contents (except for the initialization vector, which is not modified). Your application should call this function before initializing the device with a new initialization vector.

Prototype `INT4 apexReset(APEX apex)`

Inputs `apex`: Device handle

Outputs None

Returns

`APX_SUCCESS`

`APX_ERR_INVALID_DEV`

Side Effects The device state changes to `APX_PRESENT`. Therefore, your application must initialize the device after a reset.

Initializing Devices: apexInit

This function initializes the device based on an initialization vector passed by the application. This driver validates the vector and copies it into the DDB. Then the driver configures the device registers according to the contents of the initialization vector. Alternatively, you can also use an initialization vector profile number. In this case, driver copies the profile contents (stored in GDD) into the DDB. The driver has now finished initializing the device as per the profile contents.

Note: This function may modify the mask registers in the initialization vector supplied by your application

Prototype

```
INT4 apexInit(APEX apex, sAPX_INIT_VECT *psInitVect, UINT4
u4ProfileNum)
```

Inputs

`apex`: Device handle

`psInitVect`: The initialization vector that the driver uses to program the device registers. You should set this pointer to NULL if you are using an initialization vector profile.

`u4ProfileNum`: Profile number the driver will use to get the initialization vector from the GDD. You should set this variable to 0xffffffff if you are directly passing an initialization vector.

Outputs `None`

Returns

`APX_SUCCESS`

`APX_ERR_INVALID_DEV`

`APX_ERR_INVALID_STATE`

`APX_ERR_INVALID_INIT_VECTOR`

`APX_ERR_INVALID_PROFILE_NUM`

`APX_ERR_PROFILE_VECTOR_BOTH_VALID`

Valid States `APX_PRESENT`

Side Effects The device state changes to `APX_INIT`

8.11 Device Activation and Deactivation Functions

This section describes the functions needed to activate and deactivate S/UNI-APEX devices.

Activating Devices: apexActivate

This function activates the S/UNI-APEX device by preparing it for normal operation. Activation involves installing and enabling device interrupts; enabling the queue engine's external interfaces; and enabling the transmission and reception of cells and frames from the microprocessor port.

If this is the first device that the driver activates, the DPR task and the SAR tasks, along with the associated message queues, are created.

Prototype `INT4 apexActivate(APEX apex)`

Inputs `apex`: Device handle

Outputs None

Returns `APX_SUCCESS`

`APX_ERR_INVALID_DEV`

`APX_ERR_INVALID_STATE`

`APX_ERR_SAR_INSTALL`

Valid States `APX_INIT`

Side Effects The device state changes to `APX_ACTIVE`

Deactivating Devices: apexDeactivate

This function de-activates the S/UNI-APEX device and removes it from normal operation. Deactivation involves removing device interrupts; disabling the queue engine's external interfaces; and disabling transmission and reception of cells and frames from the microprocessor port.

Prototype	`INT4 apexDeactivate(APEX apex)`
Inputs	`apex`: Device handle
Outputs	None

Returns	`APX_SUCCESS`
	`APX_ERR_INVALID_DEV`
	`APX_ERR_INVALID_STATE`
	`APX_ERR_SAR_REMOVE`

Valid States	`APX_ACTIVE`
Side Effects	The device state changes to `APX_INIT`. If this is the last device that the driver is deactivating, then the driver also deletes the DPR task, the SAR-component transmit and receive tasks, and their associated message queues.

8.12 Queue Engine Functions

This section describes the queue engine functions that include:

- Setup, disable, re-enable, and teardown of ports, classes, and connections
- Updating the congestion thresholds and scheduling parameters for direction, ports, classes and connections
- Setup and teardown of shapers
- Watchdog patrol operations

8.13 Direction Functions

This section describes how to update the congestion thresholds for the Loop and WAN directions.

Updating Direction Thresholds: apexSetDirCongThrsh

This function updates the congestion thresholds for the Loop and WAN directions

Prototype `INT4 apexSetDirCongThrsh(APEX apex, UINT1 u1Dir, UINT1 u1Clp0Thrsh, UINT1 u1Clp1Thrsh, UINT1 u1MaxThrsh)`

Inputs

`apex:` Device handle

`u1Dir:` Direction for setting thresholds (Loop or WAN direction)

`u1Clp0Thrsh:` Value for CLP0 threshold

`u1Clp1Thrsh:` Value for CLP1 threshold

`u1MaxThrsh:` Value for max threshold

Outputs None

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_DIR

APX_ERR_INVALID_DIR_THRSH

Valid States

APX_INIT

APX_ACTIVE

8.14 Port Functions

This section describes how to set up, disable, re-enable, tear down ports and update the port thresholds and class scheduling parameters for a port, which has already been set up.

Setting Up Ports: apexPortSetup

This function configures and enables a port based on a port-parameters vector passed by the application. Alternatively, the application can pass the profile number of a port-vector already registered with the driver.

Prototype `INT4 apexPortSetup(APEX apex, sAPX_PORT_ID *psPortId, sAPX_PORT_VECT *psPortVect, UINT4 u4ProfileNum)`

Inputs

`apex`: Device handle

`psPortId`: Port to be configured

`psPortVect`: Port vector that the driver uses to program the port context records. You should set this pointer to NULL if you are using a port vector profile.

`u4ProfileNum`: Profile number the driver will use to get the port vector from the GDD. You should set this variable to 0xffffffff if you are instead directly passing a port vector.

Outputs

None

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_PORT_ID

APX_ERR_PORT_NOT_FREE

APX_ERR_PROFILE_VECTOR_BOTH_VALID

APX_ERR_INVALID_PORT_VECTOR

APX_ERR_INVALID_PROFILE_NUM

APX_ERR_POLL_TIMEOUT

Valid States

APX_INIT

APX_ACTIVE

Disabling Ports: apexPortDisable

This function disables an active port. If the port is already disabled, the function returns without doing anything.

Prototype

`INT4 apexPortDisable(APEX apex, sAPX_PORT_ID *psPortId)`

Inputs

`apex`: Device handle

`psPortId`: Port to be disabled

Outputs

None

Returns APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_PORT_ID

APX_ERR_PORT_NOT_CFG

APX_ERR_POLL_TIMEOUT

Valid States APX_ACTIVE

Side Effects Disables all associated classes and connections

Re-Enabling Ports: apexPortEnable

This function enables a disabled port. If the port is already enabled, the function returns without doing anything.

Prototype `INT4 apexPortEnable(APEX apex, sAPX_PORT_ID *psPortId)`

Inputs `apex`: Device handle

`psPortId`: Port to be enabled

Outputs None

Returns APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_PORT_ID

APX_ERR_PORT_NOT_CFG

APX_ERR_POLL_TIMEOUT

Valid States APX_ACTIVE

Side Effects Enables all associated classes and connections

Tearing Down Ports: apexPortTeardown

This function tears down a port and all additional classes and connections associated with the port.

Prototype `INT4 apexPortTeardown(APEX apex, sAPX_PORT_ID *psPortId)`

Inputs
`apex`: Device handle
`psPortId`: Port to be torn down

Outputs None

Returns
APX_SUCCESS
APX_ERR_INVALID_DEV
APX_ERR_INVALID_STATE
APX_ERR_INVALID_PORT_ID
APX_ERR_PORT_NOT_CFG
APX_ERR_POLL_TIMEOUT

Valid States APX_ACTIVE

Side Effects Tears down associated classes and connections

Updating Port Congestion Thresholds: apexSetPrtCongThrsh

This function updates the congestion thresholds for a port, which has already been set up.

Prototype `INT4 apexSetPrtCongThrsh(APEX apex, sAPX_PORT_ID *psPortId, UINT1 u1Clp0Thrsh, UINT1 u1Clp1Thrsh, UINT1 u1MaxThrsh)`

Inputs
`apex`: Device handle
`psPortId`: Port id
`u1Clp0Thrsh`: Value for CLP0 threshold
`u1Clp0Thrsh`: Value for CLP1 threshold
`u1MaxThrsh`: Value for max threshold

Outputs None

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_PORT_ID

APX_ERR_PORT_NOT_CFG

APX_ERR_INVALID_PORT_THRSH

Valid States

APX_INIT

APX_ACTIVE

Updating Class Scheduling Parameters: apexSetClSchd

This function updates the class scheduling parameters for a port, which has already been setup.

Prototype

```
INT4 apexSetClSchd(APEX apex, sAPX_PORT_ID *psPortId,
sAPX_CS_VECT *psCsVect)
```

Inputs

`apex`: Device handle

`psPortId`: Port id

`psCsVect`: Pointer to structure containing class scheduling parameters

Outputs

None

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_PORT_ID

APX_ERR_PORT_NOT_CFG

APX_ERR_INVALID_CL_SCHD

Valid States

APX_INIT

APX_ACTIVE

8.15 Class Functions

This section describes how to set up, disable, re-enable, tear down classes and updating class congestion thresholds for a class which has already been set up.

Setting Up Classes: apexClassSetup

This function configures and enables a class that is based on a class-parameters-vector passed by the application. Alternatively, the application can pass the profile number of a class-vector already registered with the driver. Note: The driver will not allow a class setup operation until your application enables the associated port.

Prototype

```
INT4 apexClassSetup(APEX apex, sAPX_CLASS_ID *psClassId,
sAPX_CLASS_VECT *psClassVect, UINT4 u4ProfileNum)
```

Inputs

`apex`: Device handle

`psClassId`: Port class to be configured

`psClassVect`: Class vector that the driver uses to program the class context record. You should set this pointer to NULL if you are using a class vector profile.

`u4ProfileNum`: Profile number the driver will use to get the port vector from the GDD. You should set this variable to 0xffffffff if you are directly passing a port vector.

Outputs

None

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_CLASS_ID

APX_ERR_PORT_NOT_ENABLED

APX_ERR_CLASS_NOT_FREE

APX_ERR_PROFILE_VECTOR_BOTH_VALID

APX_ERR_INVALID_CLASS_VECTOR

APX_ERR_INVALID_PROFILE_NUM

APX_ERR_POLL_TIMEOUT

Valid States

APX_INIT

APX_ACTIVE

Disabling Classes: apexClassDisable

This function disables a configured class.

Prototype `INT4 apexClassDisable(APEX apex, sAPX_CLASS_ID *psClassId)`

Inputs

`apex`: Device handle

`psClassId`: Class to be disabled

Outputs None

Returns

`APX_SUCCESS`

`APX_ERR_INVALID_DEV`

`APX_ERR_INVALID_STATE`

`APX_ERR_INVALID_CLASS_ID`

`APX_ERR_CLASS_NOT_CFG`

`APX_ERR_POLL_TIMEOUT`

Valid States `APX_ACTIVE`

Side Effects Disables all associated connections

Re-Enabling Classes: apexClassEnable

This function enables a disabled class. If the class is enabled, the function returns without doing anything. Note: The driver will not allow a class-enable operation unless your application has enabled the associated port.

Prototype `INT4 apexClassEnable(APEX apex, sAPX_CLASS_ID *psClassId)`

Inputs

`apex`: Device handle

`psClassId`: Class to be enabled

Outputs None

Returns APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_CLASS_ID

APX_ERR_PORT_NOT_ENABLED

APX_ERR_CLASS_NOT_CFG

APX_ERR_POLL_TIMEOUT

Valid States APX_ACTIVE

Side Effects Enables all associated connections

Tearing Down Classes: apexClassTeardown

This function first tears down all connections associated with this class, thereafter tearing down the class itself.

Prototype `INT4 apexClassTeardown(APEX apex, sAPX_CLASS_ID *psClassId)`

Inputs `apex`: Device handle

`psClassId`: Class to be torn down

Outputs None

Returns APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_CLASS_ID

APX_ERR_CLASS_NOT_CFG

APX_ERR_POLL_TIMEOUT

Valid States APX_ACTIVE

Side Effects Tears down all associated connections

Updating Class Congestion Thresholds: apexSetClCongThrsh

This function updates the congestion thresholds for a class, which has already been set up.

Prototype `INT4 apexSetClCongThrsh(APEX apex, sAPX_CLASS_ID *psClassId, UINT1 u1Clp0Thrsh, UINT1 u1Clp1Thrsh, UINT1 u1MaxThrsh)`

Inputs

`apex`: Device handle

`psClassId`: Class id

`u1Clp0Thrsh`: Value for CLP0 threshold

`u1Clp0Thrsh`: Value for CLP1 threshold

`u1MaxThrsh`: Value for max threshold

Outputs None

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_CLASS_ID

APX_ERR_CLASS_NOT_CFG

APX_ERR_INVALID_CLASS_THRSH

Valid States

APX_INIT

APX_ACTIVE

8.16 Shaper Functions

This section describes the functions that set up and tear down shapers.

Setting Up Shapers: apexShprSetup

This function configures and enables a shaper based on a shaper parameters-vector passed by the application.

Note: Your application should configure shapers before activating a device and before configuring the associated port-classes

Prototype `INT4 apexShprSetup(APEX apex, UINT1 u1ShprId, sAPX_SHPR_VECT *psShprVect)`

Inputs `apex`: Device handle

`u1ShprId`: Shaper to be configured (0-3)

`psShprVect`: Shaper parameters vector the driver uses to program the port context records

Outputs None

Returns APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_SHPR_ID

APX_ERR_SHPR_NOT_FREE

APX_ERR_INVALID_SHPR_VECTOR

APX_ERR_POLL_TIMEOUT

Valid States APX_INIT

Tearing Down Shapers: apexShprTeardown

This function tears down a shaper. Shapers can only be torn down if the queue engine is disabled (the device is de-activated). Note: A shaper should be torn down only after its associated port-class is torn down.

Prototype `INT4 apexShprTeardown(APEX apex, UINT1 u1ShprId)`

Inputs `apex`: Device handle

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

Outputs None

Returns
APX_SUCCESS
APX_ERR_INVALID_DEV
APX_ERR_INVALID_STATE
APX_ERR_INVALID_SHPR_ID
APX_ERR_SHPR_NOT_FREE
APX_ERR_POLL_TIMEOUT

Valid States APX_INIT

8.17 Connection Functions

This section describes how to set up, disable, re-enable, and tear down connections. It also describes how to update the congestion thresholds, WFQ weights and shape single rate parameters for connections that have already been set up.

Setting Up Connections: apexConnSetup

This function configures and enables a connection based on a connection-parameters vector passed by the application. Alternatively, the application can pass the profile number of a connection-vector already registered with the driver. Note: The driver will not allow a connection setup operation until your application has enabled the associated port and class. The connection vector must contain shape fair queue (SFQ) parameters if the class has been configured to be shaped.

Prototype
```
INT4 apexConnSetup(APEX apex, sAPX_CONN_ID *psConnId,
sAPX_CONN_VECT *psConnVect, UINT4 u4ProfileNum)
```

Inputs

`apex`: Device handle

`psConnId`: Connection to be configured

`psConnVect`: Connection vector that the driver uses to program the VC context records. You should set this pointer to NULL if you are using a connection vector profile.

`u4ProfileNum`: The driver uses this profile number to get the connection vector from the GDD. You should set this variable to 0xffffffff if you are directly passing a connection vector.

Outputs None

Returns

APX_SUCCESS
APX_ERR_INVALID_DEV
APX_ERR_INVALID_STATE
APX_ERR_INVALID_CONN_ID
APX_ERR_INVALID_CONN_VECTOR
APX_ERR_INVALID_PROFILE_NUM
APX_ERR_PROFILE_VECTOR_BOTH_VALID
APX_ERR_PORT_NOT_ENABLED
APX_ERR_CLASS_NOT_ENABLED
APX_ERR_CONN_NOT_FREE
APX_ERR_POLL_TIMEOUT

Valid States

APX_INIT
APX_ACTIVE

Disabling Connections: apexConnDisable

This function disables a configured connection.

Prototype

`INT4 apexConnDisable(APEX apex, UINT4 u4ICI)`

Inputs

`apex`: Device handle
`u4ICI`: ICI of the connection to be disabled

Outputs

None

Returns

APX_SUCCESS
APX_ERR_INVALID_DEV
APX_ERR_INVALID_STATE
APX_ERR_INVALID_ICI
APX_ERR_CONN_NOT_CFG
APX_ERR_POLL_TIMEOUT

Valid States

APX_ACTIVE

Re-Enabling Connections: apexConnEnable

This function enables a disabled connection. If the connection is already enabled, the function returns without doing anything.

Note: The driver will not allow a connection enable operation unless your application has enabled the associated port and class.

Prototype `INT4 apexConnEnable(APEX apex, UINT4 u4ICI)`

Inputs

`apex`: Device handle

`u4ICI`: ICI of the connection to be enabled

Outputs None

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_PORT_NOT_ENABLED

APX_ERR_CLASS_NOT_ENABLED

APX_ERR_CONN_NOT_CFG

APX_ERR_INVALID_ICI

APX_ERR_POLL_TIMEOUT

Valid States APX_ACTIVE

Tearing Down Connections: apexConnTeardown

This function tears down a configured connection. If the connection is currently active, the driver first disables it, and then tears it down.

Prototype `INT4 apexConnTeardown(APEX apex, UINT4 u4ICI)`

Inputs

`apex`: Device handle

`u4ICI`: ICI of the connection to be disabled

Outputs None

Returns
APX_SUCCESS
APX_ERR_INVALID_DEV
APX_ERR_INVALID_STATE
APX_ERR_INVALID_ICI
APX_ERR_CONN_NOT_CFG
APX_ERR_POLL_TIMEOUT

Valid States
APX_ACTIVE

Updating Connection Congestion Thresholds: apexSetConnCongThrsh

This function updates the congestion thresholds for an already set-up connection.

Prototype

```
INT4 apexSetConnCongThrsh(APEX apex, UINT4 u4ICI,
UINT1 u1Clp0Thrsh, UINT1 u1Clp1Thrsh, UINT1 u1MaxThrsh)
```

Inputs
`apex`: Device handle
`u4Ici`: Connection id
`u1Clp0Thrsh`: Value for CLP0 threshold
`u1Clp0Thrsh`: Value for CLP1 threshold
`u1MaxThrsh`: Value for max threshold

Outputs
None

Returns
APX_SUCCESS
APX_ERR_INVALID_DEV
APX_ERR_INVALID_STATE
APX_ERR_INVALID_ICI
APX_ERR_CONN_NOT_CFG
APX_ERR_INVALID_CONN_THRSH

Valid States
APX_INIT
APX_ACTIVE

Updating Class Queuing Weight: apexSetConnWfqWt

This function updates the class queuing weight for a connection with queue type WFQ.

Prototype `INT4 apexSetConnWfqWt(APEX apex, UINT4 u4ICI, UINT1 u1Wt)`

Inputs
`apex`: Device handle
`u4Ici`: Connection id
`u1Wt`: Class queue weight for WFQ connection

Outputs None

Returns
APX_SUCCESS
APX_ERR_INVALID_DEV
APX_ERR_INVALID_STATE
APX_ERR_INVALID_ICI
APX_ERR_CONN_NOT_CFG
APX_ERR_INVALID_CONN_TYPE
APX_ERR_INVALID_WFQ_WT

Valid States
APX_INIT
APX_ACTIVE

Updating Shaped Single Rate Parameters: apexSetConnShpSnglRt

This function updates the shape single rate parameters for a shaped connection.

Prototype
```
INT4 apexSetConnShpSnglRt(APEX apex, UINT4 u4ICI,
UINT1 u1ShpPrescale, UINT2 u2ShpLateBits, UINT2 u2ShpCdvt,
UINT2 u2ShpIncr)
```

Inputs
`apex`: Device handle
`u4ICI`: Connection id
`u1ShpPrescale`: Determines resolution of shape increment
`u2ShpLateBits`: Number of bits required to represent ShpTxSlotLate field
`u2ShpCdvt`: Cell delay variance tolerance
`u2ShpIncr`: Shape increment

Outputs None

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_ICI

APX_ERR_CONN_NOT_CFG

APX_ERR_INVALID_CONN_TYPE

APX_ERR_INVALID_SHP_PARAM

Valid States

APX_INIT

APX_ACTIVE

8.18 Watchdog Patrol Functions

Setting Watchdog Patrol Parameters: apexSetWdgPatrolRng

This function sets the ICI watchdog patrol range.

Prototype

```
INT4 apexSetWdgPatrolRng(APEX apex, UINT2 u2StartICI,
                         UINT2 u2EndICI)
```

Inputs

`apex`: Device handle

`u2StartICI`: First ICI in watchdog patrol range

`u2EndICI`: Last ICI in watchdog patrol range

Outputs

None.

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_RANGE

APX_ERR_INVALID_ICI

APX_ERR_WDG_PTRL_BUSY

Valid States

APX_INIT

APX_ACTIVE

Getting Watchdog Patrol Parameters: apexGetWdgPatrolRng

This function gets the current settings for the ICI watchdog patrol range.

Prototype

```
INT4 apexGetWdgPatrolRng(APEX apex, UINT2 *pu2StartICI,
                         UINT2 *pu2EndICI)
```

Inputs

`apex`: Device handle

Outputs

`pu2StartICI`: Pointer to first ICI in watchdog patrol range

`pu2EndICI`: Pointer to last ICI in watchdog patrol range

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

Valid States

APX_INIT

APX_ACTIVE

Initiating a Watchdog Patrol: apexWatchdogPatrol

This function can be used to invoke the APEX watchdog macro that checks a specified range of ICIs (frame continuous queuing VCs) for frame re-assembly timeouts. If at least one VC has timed out, the APEX generates an interrupt and stores the last found ICI in the miscellaneous context record.

Prototype

```
INT4 apexWatchdogPatrol(APEX apex)
```

Inputs

`apex`: Device handle

Outputs

None

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_POLL_TIMEOUT

APX_ERR_WDG_PTRL_BUSY

Valid States APX_ACTIVE

8.19 Segmentation and Re-assembly Assist Functions

This section describes the segmentation and re-assembly (SAR) assist functions.

Transmitting Cells: apexTxCell

Your application can use this function to transmit cells from the device's SAR interface. This function encapsulates the cell information (header, payload, and so on) in a message structure and sends it to the driver's SAR transmit task. The SAR transmit task transmits the cell and sends a transmit-cell indication, `indTxCell`, back to your application.

Prototype

```
INT4 apexTxCell(APEX apex, UINT4 u4ICI, sAPX_CELL_HDR *psHdr,
UINT1 *pu1Pyld, UINT1 u1CrcFlg)
```

Inputs

`apex`: Device handle

`u4ICI`: ICI of the connection

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

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

`u1CrcFlg`: A control flag, which can be:

- 0: No CRC protection required
- 1: Overwrite end-of-cell with CRC-10

Outputs None

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_SAR_TX_MSG

APX_ERR_INVALID_ICI

APX_ERR_CONN_NOT_CFG

Valid States APX_ACTIVE

Transmitting AAL5 Frames: apexTxFrm

Your application can use this function to transmit AAL5 frames using the device's SAR interface. This function forms an AAL5 PDU from the frames you provide (padding, CRC, and AAL5 trailer) for transmission on a specified connection. The AAL5 PDU is then queued for transmission. After the driver completes transmission, it reports the results of the transmission via the indication call back, `indTxFrm`.

Prototype `INT4 apexTxFrm(APEX apex, UINT4 u4ICI, sAPX_CELL_HDR *psHdr, UINT1 *pu1Frm, UINT4 u4Len)`

Inputs

`apex`: Device handle

`u4ICI`: ICI of the connection that will carry the frame

`psHdr`: Cell header that will ride with each cell in the frame

`pu1Frm`: Pointer to first byte of frame (buffer chain)

`u4Len`: Frame length (in bytes)

Outputs None

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_SAR_TX_FRM_LENGTH

APX_ERR_INVALID_ICI

APX_ERR_CONN_NOT_CFG

APX_ERR_SAR_TX_MSG

Valid States APX_ACTIVE

SAR Transmit Task Function: apexSarTxTaskFn

This function represents the SAR transmit operation. It executes in the context of a separate task within the RTOS. Your implementation of the system-specific function, `sysApexSarTxTaskFn`, should invoke this function. This function will determine whether the information passed to it is cell information or frame information and will call the relevant functions to transmit a cell or a frame. After the transmission is complete, it will report the results of the transmission via the indication callback, `indTxCell` or `indTxFrm`.

Prototype `INT4 apexSarTxTaskFn(APX_TX_CTXT sTxCtxt)`

Inputs `sTxCtxt`: Structure containing the following information:

- `apex`: Device handle
- `txType`: Determines whether information is for cell or frame
- `cellInfo`: Cell information if txType indicates a cell
- `frmInfo`: Frame information if txType indicates a frame

Outputs None

Returns
APX_SUCCESS
APX_ERR_INVALID_DEV
APX_ERR_INVALID_STATE
APX_ERR_SAR_TX_TYPE
APX_ERR_SAR_TX_BUSY
APX_ERR_SAR_TX_NXT_FRM_BUF

Valid States APX_ACTIVE

SAR Receive Task Function: apexSarRxTaskFn

This function represents the SAR receive operation. It executes in the context of a separate task within the RTOS. Your implementation of the system-specific function, `sysApexSarRxTaskFn`, should invoke this function. The function will go through the four class queues in the order of priority that you specify. It will then read the cell header to determine whether it has to read a cell or an AAL5 frame and call the appropriate function to extract the cell or frame. After the extraction is complete, the function will invoke the indication callback function, `indRxCell` or `indRxFrm`, to notify the application.

Prototype `INT4 apexSarRxTaskFn(APEX apex)`

Inputs `apex`: Device handle

Outputs None

Returns APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_SAR_RX_CRC10_FAIL

APX_ERR_SAR_RX_BUF_FULL

Valid States APX_ACTIVE

8.20 Multicasting Support Functions

This section describes the functions that install and reset the multicasting callback function.

Installing the Multicasting Callback Function: apexInstallMulticastFn

Installs a user provided function pointer as the multicast callback function. The installed function is invoked each time a cell/frame is received by the SAR Rx task. The callback function is responsible for determining whether the connection on which the cell/frame is received is part of a multicasting group. If so, it returns a list of connection IDs for the outgoing connections. The SAR Rx task then transmits the received cell/frame on these outgoing connections.

Prototype

```
INT4 apexInstallMulticastFn(APEX apex,
APX_MULTICAST_CB_FN multicastCbFn)
```

Inputs `apex`: Device handle

`multicastFn`: pointer to the multicasting callback function

Outputs None

Returns APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

Valid States APX_INIT

APX_ACTIVE

Side Effects Enables the multicasting support provided by the driver

Resetting the Multicasting Callback Function: apexResetMulticastFn

This function is used to reset the multicasting callback function to a null pointer.

Prototype `INT4 apexResetMulticastFn(APEX apex)`

Inputs `apex`: Device handle

Outputs None

Returns `APX_SUCCESS`

`APX_ERR_INVALID_DEV`

`APX_ERR_INVALID_STATE`

Valid States `APX_INIT`

`APX_ACTIVE`

Side Effects Disables the multicasting support provided by the driver

8.21 Loop Port Scheduler Functions

Setting Contents of the Port-Weight Table: apexLpsSetPortWts

This function sets the LPS port weight table contents.

Prototype
```
INT4 apexLpsSetPortWts(APEX apex, UINT4 u4NumPorts,
sAPX_PORT_WT *psPortWtTable)
```

Inputs `apex`: Device handle

`u4NumPorts`: Number of ports

`psPortWtTable`: Pointer to structure containing port numbers and the corresponding weights

Outputs None

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_PORT_ID

APX_ERR_PORT_NOT_CFG

APX_ERR_LPS_INVALID_WT

Valid States

APX_INIT

APX_ACTIVE

Getting Contents of the Port-Weight Table: apexLpsGetPortWts

This function retrieves the contents of the LPS port weight table contents.

Prototype

```
INT4 apexLpsGetPortWts(APEX apex, UINT4 u4NumPorts, UINT4
u4PortStart, sAPX_PORT_WT *psPortWtTable)
```

Inputs

`apex`: Device handle

`u4NumPorts`: Number of ports

`u4PortStart`: Starting port number

Outputs

`psPortWtTable`: Pointer to the port weight table, which contains the port numbers and weights

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_PORT_ID

Valid States

APX_INIT

APX_ACTIVE

Setting Contents of the Poll Sequence Table: apexLpsSetPollSeq

This function sets the LPS port weight table contents.

Prototype

```
INT4 apexLpsSetPollSeq(APEX apex, UINT4 u4NumPorts,
sAPX_PORT_SEQ *psPortSeqTable)
```

Inputs

`apex`: Device handle

`u4NumPorts`: Number of ports

`psPortSeqTable`: Pointer to structure containing port numbers and the corresponding sequence numbers

Outputs

None

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_PORT_ID

APX_ERR_PORT_NOT_CFG

APX_ERR_LPS_INVALID_SEQ

Valid States

APX_INIT

APX_ACTIVATE

Getting Contents of the Poll Sequence Table: apexLpsGetPollSeq

This function retrieves the contents of the LPS port weight table contents.

Prototype

```
INT4 apexLpsGetPollSeq(APEX apex, UINT4 u4NumPorts, UINT4
u4PortStart, sAPX_PORT_SEQ *psPortSeqTable)
```

Inputs

`apex`: Device handle

`u4NumPorts`: Number of ports

`u4PortStart`: Starting port number

Outputs

`psPortSeqTable`: Pointer to the port sequence table, which contains the port numbers and sequence numbers. If the port is not configured, then the poll sequence is set to `0xff`.

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_PORT_ID

Valid States APX_INIT

APX_ACTIVE

8.22 WAN Port Scheduler Functions

Setting Contents of the Port-Weight Table: apexWpsSetPortWts

This function sets the WPS port weight table contents.

Prototype

```
UINT4 apexWpsSetPortWts(APEX apex, UINT4 u4NumPorts,
sAPX_PORT_WT *psPortWtTable)
```

Inputs `apex`: Device handle

`u4NumPorts`: Number of ports

`psPortWtTable`: Pointer to structure containing port numbers and the corresponding weights

Outputs None

Returns APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_PORT_ID

APX_ERR_PORT_NOT_CFG

APX_ERR_WPS_INVALID_WT

Valid States APX_INIT

APX_ACTIVE

Getting Contents of the Port-Weight Table: apexWpsGetPortWts

This function retrieves the contents of the WPS port weight table contents.

Prototype

```
UINT4 apexWpsGetPortWts(APEX apex, sAPX_PORT_WT *psPortWtTable)
```

Inputs apex: Device handle

Outputs psPortWtTable: Pointer to the port weight table, which contains the port numbers and weights. If the port is not configured then the port weight is set to `0xff`.

Returns `APX_SUCCESS`

`APX_ERR_INVALID_DEV`

`APX_ERR_INVALID_STATE`

Valid States `APX_INIT`

`APX_ACTIVE`

8.23 Statistic Functions

The S/UNI-APEX device provides two types of device counts: statistical counts and congestion counts. The statistical counts are counts that increase monotonically as they accumulate over time. The congestion counts are snapshots of the current congestion counts. They need not increase monotonically. The following functions retrieve these device counts for the application. By periodically invoking these functions, the application can maintain a steady count of the types mentioned.

8.24 Statistical Counts

Getting Cell Discard Counts: apexGetStatDiscardCnts

This function retrieves the discarded cell counts accumulated by the S/UNI-APEX device. These counts include the number of `CLP0` and `CLP1` cells discarded to congestion, as well as the number cells discarded for reasons other than congestion.

Prototype

```
INT4 apexGetDiscardCnts(APEX apex, UINT4 *pu4DiscardCnt, UINT4
*pu4Clp0DiscardCnt, UINT4 *pu4Clp1DiscardCnt)
```

Inputs `apex`: Device handle

Outputs

`pu4DiscardCnt`: General discard count of all cells that have been discarded due to reasons other than congestion (such as re-assembly timeout and re-assembly maximum-length error)

`pu4Clp0DiscardCnt`: Count of all CLP0 cells discarded due to congestion

`pu4Clp1DiscardCnt`: Count of all CLP1 cells discarded due to congestion

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_POLL_TIMEOUT

Valid States

APX_INIT

APX_ACTIVE

Getting Connection-Level Cell-Transmission Counts: apexGetStatConnTxCnts

This function retrieves the connection-level cell-transmission counts.

Prototype

```
INT4 apexGetStatConnTxCnts(APEX apex, UINT4 u4ICI, UINT4
*pu4VcClp0TxCnt, UINT4 *pu4VcClp1TxCnt)
```

Inputs

`apex`: Device handle

`u4ICI`: ICI of the connection

Outputs

`pu4VcClp0TxCnt`: Count of all CLP0 cells transmitted

`pu4VcClp1TxCnt`: Count of all CLP1 cells transmitted

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_ICI

APX_ERR_CONN_NOT_CFG

APX_ERR_POLL_TIMEOUT

Valid States APX_INIT
APX_ACTIVE

8.25 Congestion Counts

Getting Device-Level Congestion Counts: apexGetCongDevCnt

This function returns the total number of cells available for buffering in the device (FreeCnt).

Prototype `INT4 apexGetCongDevCnt(APEX apex, UINT4 *pu4Cnt)`

Inputs `apex`: Device handle

Outputs `pu4Cnt`: Snapshot of `FreeCnt`

Returns APX_SUCCESS
APX_ERR_INVALID_DEV
APX_ERR_INVALID_STATE
APX_ERR_POLL_TIMEOUT

Valid States APX_INIT
APX_ACTIVE

Getting Direction-Level Congestion Counts: apexGetCongDirCnt

This function retrieves the count of cells queued for all loop and WAN ports.

Prototype `INT4 apexGetCongDirCnt(APEX apex, UINT1 u1Dir, UINT4 *pu4Cnt)`

Inputs `apex`: Device handle
`u1Dir`:

- 1: Loop
- 2: WAN

Outputs `pu4Cnt`: Loop/WAN cells queue count

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_PORT_ID

APX_ERR_POLL_TIMEOUT

Valid States

APX_INIT

APX_ACTIVE

Getting Port-Level Congestion Counts: apexGetCongPortCnt

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

Prototype

```
INT4 apexGetCongPortCnt(APEX apex, sAPX_PORT_ID *psPortId,
UINT4 *pu4Cnt)
```

Inputs

`apex`: Device handle

`psPortId`: Port type (loop, WAN, uP) and number

Outputs

`pu4Cnt`: Cells queued for this port

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_PORT_ID

APX_ERR_PORT_NOT_CFG

APX_ERR_POLL_TIMEOUT

Valid States

APX_INIT

APX_ACTIVE

Getting Class-Level Congestion Counts: apexGetCongClassCnt

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

Prototype

```
INT4 apexGetCongClassCnt(APEX apex, sAPX_CLASS_ID *psClassId,
UINT4 *pu4Cnt)
```

Inputs `apex`: Device handle

`psClassId`: Port-class identifier

Outputs `pu4Cnt`: Cells queued for this class

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_CLASS_ID

APX_ERR_CLASS_NOT_CFG

APX_ERR_POLL_TIMEOUT

Valid States

APX_INIT

APX_ACTIVE

Getting Connection-Level Congestion Counts: apexGetCongConnCnts

This function retrieves the following congested-connection counts:

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

Prototype

```
INT4 apexGetCongConnCnts(APEX apex, UINT4 u4ICI, UINT4
*pu4VcClp0Cnt, UINT4 *pu4VcQClp01Cnt, UINT4
*pu4VcClassQClp01Cnt)
```

Inputs `apex`: Device handle

`u4ICI`: ICI of the connection

Outputs `pu4VcClp0Cnt`: Snapshot of `VcCLP0Cnt`

`pu4VcQClp01Cnt`: Snapshot of `VcQCLP01Cnt`

`pu4VcClassQClp01Cnt`: Snapshot of `VcClassQCLP01Cnt`

Returns `APX_SUCCESS`

`APX_ERR_INVALID_DEV`

`APX_ERR_INVALID_STATE`

`APX_ERR_INVALID_ICI`

`APX_ERR_CONN_NOT_CFG`

`APX_ERR_POLL_TIMEOUT`

Valid States `APX_INIT`

`APX_ACTIVE`

8.26 Interrupt Service Functions

This section describes interrupt service functions that perform the following tasks:

- Read and process high-priority interrupt-status registers
- Read and process low-priority interrupt-status registers
- Set and get interrupt masks
- Enable and disable interrupts
- Get and reset interrupt counts
- Set interrupt-count thresholds

Servicing High-Priority Interrupts: apexHiISR

This function reads the high priority interrupt status register of the interrupting device and compares it with the mask that you define for this register (logical AND operation). If there are any valid bits set in this register, this function returns a value greater than zero. If there are no bits set, this function returns a zero. The system-specific interrupt handler routine, `sysApexHiIntHandler`, invokes this function.

Prototype `UINT4 apexHiISR (APEX apex, UINT4 *pu4Stat)`

Inputs `apex`: Device handle

Outputs `pu4Stat`: Valid interrupt conditions detected in high-priority interrupt status register

Returns = 0: No valid interrupt conditions detected

> 0: At least one valid interrupt condition detected

Valid States `APX_ACTIVE`

Side Effects If this function returns a non-zero value (meaning the driver detected an interrupt condition) then all high-priority device interrupts are disabled

Servicing Low-Priority Interrupts: apexLoISR

This function reads the low priority interrupt error and status registers of the interrupting device and compares the contents with the corresponding masks that you define (logical AND operations). If there are any bits set in these registers, this function returns a value greater than zero. Otherwise, it returns a zero. The system-specific interrupt handler routine, `sysApexLoIntHandler`, invokes this function.

Prototype `UINT4 apexLoISR(APEX apex, UINT4 *pu4Err, UINT4 *pu4Stat)`

Inputs `apex`: Device handle

Outputs `pu4Err`: Valid interrupt conditions detected in low-priority interrupt error-register

`pu4Stat`: Valid interrupt conditions detected in low-priority interrupt status-register

Returns = 0: No valid interrupt conditions detected

> 0: At least one valid interrupt condition detected

Valid States `APX_ACTIVE`

Side Effects If this function returns a non-zero value (meaning the driver detected an interrupt condition), then all low-priority device interrupts are disabled

Processing High-Priority Interrupt-Status Information: apexHiDPR

This function processes the high-priority interrupt status information sent to the DPR task by the hi-priority ISR routine. Processing involves updating the interrupt counters corresponding to the interrupt events sent by the ISR. It also involves invoking the `indCritical` callback, which informs the application of the events that have crossed their thresholds. The system-specific DPR function, `sysApexDPRtask`, invokes this function.

Prototype `UINT4 apexHiDPR(APEX apex, UINT4 u4Stat)`

Inputs `apex`: Device handle

`u4Stat`: Interrupt conditions detected by `apexHiISR` in the high-priority interrupt-status register

Outputs None

Returns `APX_SUCCESS`

Valid States `APX_ACTIVE`

Side Effects Enables high-priority interrupts processing after servicing all existing interrupt conditions

Processing Low-Priority Interrupt-Status Information: apexLoDPR

This function processes the low-priority interrupt error information sent to the DPR task by the low-priority ISR routine. Processing involves updating the interrupt counters corresponding to the interrupt events sent by the ISR. It also involves invoking the `indError` callback, which informs the application of the events that before have crossed their thresholds. The system-specific DPR task routine, `sysApexDPRtask`, invokes this function.

Prototype `UINT4 apexLoDPR(APEX apex, UINT4 u4Err)`

Inputs `apex`: Device handle

`u4Err`: Interrupt conditions detected by `apexLoISR` in the low-priority interrupt-error register

Outputs None

Returns APX_SUCCESS

Valid States APX_ACTIVE

Side Effects Enables low-priority interrupts processing after servicing all existing interrupt conditions

Setting Interrupt Masks: apexSetIntMsk

This function sets the desired interrupt masks for the device's interrupt registers located in the ISM control block. The driver writes these masks to the device registers when the driver enables interrupt processing for the device.

Note: The driver masks `MpIdle`, `SarRxRdy`, and `SarRxEmpty`, as well as all the reserved bits in the mask specified by your application.

Prototype `INT4 apexSetIntMsk(APEX apex, UINT1 u1Ctrl, sAPX_INTS *psMskVal)`

Inputs

`apex`: Device handle

`u1Ctrl`: Specifies which mask register(s) to set:

- APX_HI_INT
- APX_LO_ERROR_INT
- APX_LO_STAT_INT
- APX_ALL_INTS

`psMskVal`: Mask value(s) to be set. Only those masks will be set that the driver specifies in `u1Ctrl`.

Outputs None

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_MSK_ID

Valid States

APX_INIT

APX_ACTIVE

Getting Interrupt Masks: apexGetIntMsk

This function returns the interrupt masks set by the application from the ISM control block.

Prototype `INT4 apexGetIntMsk(APEX apex, sAPX_INTS *psMskVal)`

Inputs `apex`: Device handle

Outputs `psMskVal`: Mask values for the three interrupt-mask registers (you allocate this structure)

Returns
`APX_SUCCESS`
`APX_ERR_INVALID_DEV`
`APX_ERR_INVALID_STATE`

Valid States
`APX_INIT`
`APX_ACTIVE`

Enabling and Disabling Interrupts: apexIntCtrl

This function enables and disables device interrupts by directly writing to the interrupt mask registers of the S/UNI-APEX device.

Prototype `INT4 apexIntCtrl(APEX apex, UINT1 u1EnFlg, UINT1 u1Ctrl)`

Inputs `apex`: Device handle

u1EnFlg:

- `APX_ENABLE`
- `APX_DISABLE`

`u1Ctrl`: Specifies which interrupt to enable or disable:

- `APX_HI_INT`
- `APX_LO_ERROR_INT`
- `APX_LO_STAT_INT`
- `APX_ALL_INTS`

Outputs None

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_FLAG

APX_ERR_INVALID_CTRL_PARAM

Valid States

APX_INIT

APX_ACTIVE

Getting Interrupt Counts: apexGetIntCnts

This function returns the interrupt event counts for the high-priority status and low-priority error interrupt event-counters.

Prototype

```
INT4 apexGetIntCnts(APEX apex, UINT4 *pu4HiCnts, UINT4
*pu4LoErrCnts)
```

Inputs

`apex`: Device handle

Outputs

`pu4HiCnts`: Pointer to an array of 32 words, which you allocate. The driver fills in the elements of the array corresponding to the valid high-priority interrupt events.

`pu4LoErrCnts`: Pointer to an array of 32 words, which you allocate. The driver fills in the elements of the array corresponding to the valid low-priority error interrupt events.

Note: The application should retrieve the valid counts from the array by using the valid interrupt event definitions (`apx_api.h`) corresponding to enabled interrupts.

Returns

APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

Valid States

APX_INIT

APX_ACTIVE

Resetting Interrupt Counters: apexResetIntCnts

This function resets the interrupt event counters to zero.

Prototype `INT4 apexResetIntCnts(APEX apex)`

Inputs `apex`: Device handle

Outputs None

Returns `APX_SUCCESS`

`APX_ERR_INVALID_DEV`

`APX_ERR_INVALID_STATE`

Valid States `APX_INIT`

`APX_ACTIVE`

Setting Interrupt-Count Thresholds: apexSetIntThresh

This function sets thresholds for the interrupt event counters corresponding to the interrupt bits in the high-priority status and low-priority error interrupt registers. When an interrupt event-counter crosses its threshold, the driver's DPR task invokes a callback (`indCritical` for hi-priority, `indError` for low-priority interrupt events) that informs the application about the event(s) that crossed their thresholds.

Prototype `INT4 apexSetIntThresh(APEX apex, UINT1 u1IntType, UINT1 u1EvtId, UINT4 u4Thrsh)`

Inputs `apex`: Device handle

`u1IntType`:

- `APX_HI_INT`
- `APX_LO_ERR_INT`

`u1EvtId`: Event for which threshold is to be set

`u2Thrsh`: Threshold value to be set for the event

Outputs None

Returns APX_SUCCESS

APX_ERR_INVALID_DEV

APX_ERR_INVALID_STATE

APX_ERR_INVALID_INT_TYPE

Valid States APX_INIT

APX_ACTIVE

8.27 Application Callback Functions

The S/UNI-APEX driver uses the following application callback functions to notify the application of events within the device and driver.

Indicating the Success or Failure of Cell Transmissions: indTxCell

The segmentation and re-assembly (SAR) assist transmit task uses this callback to confirm the success or failure of a cell transmission request made by the application. Pointers to the cell header and payload are passed to the application. The application should de-allocate the cell buffer payload and header.

Prototype `void indTxCell(USR_CTXT usrCtxt, UINT4 u4ICI, sAPX_CELL_HDR *psHdr, UINT1 *pu1Pyld, INT4 result)`

Inputs `usrCtxt`: Pointer to device context information, which the application maintains

`u4ICI`: ICI on which cell was transmitted

`psHdr`: Header of the transmitted cell

`psPyld`: Payload of the transmitted cell

`result`:

- 0 : Success
- <0 : Failure
  - APX_ERR_SAR_TX_BUSY (SAR TX is busy)

Outputs None

Returns None

Indicating the Success or Failure of Cell Receptions: indRxCell

This function is invoked by the SAR receive task after it extracts a cell from the microprocessor interface. The application should free the cell header and payload buffers.

Prototype

```
void indRxCell(USR_CTXT usrCtxt, UINT4 u4ECI, sAPX_CELL_HDR
*psHdr, UINT1 *pu1Pyld, INT4 result)
```

Inputs

`usrCtxt`: Pointer to device context information, which the application maintains

`u4ECI`: ICI of the connection on which cell was received

`psHdr`: Header of the received cell

`psPyld`: Payload of the received cell

result:

- 0 : Success
- <0 : Failure
  - `APX_ERR_SAR_RX_CELL_BUF_FULL` (Cell buffers are full)
  - `APX_ERR_SAR_RX_CRC10_FAIL` (CRC10 failure in OAM cell)

Outputs None

Returns None

Indicating the Success or Failure of Frame Transmissions: indTxFrm

This function is used by the SAR transmit task to confirm the success/failure of an AAL5 frame transmission request made by the application. A pointer to the first byte of the AAL5 frame buffer chain, the header of the last cell in the payload and the ICI of the connection is passed to the application. The application should de-allocate the frame payload chain buffer and the frame header buffer.

Prototype

```
void indTxFrm(USR_CTXT usrCtxt, UINT4 u4ICI, sAPX_CELL_HDR
*psHdr, UINT1 *pu1Frm, INT4 result)
```

Inputs

`usrCtxt`: Pointer to device context information, which the application maintains

`u4ICI`: ICI on which frame was transmitted

`psHdr`: Header of the transmitted frame

`psPyld`: Points to the first buffer in the frame payload buffer chain

`result`:

- 0: Success
- <0: Failure
    - `APX_ERR_SAR_TX_BUSY` (SAR TX is busy)
    - `APX_ERR_SAR_TX_NXT_FRM_BUF` (Error in accessing next buffer in the frame payload buffer chain)

Outputs None

Returns None

Indicating the Success or Failure of Frame Receptions: indRxFrm

The SAR receive task invokes this function after it extracts an AAL5 frame from the microprocessor interface. A pointer to the first byte of the AAL5 frame buffer chain, the header of the last cell in the payload and the ECI of the connection is passed to the application. The application should de-allocate the frame buffer chain and the cell header buffer, except for the case when the `result` returned is `APX_ERR_SAR_RX_FRM_BUF_FULL` (returned when frame buffers not available and frame was discarded)

Prototype

```
void indRxFrm(USR_CTXT usrCtxt, UINT4 u4ECI, sAPX_CELL_HDR
*psHdr, UINT1 *pu1Frm, UINT4 u4Len, INT4 result)
```

Inputs

`usrCtxt`: Pointer to device context information, which the application maintains

`u4ECI`: ICI of the connection on which frame was received

`psHdr`: Header of the last cell in frame

0 if result is `APX_ERR_SAR_RX_FRM_BUF_FULL`

`pu1Frm`: Points to the first buffer in the frame payload buffer chain

0 if result is `APX_ERR_SAR_RX_FRM_BUF_FULL`

`u4Len`: Length of the frame in bytes

0 if result is `APX_ERR_SAR_RX_FRM_BUF_FULL`

`result`:

- 0: Success
- <0: Failure
  - `APX_ERR_SAR_RX_CRC32_FAIL` (CRC32 failure in frame)
  - `APX_ERR_SAR_RX_FRM_BUF_FULL` (Frame buffers are full)
  - `APX_ERR_SAR_RX_FRM_LENGTH` (Mismatch in frame length received by SAR module and frame length in AAL5 trailer)
  - `APX_ERR_SAR_RX_TIMEOUT` (Timeout error while assembling AAL5 frame)

Outputs None

Returns None

Inquiring Whether the Received Cell or Frame is part of Multicasting Group: isVcMulticast

The multicasting support feature in the driver is enabled when this callback function is installed by invoking `apexInstallMulticastFn`. Thereafter this callback function is invoked within the context of the SAR receive task, each time it receives a cell or frame. Based on the connection ID of the incoming cell or frame, the application will then decide whether this cell or frame has to be forwarded to certain outgoing connections. If so, the function will return the number of outgoing connections and the connection ID for each connection. If the application does not want the cell to be forwarded, it should set the number of outgoing connections to 0.

Prototype

```
void isVcMulticast(USR_CTXT usrCtxt, UINT2 u2ICI,
 UINT2 *pu2NumICI, UINT2 **ppu2ICIList)
```

Inputs

`usrCtxt`: Pointer to device context information, which the application maintains

`u4ICI`: ICI on which cell or frame was received

Outputs

`pu2NumICI`: Pointer to a variable specifying the number of outgoing connections on which the application wants the cell/frame to be forwarded. If the application does not want the cell to be forwarded, then the number of outgoing connections should be set to 0.

`ppu2ICIList`: Pointer to an array of connection IDs of the outgoing connections on which the application wants the cell/frame to be forwarded. Note that after multicasting the cell or frame the driver will de-allocate, the memory allocated by the application code for the array of connection id's.

Returns None

Indicating Critical Events: indCritical

`apexHiDPR`, which executes in the context of the DPR task, invokes this function whenever a high priority interrupt count exceed the corresponding threshold value set by the user. The DPR task provides the application with an eventId, which identifies the interrupt counter that crossed the threshold.

Prototype

```
void indCritical(USR_CTXT usrCtxt, UINT4 u4EventId, UINT4
u4Arg1, UINT4 u4Arg2, UINT4 u4Arg3)
```

Inputs `usrCtxt`: Pointer to device context information, which the application maintains

`u4EventId`: ID of the critical event as listed below

APX_EVT_SDRAM_CRC_ERR

APX_EVT_SSRAM_PAR_ERR

APX_EVT_Q_FREE_CNT_ZERO_ERR

APX_EVT_LR_PAR_ERR

APX_EVT_LR_RUNT_CELL_ERR

APX_EVT_LT_CELL_XF_ERR

APX_EVT_WR_PAR_ERR

APX_EVT_WR_RUNT_CELL_ERR

APX_EVT_WR_CELL_XF_ERR

`u4Arg1`: not used

`u4Arg2`: not used

`u4Arg3`: not used

Outputs None

Returns None

Indicating Errors: indError

`apexLoDPR`, which executes in the context of the DPR task, invokes this function whenever a low priority error interrupt count exceed the corresponding threshold value set by the user. The DPR task provides the application with an eventId, which identifies the interrupt counter that crossed the threshold. Based on the event the DPR task will provide additional information relevant to the interrupt, for example the ICI of the connection, which caused the interrupt.

Prototype

```
void indError(USR_CTXT usrCtxt, UINT4 u4EventId, UINT4 u4Arg1,
UINT4 u4Arg2, UINT4 u4Arg3)
```

Inputs

`usrCtxt`: Pointer to device context information, which the application maintains

`u4EventId`: ID of the error event as listed below

`u4Arg1` : value depends on error event as shown below

`u4Arg2` : value depends on error event as shown below

`u4Arg3` : value depends on error event as shown below

Event ID	Arg1	Arg2	Arg3
APX_EVT_Q_VC_REAS_TIME_ERR	ICI	-	-
APX_EVT_Q_VC_REAS_LEN_ERR	ICI	-	-
APX_EVT_Q_CELL_RX_ERR	ICI	-	-
APX_EVT_Q_VC_MAX_THRESH_ERR	ICI	-	-
APX_EVT_Q_CLASS_MAX_THRESH_ERR	portType	portNum	classNum
APX_EVT_Q_PORT_MAX_THRESH_ERR	portType	portNum	-
APX_EVT_Q_DIR_MAX_THRESH_ERR	loopCnt	wanCnt	-
APX_EVT_Q_SHP0_ICTR_ERR	-	-	-
APX_EVT_Q_SHP1_ICTR_ERR	-	-	-
APX_EVT_Q_SHP2_ICTR_ERR	-	-	-
APX_EVT_Q_SHP3_ICTR_ERR	-	-	-

Outputs

None

Returns

None

9 HARDWARE INTERFACE

9.1 Device Input and Output Functions

Reading the Contents of Address Locations: sysApexRawRead

This low-level macro reads the contents of a specific address location. Define this macro to reflect the application's addressing logic.

Prototype `UINT4 sysApexRawRead(UINT4 addr)`

Inputs `addr`: Address location to be read

Outputs None

Returns Value read from the address location

Writing the Contents of Address Locations: sysApexRawWrite

This low-level macro writes the contents of a specific address location. Define this macro to reflect the application's addressing logic.

Prototype `void sysApexRawWrite(UINT4 addr, UINT4 val)`

Inputs `addr`: Address location to write

`val`: Value to be written

Outputs None

Returns None

Detecting New Devices: sysApexDeviceDetect

This function detects the device in the underlying hardware and retrieves system-specific information about the device (such as the base address of device). The function is called within the `apexAdd` API function.

Prototype `INT4 sysApexDeviceDetect(APX_USR_CTXT usrCtxt, void **ppSysInfo, UINT4 *pu4BaseAddr)`

Inputs `usrCtxt`: Pointer to device context information, which the application maintains

Outputs `ppSysInfo`: Application information that you maintain (such as PCI slot and IRQ). The driver stores this pointer.

`pu4BaseAddr`: Base address of device

Returns = 0: Device detected successfully

< 0: Device detection failed

9.2 Interrupt Service Functions

This section describes the functions that the driver needs for interrupt processing. For details on the interrupt service architecture, go to page 26.

ISR Installation and Removal Functions

The following functions install and remove the system-specific interrupt handlers (`sysApexHiIntHandler` and `sysApexLoIntHandler`) and deferred processing routines for the S/UNI-APEX devices.

Installing System-Specific Interrupt Handlers: sysApexIntInstallHandler

This function installs the functions `sysApexHiIntHandler` and `sysApexLoIntHandler`, in the processor's interrupt vector table. It also spawns the DPR and creates the message queue. The ISR routines use the message queue to send interrupt context information to the DPR task.

Prototype `INT4 sysApexIntInstallHandler(sAPX_DDB *psDdb)`

Inputs `psDdb`: Device handle

Outputs None

Returns = 0: Interrupts installed successfully

< 0: Interrupt installation failed

Removing System-Specific Interrupt Handlers: sysApexIntRemoveHandler

This function removes interrupt processing for the device. If the device is the last device for which the driver has enabled interrupt processing, it removes `sysApexHiIntHandler` and `sysApexLoIntHandler` from the processor's interrupt vector table. Then it deletes the `sysApexDPRtask` task and its associated message queue.

Prototype `INT4 sysApexIntRemoveHandler(sAPX_DDB *psDdb)`

Inputs `psDdb`: Device handle

Outputs None

Returns = 0: Interrupts removed successfully

< 0: Interrupt removal failed

System-Specific ISR Functions

The driver invokes the system-specific ISR functions, `sysApexHiIntHandler` and `sysApexLoIntHandler`, when the device(s) raise high priority and low priority interrupts respectively. You should implement these routines as described below:

Handling High-Priority Interrupts: sysApexHiIntHandler

The driver invokes this function when one or more devices raise the high-priority interrupt line to the microprocessor. This function invokes the driver-provided function, `apexHiISR`, for each device registered with the driver.

Prototype `void sysApexHiIntHandler(UINT4 u4IntId)`

Inputs `u4IntId`: System-specific interrupt identifier (such as IRQ)

Outputs None

Returns None

Handling Low-Priority Interrupts: sysApexLoIntHandler

The driver invokes this function when one or more devices raise the low-priority interrupt to the microprocessor. This function invokes `apexLoISR` for each device registered with the driver. If `apexLoISR` detects at least one valid pending interrupt condition, then `sysApexLoIntHandler` queues the interrupt context information (output by `apexLoISR`) for later processing by `sysApexDPRtask` and/or `sysApexSarRxTask` depending on the nature of the interrupt conditions detected.

Prototype `void sysApexLoIntHandler(UINT4 u4IntId)`

Inputs `u4IntId`: System-specific interrupt identifier (such as IRQ)

Outputs None

Returns None

System-Specific DPR Functions

Deferred Interrupt Processing: sysApexDPRtaskFn

The driver spawns this function as a separate task within the RTOS. It retrieves interrupt status information saved for it by the `sysApexLoIntHandler` function and invokes the `apexDPR` routine for the appropriate device.

Prototype `void sysApexDPRtaskFn(void)`

Inputs None

Outputs None

Returns None

10 RTOS INTERFACE

The S/UNI-APEX driver uses the following macros to access RTOS services.

10.1 Memory Allocation and De-allocation Functions

This section describes the functions that allocate and free memory.

Allocating Memory: sysApexMemAlloc

This function allocates the specified number of bytes.

Prototype	`void *sysApexMemAlloc(UINT4 u4Bytes)`
Inputs	`u4Bytes`: Number of bytes to be allocated
Outputs	None
Returns	Pointer to first byte of allocated memory NULL pointer (memory allocation failed)

Freeing Memory: sysApexMemFree

This function frees allocated memory.

Prototype	`void sysApexMemFree(UINT1 *pu1First)`
Inputs	`pu1First`: Pointer to first byte of the memory region being de-allocated
Outputs	None
Returns	None

10.2 Buffer Management Functions

Cell Buffer Functions

Allocating Cell Header Structures and Buffers: sysApexAllocCellBuf

This function allocates a cell header structure and a cell payload buffer.

Prototype `INT4 sysApexAllocCellBuf(sAPX_CELL_HDR **ppsHdr, UINT1 **ppu1Pyld)`

Inputs None

Outputs `ppsHdr`: Contains pointer to allocated cell header buffer

`ppu1Pyld`: Contains pointer to allocated cell payload buffer

Returns = 0: Success

< 0: Failure

Freeing Cell Header Structures and Buffers: sysApexFreeCell

This function returns a cell header structure and payload buffer pair to the free pool.

Prototype `void sysApexFreeCell(sAPX_CELL_HDR *pHdr, UINT1 *pu1Pyld)`

Inputs `psHdr`: Pointer to cell header buffer

`pu1Pyld`: Pointer to cell payload buffer

Outputs None

Returns None

Frame Buffer Functions

Allocating the First Frame Buffer in a Chain: sysApexAllocFrmBuf

This function allocates the first buffer of a frame buffer chain.

Prototype `INT4 sysApexAllocFrmBuf(UINT4 u4Size, sAPX_CELL_HDR **ppsHdr, UINT1 **ppu1Buf)`

Inputs `u4Size`: Size of buffer in bytes

Outputs `ppsHdr`: Contains pointer to allocated cell-header buffer

`ppu1Pyld`: Contains pointer to allocated cell-payload buffer

Returns = 0: Success

< 0: Failure

Adding the Next Frame Buffer to a Chain: sysApexAllocNxtFrmBuf

This function allocates and chains a new buffer to the tail of a frame-buffer-chain. In doing so it provides a pointer to the last buffer of the frame chain.

Prototype `UINT1 *sysApexAllocNxtFrmBuf(UINT4 u4Size, UINT1 *pu1PrevBuf)`

Inputs `u4Size`: Size of buffer in bytes

`pu1PrevBuf`: Pointer to last buffer of current frame chain

Outputs None

Returns Pointer to first data byte in the new frame buffer (chained to the previous buffer)

NULL pointer (buffer unavailable)

Getting a Frame Buffer's Size: sysApexGetFrmBufSz

This function retrieves the size of a frame buffer given a pointer to the first byte of the buffer.

Prototype `UINT4 sysApexGetFrmBufSz(UINT1 *pu1Buf)`

Inputs `pu1Buf`: Pointer to first data byte in buffer

Outputs None

Returns Size in bytes (zero if buffer is invalid)

Getting the Next Frame Buffer's Size: sysApexGetNxtFrmBuf

This function retrieves the pointer to the first byte of the next frame buffer, given the first byte pointer of the previous buffer in the buffer chain.

Prototype `UINT1 *sysApexGetNxtFrmBuf(UINT1 *pu1PrevBuf, UINT4 *pu4Size)`

Inputs `pu1PrevBuf`: Pointer to last buffer of current frame chain

Outputs `pu4Size`: Size of the next buffer in bytes

Returns Pointer to first data byte in the next frame buffer (chained to the previous buffer)

NULL pointer (buffer unavailable)

Freeing Frame Buffers: sysApexFreeFrm

This function frees all frame buffers in the frame buffer chain.

Prototype `void sysApexFreeFrm(UINT1 *pu1FirstBuf)`

Inputs `pu1FirstBuf`: Pointer to the first data byte of the first buffer in the frame buffer chain

Outputs None

Returns None

10.3 Timer Functions

This section describes the timer-related service needed by the driver.

Delaying Tasks: sysApexTaskDelay

This function suspends execution of the calling task for a specified time.

Prototype `INT4 sysApexTaskDelay(UINT4 u4Msecs)`

Inputs `u4Msecs`: Delay length in milliseconds

Outputs None

Returns = 0: Success

< 0: Failure

10.4 Semaphore Functions

This section describes the functions that perform the following semaphore tasks:

- Create semaphores
- Delete semaphores
- Take semaphores
- Release semaphores

Creating Semaphores: sysApexSemCreate

This function creates a mutual-exclusion semaphore.

Prototype `void *sysApexSemCreate(void)`

Inputs None

Outputs None

Returns Pointer to semaphore object or null

Deleting Semaphores: sysApexSemDelete

This function deletes a semaphore.

Prototype `void sysApexSemDelete(void *semId)`

Inputs `semId`: Semaphore identifier

Outputs None

Returns None

Taking Semaphores: sysApexSemTake

This function acquires a semaphore.

Prototype `INT4 sysApexSemTake(void *semId)`

Inputs `semId`: Semaphore identifier

Outputs None

Returns = 0: Success

< 0: Failure

Releasing Semaphores: sysApexSemGive

This function relinquishes a semaphore.

Prototype `INT4 sysApexSemGive(void *semId)`

Inputs `semId`: Semaphore identifier

Outputs None

Returns = 0: Success

< 0: Failure

10.5 Pre-Emption Control Functions

This section describes the functions used to disable and enable pre-emption of the currently executing task.

Disabling Task Pre-emption: sysApexPreemptDis

This function disables possible pre-emption of the currently executing task by other tasks or the interrupt handler.

Prototype `INT4 sysApexPreemptDis(void)`

Inputs None

Outputs None

Returns Pre-emption key (this is passed back as an input argument when re-enabling pre-emption)

Enabling Task Pre-Emption: sysApexPreemptEn

This function enables pre-emption of the currently executing task.

Prototype `void sysApexPreemptEn(INT4 i4Key)`

Inputs `i4Key`: Pre-emption key returned by `sysApexPreemptDis` when disabling preemption for this task

Outputs None

Returns None

10.6 Segmentation and Re-Assembly Assist Functions

This section describes the segmentation and re-assembly (SAR) assist component functions.

Creating SAR Tasks: sysApexSarInstall

This function creates the `apxSarTx` and the `apxSarRx` tasks. The role of `apxSarTx` is to transmit cells and frames to the microprocessor port of the S/UNI-APEX device. The `apxSarRx` receives cells and frames from the microprocessor port of the device. The function also creates the message queues, `SarTxMsgQ` and the `SarRxMsgQ`.

Prototype `INT4 sysApexSarInstall(void)`

Inputs None

Outputs None

Returns = 0: Success

< 0: Failure

Removing SAR Tasks: sysApexSarRemove

This function deletes the `apxSarTx` and `apxSarRx` tasks and the corresponding message queues, `SarTxMsgQ` and `SarRxMsgQ`.

Prototype `INT4 sysApexSarRemove(void)`

Inputs None

Outputs None

Returns = 0: Success

< 0: Failure

SAR Transmit Task Function: sysApexSarTxTaskFn

The driver spawns this function as a separate task within the RTOS. It waits for a cell or frame transmission-request message on the `SarTxMsgQ`. Upon receiving a message, it invokes `apexSarTxTaskFn` for the appropriate device.

Prototype `void sysApexSarTxTaskFn(void)`

Inputs None

Outputs None

Returns None

SAR Receive Task Function: sysApexSarRxTaskFn

The driver spawns this function as a separate task within the RTOS. It retrieves interrupt status information saved for it by the `sysApexLoIntHandler` function and invokes the `apexSarRxTaskFn` function for each device handle received in the message.

Prototype `void sysApexSarRxTask(void)`

Inputs None

Outputs None

Returns None

Sending Transmission Request Messages: sysApexSarTxMsg

This function is invoked by `apexTxCell` and `apexTxFrame` API functions in order to send cell and frame transmission requests to the SAR Tx task. The function puts the cell/frame information into a message structure and queues it in the `SarTxMsgQ`.

Prototype `INT4 sysApexSarTxMsg(sAPX_TX_CTXT sTxMsg)`

Inputs `sTxMsg`: Cell/frame transmission-request information

Outputs None

Returns = 0: Success

< 0: Failure

11 PORTING DRIVERS

This section outlines how to port the S/UNI-APEX device driver to your hardware and OS platform. However, this manual can offer only guidelines for porting the S/UNI-APEX driver because each platform and application is unique.

11.1 Driver Source Files

The C source files listed in Table 21 and Table 22 contain the code for the S/UNI-APEX driver. You may need to modify the code or develop additional code. The code is in the form of constants, macros, and functions. For ease of porting, the code is grouped into source files (`src`) and include files (`inc`). The `src` files contain the functions and the `inc` files contain the constants and macros. A makefile is also included.

Table 21: Source Files

File	Description
apx_api1.c	Top-level API functions
apx_api2.c	Low-level utility API functions
apx_hw.c	Hardware interface functions
apx_rtos.c	RTOS interface functions
apx_io.c	Input/Output functions
apx_ism.c	Interrupt control functions
apx_qe.c	Queue engine operations
apx_sar.c	SAR-assist operations
apx_stat.c	Statistics functions
apx_util.c	Commonly used utility functions
apx_lps.c	Loop port and WAN port scheduler functions
apx_prof.c	Profile management routines
apx_eg.c	Example implementation of callback and other functions

Table 22: Include Files

File	Description
apx_api.h	API function prototypes, data structures, constants, and definitions

File	Description
apx_typs.h	Variable type definitions
apx_hw.h	Hardware interface constants and macro definitions
apx_rtos.h	RTOS interface constants and macro definitions
apx_err.h	Error codes returned by the driver
apx_defs.h	Driver's internal constants and macro definitions
apx_strs.h	Driver's internal data structures
apx_fns.h	Prototypes of driver's internal functions
apx_eg.h	Data structures, constants, and definitions used by sample code in apx_eg.c

11.2 Porting Procedure

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

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

Step 1: Port the driver's hardware interface (page 130):

Step 2: Port the driver's OS extensions (page 132):

Step 3: Port the driver's application-specific elements (page 134):

Step 4: Build the driver (page 135).

Step 1: Porting the Hardware Interface

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

To port the driver to your hardware platform:

1. Modify the variable type definitions in `apx_typs.h`.
2. Modify the low-level device read/write macros in the `apx_hw.h file`. You may need to modify the raw read/write access macros (`sysApexRawRead` and `sysApexRawWrite`) to reflect the application's addressing logic.

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

Device Constant	Description	Default
`APX_MAX_DEVS`	The maximum number of S/UNI-APEX devices to be controlled by the driver	`2`
`APX_MAX_CELL_BUFS`	The greatest of the per-device cell buffer requirements	`256K`
`APX_MAX_NUM_VCS`	The greatest of the per-device VC requirements	`64K`
`APX_POLL_DELAY`	Delay between two consecutive polls of a busy bit	`5uS`
`APX_MAX_POLL_TRIES`	Maximum number of times a busy bit will be polled before the operation times out	`100`
`APX_PORT_DISABLE_DELAY_MSECS`	The number of milliseconds the driver waits during the port disable operation for the `PortCnt` parameter to become 0	`1`
`APX_CLASS_DISABLE_DELAY_MSECS`	The number of milliseconds the driver waits during the class disable operation (shaped classes only) for the associated shaper slot table to deplete itself completely into the class queue	`1`
`APX_CONN_DISABLE_DELAY_MSECS`	The number of milliseconds the driver waits during the connection disable operation for the `VcClassQClp01Cnt` parameter to become 0	`1`
`APX_SDRAM_REFRESH_RT`	Default SDRAM refresh rate used for SDRAM tests	`0xf`

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

5. Modify the sysApexBusyBitPoll function if necessary. This function polls a specified busy bit APX_MAX_POLL_TRIES with a APX_POLL_DELAY polling interval. If the bit does not reach its desired value, the function returns with an error code of –1.

6. (OPTIONAL) Modify the sysApexDebugRead, sysApexDebugWrite and sysApexTrace functions. Porting these functions is only required if you want to use the debug message printing feature of register accesses and error messages (enabled by compile switch, APX_CSW_DEBUG).

Step 2: Porting the RTOS interface

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

To port the driver's RTOS interface:

1. Redefine the following macros in apx_rtos.h to the corresponding system calls that your target system supports. See apx_eg.c for example implementations of the buffer management routines

Service Type	Macro Name	Description
Memory	sysApexMemAlloc	Allocates a memory block
	sysApexMemFree	Frees a memory block
	sysApexMemSet	Fills a memory block with a specified value
	sysApexMemCpy	Copies the contents of one memory block to another
	sysApexMemCmp	Compares the contents of one memory block with another
Buffer Management	sysApexAllocCellBuf	Allocates a cell buffer
	sysApexFreeCell	Frees a cell buffer
	sysApexAllocFrmBuf	Allocates a frame buffer
	sysApexAllocNxtFrmBuf	Allocates and chains a new frame buffer to the previous buffer of the specified frame
	sysApexGetFrmBufSz	Obtains the length of the payload in a frame buffer
	sysApexGetNxtFrmBuf	Retrieves the frame buffer (and its size) immediately following the specified frame buffer

Service Type	Macro Name	Description
	`sysApexFreeFrm`	Frees the entire chain of frame buffers that comprise the specified frame
Semaphores	`sysApexSemCreate`	Creates a mutual-exclusion semaphore
	`sysApexSemDelete`	Destroys the specified semaphore
	`sysApexSemTake`	Acquires the specified semaphore
	`sysApexSemGive`	Relinquishes the specified semaphore

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

Service Type	Function Name	Description
Interrupt Service/Polling	`sysApexIntInstallHandler`	Installs the interrupt handler for the OS
	`sysApexIntRemoveHandler`	Removes the interrupt handler from the OS
	`sysApexHiIntHandler`	Interrupt handler for the high-priority S/UNI-APEX interrupt line
	`sysApexLoIntHandler`	Interrupt handler for the low-priority S/UNI-APEX interrupt line
	`sysApexDPRTaskFn`	Deferred processing routine that waits for interrupt context information to be sent by the ISR routines and then processes the interrupt status information
SAR Processing	`sysApexSarInstall`	Spawns the SAR Rx and Tx tasks and associated message queues
	`sysApexSarRemove`	Deletes the SAR Rx and Tx tasks and associated message queues

Service Type	Function Name	Description
	`sysApexSarTxTaskFn`	This function is executed in the context of the SAR Tx task. It receives cell and frame transmission requests from the application task and invokes the appropriate cell/frame transmission API function.
	`sysApexSarRxTaskFn`	This function is executed in the context of the SAR Rx task. It extracts cells and frames from the S/UNI-APEX SAR interface and sends them to the application task using the `indRxCell`/`indRxFrm` callback functions
	`sysApexSarTxMsg`	This routine is used by the application task to send cell/frame transmission requests to the SAR Tx task
Timer	`sysApexTaskDelay`	Puts the currently executing task to sleep for a specified number of milliseconds
Pre-emption Lock/Unlock	`sysApexPreemptDis`	Disables pr-eemption of the currently executing task by any other task or interrupt
	`sysApexPreemptEn`	Re-enables pre-emption of a task by other tasks and/or interrupts

Step 3: Porting the Application-Specific Elements

Porting the application-specific elements includes coding the indication callback functions and defining the base value from which the S/UNI-APEX driver's error codes start.

To port the driver's system-specific elements:

1. Modify the base value of `APX_ERR_BASE` (default = `300`) in `apx_err.h`.
2. Code the callback functions according to the application. Example implementations of these callback functions are provided in `apx_eg.c`. The callback functions are the following:

- `void indCritical(APX_USR_CTXT usrCtxt, UINT4 u4EventId, UINT4 u4Arg1, UINT4 u4Arg2, UINT4 u4Arg3)`
- `void indError(APX_USR_CTXT usrCtxt, UINT4 u4EventId, UINT4 u4Arg1, UINT4 u4Arg2, UINT4 u4Arg3)`
- `void indTxCell(APX_USR_CTXT usrCtxt, UINT4 u4ICI, sAPX_CELL_HDR *psHdr, UINT1 *pu1Pyld, INT4 i4Result)`
- `void indRxCell(APX_USR_CTXT usrCtxt, UINT4 u4ECI, sAPX_CELL_HDR *psHdr, UINT1 *pu1Pyld, INT4 i4Result)`
- `void indTxFrm(APX_USR_CTXT usrCtxt, UINT4 u4ICI, sAPX_CELL_HDR *psHdr, UINT1 *pu1Frm, INT4 i4Result)`
- `void indRxFrm(APX_USR_CTXT usrCtxt, UINT4 u4ECI, sAPX_CELL_HDR *psHdr, UINT1 *pu1Frm, UINT4 u4Length, INT4 i4Result)`
- `void isVcMulticast(APX_USR_CTXT usrCtxt, UINT2 u2ICI, UINT2 *pu2NumICI, UINT2 **ppu2ICIList);`

Step 4: Building the Driver

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

To build the driver:

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

APPENDIX A: DRIVER RETURN CODES

Table 23 describes the driver's return types.

Table 23: Return Types

Return Type	Description
APX_ERR_CLASS_NOT_ENABLED	Associated class not enabled
APX_ERR_CLASS_NOT_FREE	Class already configured
APX_ERR_DEV_ALREADY_ADDED	Device already added
APX_ERR_DEV_NOT_DETECTED	Device was not detected
APX_ERR_DEVS_FULL	Maximum number of devices already added
APX_ERR_INVALID_CLASS_ID	Invalid class ID
APX_ERR_INVALID_CLASS_VECTOR	Invalid class vector
APX_ERR_INVALID_CONN_ID	Invalid connection ID
APX_ERR_INVALID_CONN_VECTOR	Invalid connection vector
APX_ERR_INVALID_CTRL_PARAM	Invalid control parameter
APX_ERR_INVALID_DEV	Invalid device handle
APX_ERR_INVALID_FLAG	Invalid value for u1enflg
APX_ERR_INVALID_INIT_VECTOR	Invalid initialization vector
APX_ERR_INVALID_INT_TYPE	Invalid interrupt type
APX_ERR_INVALID_MSK_ID	Invalid mask register
APX_ERR_INVALID_PORT_ID	Invalid port ID
APX_ERR_INVALID_PORT_VECTOR	Invalid port vector
APX_ERR_INVALID_PROFILE_NUM	Invalid profile number
APX_ERR_INVALID_SHPR_ID	Invalid shaper number
APX_ERR_INVALID_SHPR_VECTOR	Invalid shaper vector
APX_ERR_INVALID_STATE	Invalid device state
APX_ERR_INVALID_TEST_PARAM	Invalid test parameter
APX_ERR_LPS_INVALID_WT	Invalid contents in the loop port weight table
APX_ERR_WPS_INVALID_WT	Invalid contents in the WAN port weight table
APX_ERR_MEM_ALLOC	Memory allocation failure
APX_ERR_MODULE_NOT_INIT	Driver has not been initialized

Return Type	Description
APX_ERR_POLL_TIMEOUT	Memory port access failed
APX_ERR_PORT_NOT_CFG	Port not configured
APX_ERR_PORT_NOT_ENABLED	Port is not enabled
APX_ERR_PORT_NOT_FREE	Port already configured
APX_ERR_PROFILES_FULL	All initialization profiles are in use
APX_ERR_PROFILE_VECTOR_BOTH_VALID	Both vector profile number are valid
APX_ERR_SAR_RX_CRC10_FAIL	CRC-10 check failed
APX_ERR_SAR_TX_BUSY	SAR transmit component is busy
APX_ERR_SAR_TX_MSG	Error in sending message to SAR transmit message queue
APX_ERR_SAR_TX_TYPE	Error in value of `txtype`
APX_ERR_SHPR_NOT_FREE	Associated port-class still configured
APX_FAILURE	Test failed
APX_SUCCESS	The function succeeded
APX_ERR_DLL_PHASE_LOCK	DLL phase lock failure
APX_ERR_SEMAPHORE	Semaphore allocation error
APX_ERR_INVALID_EVENT_ID	Invalid event ID
APX_ERR_MODULE_ALREADY_INIT	Driver already initialized
APX_ERR_INVALID_TYPE_ID	Device detected has invalid TYPE/ID
APX_ERR_INT_INSTALL	Error installing interrupts
APX_ERR_INT_REMOVE	Error removing interrupts
APX_ERR_INVALID_MODE	Invalid mode parameter specified
APX_ERR_INVALID_REG	Invalid register offset
APX_ERR_INVALID_ADDR	Invalid address
APX_ERR_INVALID_MSKDATA	Invalid mask data
APX_ERR_INVALID_MP_CTRL	Invalid memory port control parameter(s)
APX_ERR_INVALID_CELL_START	Invalid cell start address
APX_ERR_INVALID_CELL_NUM	Invalid number of cells
APX_ERR_INVALID_CTXT	Invalid context type
APX_ERR_INVALID_WORD	Invalid context word
APX_ERR_INVALID_NUM_CELL_BUFS	Invalid number of cell buffers

Return Type	Description
APX_ERR_WDG_PTRL_BUSY	Watchdog patrol already active
APX_ERR_PORT_CTXT_CHK	Port context image mismatch
APX_ERR_CLASS_NOT_CFG	Class not configured
APX_ERR_CLASS_CTXT_CHK	Class context image mismatch
APX_ERR_INVALID_TX_SLOT	Invalid shaper txslot
APX_ERR_SHPR_NOT_CFG	Shaper not configured
APX_ERR_INVALID_ICI	Invalid ICI
APX_ERR_CONN_NOT_CFG	Connection not configured
APX_ERR_CONN_NOT_FREE	Connection not free
APX_ERR_INVALID_RANGE	Invalid ICI watchdog patrol range
APX_ERR_CONN_CTXT_CHK	Connection context image mismatch
APX_ERR_SAR_INSTALL	SAR assist module installation error
APX_ERR_SAR_REMOVE	SAR assist module removal error
APX_ERR_SAR_TX_NXT_FRM_BUF	Error getting SAR transmit frame buffer
APX_ERR_SAR_TX_FRM_LENGTH	Invalid frame length
APX_ERR_SAR_RX_CELL_BUF_FULL	SAR receive cell buffer is full
APX_ERR_SAR_RX_CRC32_FAIL	SAR receive CRC32 check failure
APX_ERR_SAR_RX_TIMEOUT	SAR receive timeout
APX_ERR_SAR_RX_FRM_BUF_FULL	SAR receive frame buffer is full
APX_ERR_SAR_RX_FRM_LENGTH	Error in receive frame length
APX_ERR_LPS_INVALID_SEQ	Invalid loop port sequence number
APX_ERR_INVALID_NUM_PORTS	Invalid number of ports
APX_ERR_INVALID_DIR_THRSH	Invalid direction threshold parameter
APX_ERR_INVALID_DIR	Invalid direction
APX_ERR_INVALID_PORT_THRSH	Invalid port threshold parameter
APX_ERR_INVALID_CL_SCHD	Invalid class scheduling parameter
APX_ERR_INVALID_CLASS_THRSH	Invalid class threshold parameter
APX_ERR_INVALID_SHP_PARAM	Invalid shaping parameter
APX_ERR_INVALID_CONN_THRSH	Invalid connection threshold
APX_ERR_INVALID_WFQ_WT	Invalid weight for WFQ connection

Return Type	Description
APX_ERR_INVALID_CONN_TYPE	Invalid connection type

APPENDIX B: CODING CONVENTIONS

This section describes the coding and naming conventions used to implement the driver software. This section also describes the variable types.

Variable Types

This section describes the variable types used by the driver code.

Table 24: Variable Type Definitions

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

Naming Conventions

This section describes the naming conventions for the following items in the driver code:

- Macros
- Constants
- Structures
- Functions
- Variables

Table 25: Naming Conventions: Macros, Constants, and Structures

Type	Example	Case	Prefix	Notes
Macro	mAPX_WRITE	Upper	Lowercase "m" followed by abbreviated, uppercase device name: mAPX	Separate words with an underscore "_".
Constant	APX_REG		Abbreviated, uppercase device name: APX	
Structure	sAPX_DDB		Lowercase "s" followed by abbreviated, uppercase device name: sAPX	

Table 26: Naming Conventions: Functions and Variables

Type	Example	Case	Prefix	Notes
API Function	`apexAdd()`	Title case, but the first letter is always lowercase	Full, lowercase device name: `apex`	Follow hungarian notation. Do not separate words.
Porting Function	`sysApexRawRead()`		Lowercase “`sys`” followed by full, title case device name: `sysApex`	Porting functions are all functions that are platform dependent.
Static Function	`qeIsConnShaped`			Static functions are internal functions and have no special naming conventions other than hungarian notation.
Global Variable	`apexGdd`		Full, lowercase device name: `apex`	
Standard Variable	`u1Type, u2Num, u4Data, ret`		Optionally indicate variable type using “u1, u2, u4 etc”	No special naming conventions used
Pointer to Variable	`pu4Data, psDdb, pcb, ppTable`		Prefix single pointers with lowercase “`p`” followed by the unchanged variable name. Optionally, you can prefix double pointers with lowercase “`pp`” followed by the unchanged variable name.	

INDEX

L

M

N

O

P

Q

R

S

V

W

Z