PM73488

QSE DRIVER

5 GBIT/S ATM SWITCH FABRIC ELEMENT DRIVER

USER'S MANUAL

Preliminary
Issue 1: November 1998

Public Revision History

Issue Number	Issue Date	Details of Change
Issue 1	November 1998	Document created.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1
About this Manual

SCOPE

This manual describes the software driver for the PMC-Sierra 5 Gbit/s ATM Switch Fabric Element device (the QSE™ Driver).

REFERENCES

This manual references the following documents:

- ANSI/ISO 9899-1990, *C Programming Language* standard.
- ATM Forum, *ATM User-Network Interface Specification*, V3.1, September 10, 1993.
- PMC-Sierra, *PM73488 QSE Long Form Data Sheet (Document Number PMC-980616)*.

WHAT SHOULD YOU READ?

For the latest information about QSE Driver issues...

Refer to the latest RELNOTES.TXT file included with the source code.

If you are an application programmer or system programmer...

Read Chapter 3, "QSE Driver Architecture", for a quick overview of how the driver operates. Then read Chapter 5, "QSE Driver API", to learn the QSE Driver API functions.

If you will be porting the QSE Driver...

Read Chapter 4, "Porting Guidelines", for tips on compiling the QSE Driver on your host system, and bringing it up on your target system.

TYPOGRAPHICAL CONVENTIONS USED IN THIS MANUAL

Different fonts are used in this manual to help you understand what is explained. Table 1 describes how these different fonts are used.

Table 1. Conventions Used in this Manual

Font	Explanation	Example
Italic	Emphasis	The driver is fully tested and debugged so your initial testing will not involve *both* untested hardware and untested software.
`Courier`	Function names or Sample code	Code the `QseSigFn()` function. `#define MEM_FREE(`*a*`)` *xxx*

DEFINITIONS OF ACRONYMS USED IN THIS MANUAL

Table 2 lists the acronyms used in this manual and their definitions.

Table 2. Acronym Definitions

Acronym	Definition
API	Application Programming Interface
DCB	Device Control Block
DCT	Device Control Task
DDB	Device Data Block
DDG	Driver Diagnostic Function
DDL	Device Driver Library
DDO	Device Driver Operation
DRR	Driver Restart and Reinitialization Function
IQRT	Input QRT
GDDB	Global Device Driver Database
ISR	Interrupt Service Routine
MGVT	Multicast Group Vector Table
MPV	Multicast Port Vector
MSB	Most Significant Bit
NMS	Network Management System
OQRT	Output QRT
OS	Operating System
POST	Power-On Self Test
QRT	PM73487 622 Mbit/s ATM Traffic Management Device
QSE	PM73488 5 Gbit/s ATM Switch Fabric Element
RAM	Random Access Memory
RTOS	Real-Time Operating System
SNMP	Simple Network Management Protocol
SRAM	Static Random Access Memory

Chapter 2
QSE Driver Overview

ABOUT THIS CHAPTER

This chapter summarizes the main features of the QSE Driver.

OVERVIEW OF SOFTWARE FEATURES

The QSE Driver is a high-level interface for the QSE device that augments and supports the functions of the PM73488 QSE device. The driver hides the hardware-specific details from the higher layer software so the higher layer software views the device as a switch with ports and associated control parameters. The QSE Driver provides the functions described in the following sections.

Supporting Multiple QSE Devices

Since there can be multiple QSE devices in a switch fabric and the QSE Driver can handle multiple QSE devices, only one instance of the QSE Driver is necessary in the system. The application indicates the specific QSE device in the driver calls. Supporting multiple QSEs through one instance of the QSE Driver reduces the driver memory requirements, since the same driver data structures are shared among the QSEs, and also provides a better maintenance of the entire switch fabric. An example of a shared structure is the Global Device Driver Database (GDDB). Also, the QSE Driver code is not duplicated, thus saving system memory. Because there is only one task for all QSEs, the system runs faster and many duplicate functions (such as periodic timer event processing) are combined.

Initializing the QSE Devices

The QSE Driver performs device initialization in three steps.

1. The QSE Driver resets the device and initializes the device with a default initialization vector.
2. The QSE Driver then performs various Power-On Self Tests (POSTs) to test the device operation, the external device memories, and the processor interface.
3. Finally, the QSE Driver configures the device control registers and external RAM based on a user-configured initialization vector.

The QSE Driver and the QSE devices are ready to perform their normal functions after initialization.

Setting Up and Clearing Unicast and Multicast Port Groupings

The primary function of the PM73488 QSE device is to switch the cells received from the PM73487 622 Mbit/s ATM Traffic Management Device (the QRT™) in the input port and route them to the output ports in the switch fabric using the tags in the cell headers. Before data transfer can begin, the ports must be grouped or aggregated together in 1, 2, 4, 8, 16, or 32 sets for unicast configuration and 1, 2, or 4 sets for multicast configuration. Traffic through the switch fabric can be of the following types:

- Point-to-point traffic (unicast)
- Point-to-multipoint traffic (multicast)

Point-to-Point Traffic

Point-to-point traffic (also called unicast traffic) cells get through the switch in one-cell time. The path the cell has to take through the switch is contained in the cell header (called a tag). The tag indicates through which ports the cell

has to be sent on this switch. To have fault tolerance, a number of ports can be grouped together (through the QSE Driver) to form a single output. Thus, if one port fails, cells can be routed through any of the other valid ports in the group. Since the path a cell has to take is indicated in the cell header, there is no need to store any traffic information in tables, as is done in multicast traffic.

Point-to-Multipoint Traffic

Point-to-multipoint traffic (also called multicast traffic) setup is more complex than unicast traffic setup. Multicast cells contain an index in the cell header for the Multicast Group Vector Table (MGVT) in the SRAM. The MGVT can be configured using only the registers in the QSE device(s) through the QSE Driver. For a cell to be duplicated onto multiple ports, the Multicast Port Vector (MPV) for the index should be set to indicate the ports onto which the cells have to be duplicated. The ports can be dynamically added, deleted, or modified using the APIs provided. The multicast dequeue engine in the switch gets the MPV for the index from the SRAM and duplicates the cells onto the ports active in the MPV vector.

Figure 1 shows the flow of a cell through the switch for multicast traffic.

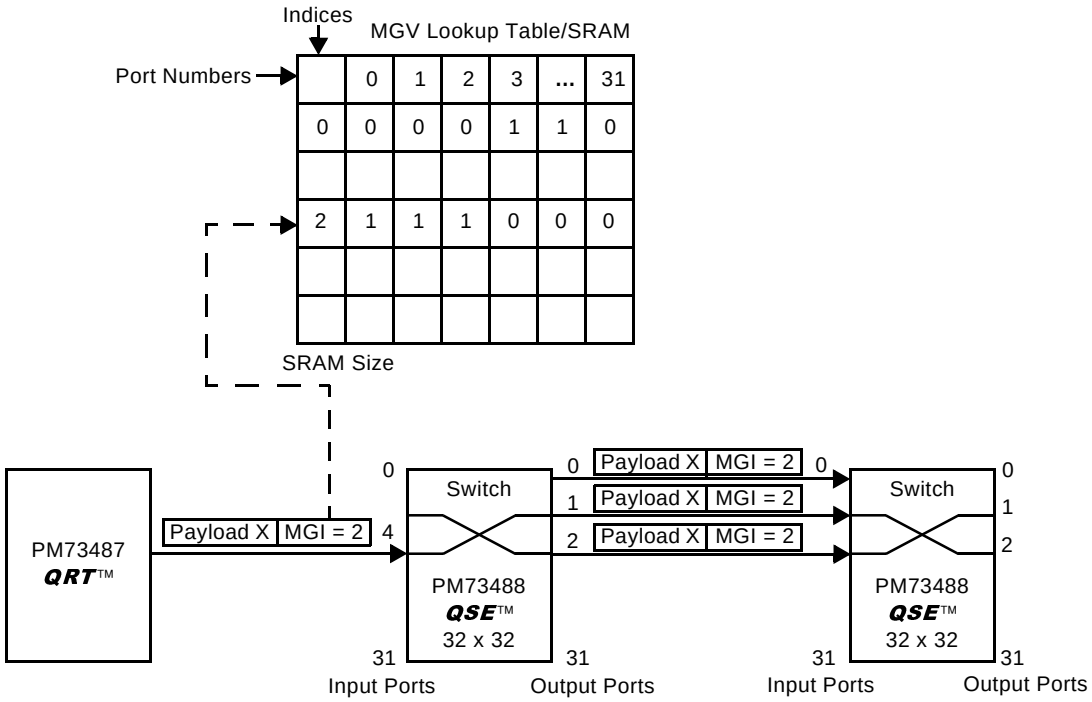

Figure 1. Multicast Cell Replication Using the MGV Lookup Table

After configuring the switch, the higher layer software sets up the MGV table in the RAM for multicast traffic. It is more efficient to write into SRAM while in software reset mode. To clear the MGV vectors, overwrite the content of the index with zeros.

Configuring Acknowledgment Payloads and Buffer Thresholds

The acknowledgment payloads are relayed back to inform the sender QRT or QSE of the state of the information received in the next or the previous cell time. The default acknowledgment payloads are: ONACK, MNACK, and ACK. The acknowledgment is a 4-bit payload that can be configured to different values, depending on the requirements of the systems using the QSE Driver. Also through the QSE Driver, thresholds can be configured for the 64 buffers available for multicast traffic.

Collecting Status, Statistics, and Diagnostic Information

The QSE status, statistics, and diagnostic information can be used by the higher layer for network management purposes. The driver accumulates all error counts, such as number of PARITY_FAILs, BP_ACK_FAILs, and marked cell counts. The status API functions provide a snapshot of the status of the device, such as the interrupts that occurred.

Providing QRT and QSE Inter-Connectivity

Unicast cells are independent of the QRT/QSE inter-connection, since the cells themselves contain the path they have to take through the switch fabric. However, multicast cells contain the index of the vector that has to be fetched from the RAM. Therefore the indexes have to be properly set in the QRT so they point to the correct vector in the SRAM in the QSE fabric. The higher layer software (above the QSE/QRT Drivers) should take care of the consistency of these indexes in both the QRT and the QSE devices. BP/ACKs are sent between the QRT and QSE devices as determined by the various thresholds set in the devices. Figure 2 shows the data/ack flow between the QSE and QRT devices.

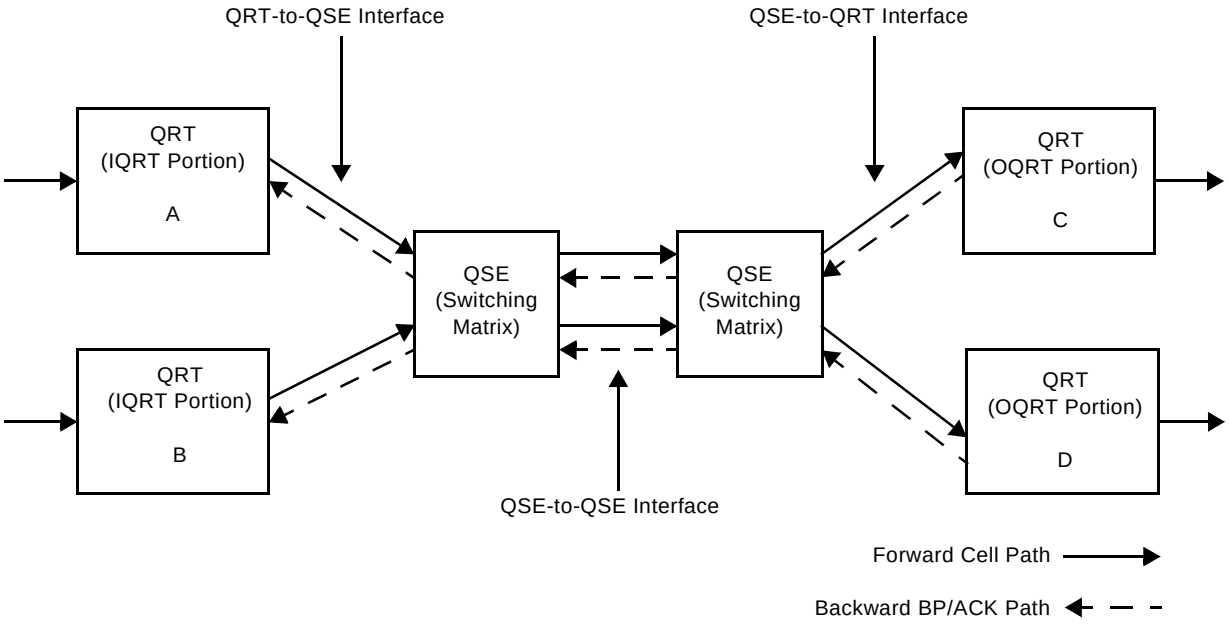

Figure 2. Basic Data Path Inter-Connectivity between the QSE Fabric and QRT Devices

HOW THE QSE DRIVER INTERACTS WITH APPLICATIONS

The QSE Driver interacts with the application layer in the following ways:

- Direct function calls (API functions) provided by the QSE Driver to the application.
- Callback functions provided by the application to the QSE Driver DCT.
- Event notifications by the DCT to the application.

Direct Function Calls

The direct function calls are the API functions that are part of the DDL and are available to the application for a variety of configurations and managements. For example, the application uses direct function calls, such as `QseAddMCGrp` (refer to "QseAddMCGrp" on page 37) and `QseClearMCGrp` (refer to "QseClearMCGrp" on page 37) for setting up and tearing down multicast groups associated with the index.

Callback Functions

Callback functions are provided by the application to the QSE Driver. When the DCT receives a signal from the ISR about the occurrence of a significant event, such as a port failure or parity error, it calls a function provided by the application as a callback routine to inform the application of this routine. Within the callback routine, the application can take any corrective or logging action the corresponding alarm requires.

Asynchronous Event Notification

Instead of using callback functions to notify the application layer of significant events, the DCT can also signal the application. This signaling can occur in the form of an event notification, sending a message in a queue, or any other mechanism the user wants to use. The signaling function used by the QSE Driver, `QseSigFn` (refer to "Step 7: Modify the qseport.c File" on page 25), is a porting function and can be modified by the user to fit the system messaging scheme.

QSE DRIVER STATES

The QSE Driver functions as a simple state machine. The QSE Driver can be in one of four possible states. The QSE Driver is assumed to be in the S0 state at power-up.

The states function as checkpoints to verify the proper initializations have occurred before calling the API functions, and to ensure the API functions are not called in the wrong order. Before the driver can enter the next state, a function must verify the driver is in the correct state. The function must then complete successfully so the driver can transition to the next state. The QSE Driver states are as follows:

- S0: Power-On Initialization State. The `QsePowerOnInit` function checks for the S0 state. If the S0 state exists and the function is successful, the state is changed to S1.
- S1: Power-On Self Test State. The `QsePowerOnSelfTest` function checks for the S1 state. If the S1 state exists and the function is successful, the state is changed to S2.
- S2: Final System Initialization State. The `QseFinalInit` function checks for the S2 state. If the S2 state exists and the function is successful, the state is changed to S3, which is the required state for most API functions.
- S3: Operational State.

Under normal conditions, the transition sequence between the states is as follows: S0 followed by S1, followed by S2 (after permission from the management entity), followed by S3 (again, after permission from the management entity). The QSE Driver is usually in the S3 state. Most API functions, such as transmit and receive operations on the device, require the QSE Driver to be in the S3 operational state.

As Figure 3 shows, the initial software of a system should bring the QSE Driver to state S3 by making the following sequence of calls:

- `QsePowerOnInit()`;
- `QsePowerOnSelfTest()`; and
- `QseFinalInit()`.

This sequence of calls is equivalent to `open()` in a UNIX/DOS environment. The QSE Driver may exit this S3 state and enter any of the preceding states in case of abnormal conditions (such as system hang-up and restart/reset) or during a system reconfiguration phase. In any function, if the required state does not exist, or if the function does not complete successfully, the state remains the same and an error message is generated.

Figure 3 shows the states. The QSE is assumed to be in the S0 state at power-up.

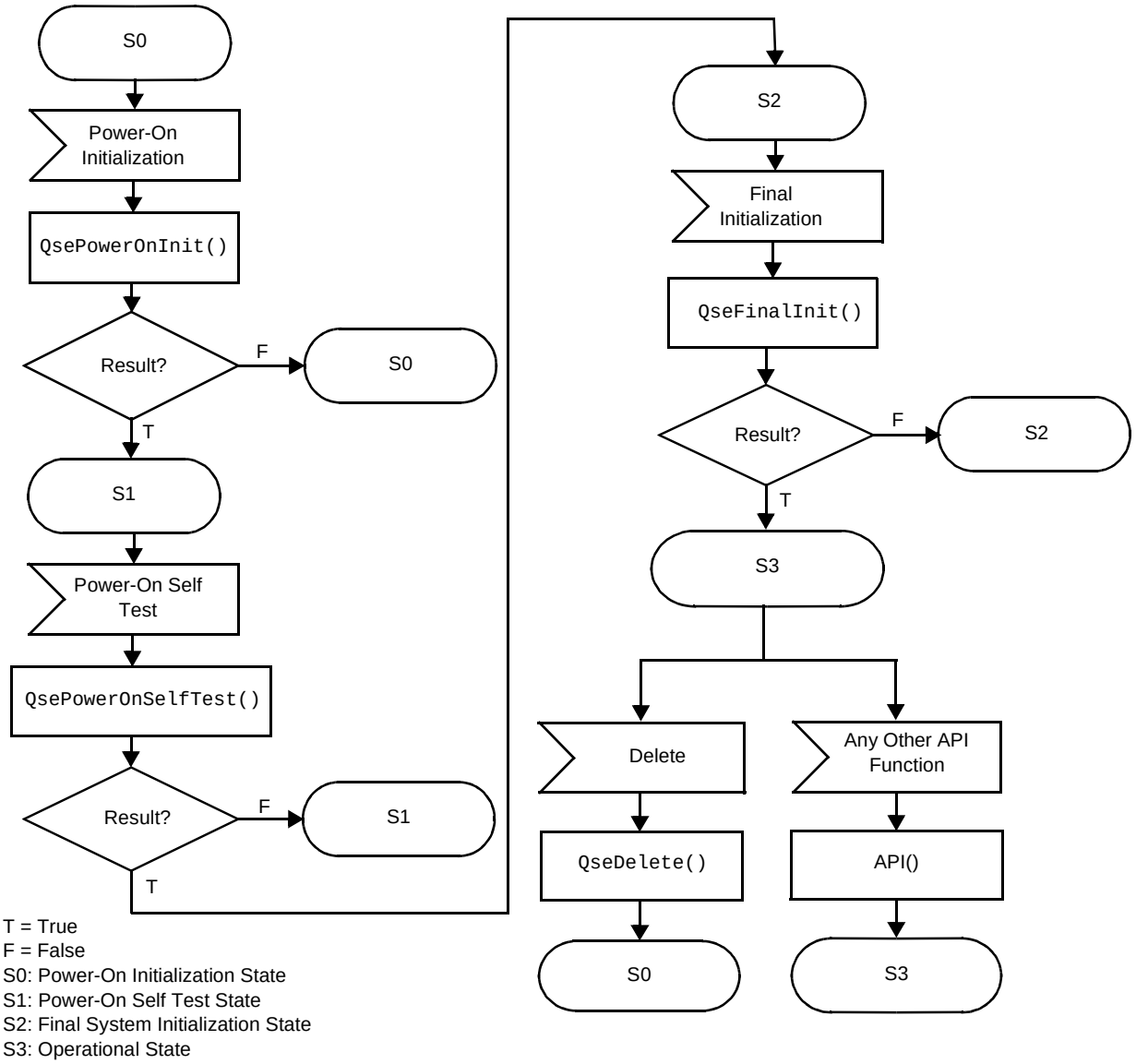

Figure 3. State Transition Diagram of the QSE Driver

Events of interest are:

- Power-up
- Operations the API function calls schedule for the device
- Device interrupts
- Timer events that notify timeout conditions

State S0 (Power-On Initialization State)

S0 is entered from the `QseDriverInit` function. The QSE Driver (in S0) then proceeds to the `QsePowerOnInit` function, which checks for the S0 state. If the S0 state exists and the `QsePowerOnInit` function is successful, the state is changed to S1. During the S0 state no RTOS functions are available.

State S1 (Power-On Self Test State)

S1 is entered from the `QsePowerOnInit` function. The QSE Driver (in S1) then proceeds to the `QsePowerOnSelfTest` function, which checks for the S1 state. If the S1 state exists and the `QsePowerOnSelfTest` function is successful, the state is changed to S2.

State S2 (Final System Initialization State)

The QSE Driver (in S2) proceeds to the `QseFinalInit` function, which checks for the S2 state. If the S2 state exists and the `QseFinalInit` function is successful, the state is changed to S3.

State S3 (Operational State)

In the S3 state, the QSE Driver APIs are available to the application. S3 is exited when the device is deleted.

RE-ENTRANCY ISSUES

In a multitasking/multithreaded environment, such as in a RTOS, the QSE Driver calls may be accidentally re-entered. The QSE Driver does not check to prevent such re-entries. In a re-entrance, the QSE Driver's data structures remain intact; however, there still may be unpredictable consequences for the application. To avoid such "race" conditions, design the application so the QSE Driver API cannot be re-entered.

SYNCHRONIZATION ISSUES

In a multithreaded environment there may be places where critical regions have to be synchronized. This is accomplished by using the preemption operations provided by the OS extensions.

Chapter 3
QSE Driver Architecture

ABOUT THIS CHAPTER

This chapter describes the internal QSE Driver architecture, the purpose of the main QSE Driver functional blocks, and the QSE Driver states.

QSE DRIVER DESIGN

The QSE Driver provides a high-level interface to the PM73488 QSE devices in the switch fabric. The QSE Driver hides hardware-dependent information to provide easier and more efficient access to the QSE device. Figure 4 shows the detailed architecture of the QSE Driver, the various modules, and the main data structures. It also shows the interaction between the QSE Driver modules and their accesses to the data structures.

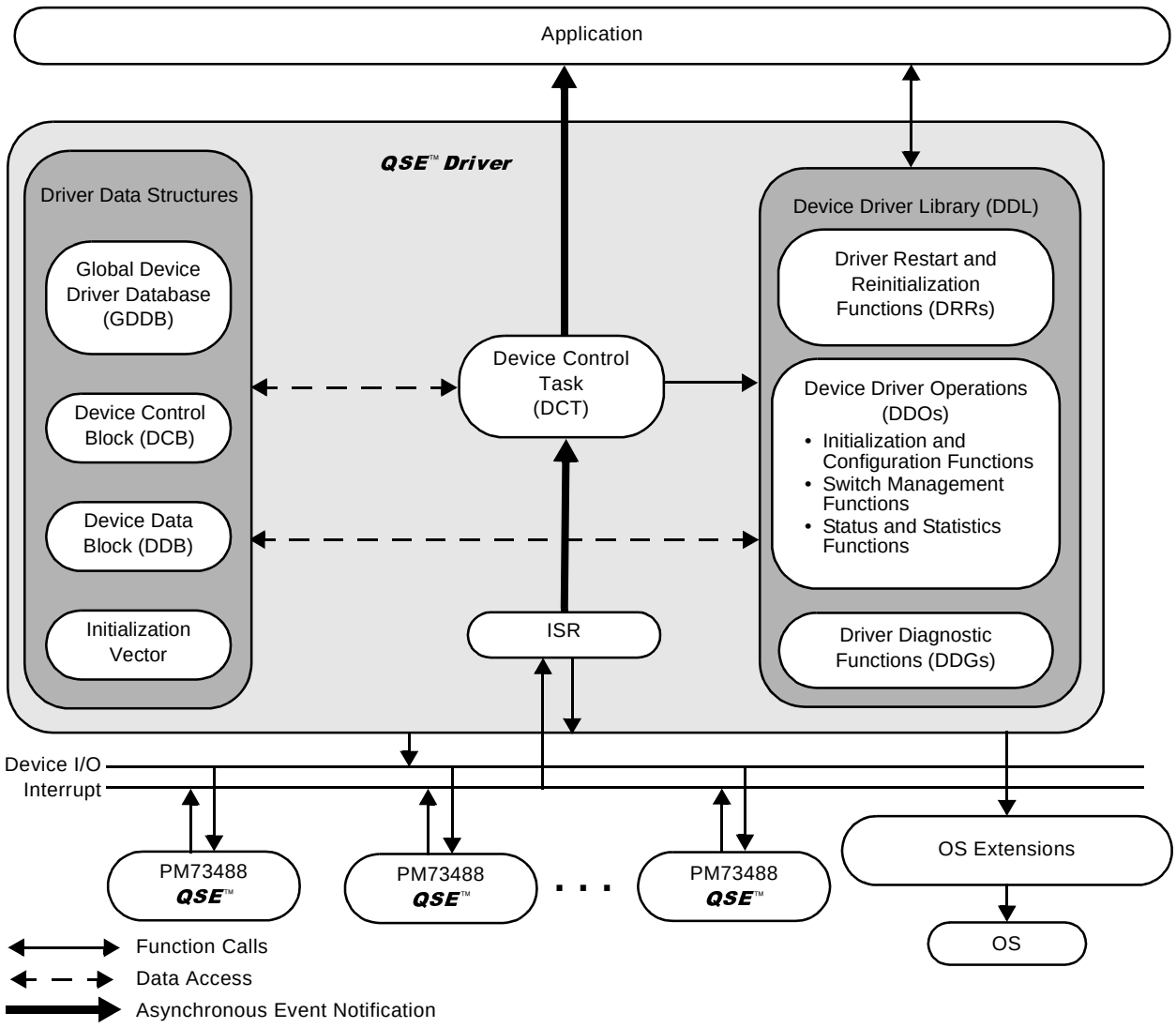

Figure 4. QSE Driver Architecture

The QSE Driver is divided into modules that perform different functions. These modules communicate with each other and access the QSE Driver data structure, as well as the QSE device's internal data structures, to service the API functions called by the application.

The QSE Driver is divided into the following major functional blocks:

- The Device Driver Library (DDL)
- The Device Control Task (DCT)
- The Interrupt Service Routine (ISR)
- The driver data structures

The OS Extensions module is a software unit that is also provided with the QSE Driver. For more information, refer to "OS Extensions" on page 19.

For any number of QSE devices controlled by the QSE Driver, there is only one instance of the DDL and the DCT. Since a single QSE device can have up to two switches, the switches in the QSE devices are identified using an incremental number (starting at 0). Depending on the device configuration, the number of QSE devices and the number of switches in the switch fabric may vary. The following sections explain the functional blocks of the QSE Driver.

The Device Driver Library (DDL)

The DDL for the QSE Driver provides the API functions to the application layer to control and configure the QSE Driver and the QSE devices.

Initialization Functions

The (DDL) contains the following initialization functions:

- Device initialization - Initialization of the device registers and associated RAMs based on a user-specified initialization vector.
- Self testing - Self testing includes testing on-chip RAM and associated external RAM.
- ISR installation - The driver supplies an interrupt handler and a location to install it during initialization when interrupts are enabled.
- Custom event handlers installation - QSE Drivers provide hooks into which you can supply calls to customize event handlers to meet system requirements.
- Buffer allocation - Buffers requiring pre-allocation will be allocated at initialization time.
- Queue creation - Required queues are created.
- Task creation - After successful initialization of the device, the QSE Driver spawns a background task that processes messages received from the ISR and periodically accumulates device counts.

Configuration Functions

The QSE Driver sets up the devices to switch cells for unicast and multicast traffic. The QSE Driver has configuration functions that perform the following:

- Add and clear multicast groups.
- Configure backpressure delay, cell start offset, maximum pending cells, interrupt mask, parity error mask, software reset, acknowledgment payloads, unicast drop mode, and aggregate mode.

Status and Statistics Functions

- The QSE Driver provides API functions for obtaining status and statistics.
- Statistics collection can be performed in response to an interrupt from the device, or in response to being polled periodically.
- For statistical information, the QSE Driver accumulates select hardware register counters into larger counters, which do not need to be read as often as the corresponding hardware register counters.

The QSE Driver Restart and Reinitialization Functions (DRRs)

The DRR helps the application initialize the QSE Driver and the switches in the QSE devices. The DRR is a set of the six functions explained in the following subsections.

QsePowerOnInit

The `QsePowerOnInit` function performs the power-on initialization of the QSE devices. The QSE_GDDB, QSE_DDB, and the QSE_DCB data structures should be initialized in this function. Parameters for the power-on initialization are taken from the QSE_DDB and the QSE_DCB data structures.

During power-on initialization, the QSE devices are taken out of the hardware reset, but not from the software reset state (assuming they are in the hardware reset state after power-on). If a QSE device is not in the hardware reset state, it is reset and taken out of the hardware reset state. Depending on the configuration parameters, power-on initialization is performed on all the QSE devices and associated switches controlled by the QSE Driver.

NOTE: The hardware reset resets the whole QSE device, whereas the software reset blocks any operation on the QSE Driver, but allows access to the QSE registers and to the external SRAM.

QseFinalInit

The `QseFinalInit` function performs the final initialization of the QSE Driver and the QSE devices. The QSE_GDDB and the QSE_DCB data structures are used to obtain the parameters for final initialization of the QSE device.

The `QseFinalInit` function performs the following operations during the final initialization:

- Resets all QSE devices to remove any after-effect of the POST.
- Reinitializes the driver global data structures.

After the final initialization, the QSE Driver is ready to execute commands from the application.

QseDelete

The `QseDelete` function removes the QSE Driver data structure, frees all the memory, and de-allocates all the resources allocated for the QSE device with `ui2DevId`. This function also resets the hardware reset bit in the chip mode register.

The QSE Driver Diagnostics Function (DDG)

QsePowerOnSelfTest

The `QsePowerOnSelfTest` function performs an exhaustive test on the various functions of the QSE devices and returns the result of the tests. It is performed after the power-on initialization phase. The following tests are performed during POST:

- Read/write test of all read/write registers of the QSE device using different patterns, such as 0xaa and 0x55 for 8-bit registers, and 0xaaaaaaaa and 0x55555555 for 32-bit registers.
- Memory tests are performed on all memory (external and internal RAM) associated with the QSE device. Memory tests are performed only when the device is used for multicast operations.
    - Pattern test: A gang mode (gang 1, 2, or 4) pattern is written in a memory location. Then the memory location is read and the read value is compared with the original pattern. The test passes if the written value and the read value are the same.
    - Walking-one test: A pattern with only one bit set is written and then read from a single memory location. The read value is compared with the written value. The set bit is rotated in the pattern and the test is repeated until all bit positions in the pattern are covered.
    - Address alias test: Each location in the memory is written with its address. Then the memory is read back and the read values are compared with the written values. If there is a memory alias error, then the read value will not be the same as the written value.

If the results of these tests do not indicate errors, then the final initialization of the QSE Driver and the QSE devices can be performed. Compile options are provided to switch off the memory tests and verification.

The POST function is separate from the other components of the QSE Driver. POST function routines can also be used for diagnostics purposes during development of hardware containing QSE devices.

The QSE Device Driver Operations (DDOs)

As shown in Figure 4 on page 9, the DDO block of the PM73488 QSE device contains the following modules:

- Initialization and Configuration module.
- Switch Management module.
- Status and Statistics module.

Configuration Module

The configuration functions set values in the QSE device control registers and set values in the data structures in the external and internal RAM. For example, `QseSetRowCol` (refer to "QseSetRowCol" on page 30) configures the row number and column number of the switch in the switch fabric.

Switch Management Module

The QSE Driver switch management functions allow the setting (Set) and the retrieving (Get) of a variety of switch management information in the QSE device(s) on a per-port basis. For example, the switch management module includes functions that support the following:

- Point-to-multipoint traffic setup.
- Point-to-point traffic setup.
- Port groupings to increase fault tolerance.
- Buffer reservations.
- Fairness threshold setup.

Status and Statistics Module

The QSE Driver status and statistics module provides status and statistics information about the different device parameters and the failures occurring in the switches. For example, the `QseGetInMarkedCellCount` function (refer to "QseGetInMarkedCellCount" on page 44) returns the number of tagged cells that entered the specified port.

SNMP Support

All the "set" API functions in Switch Management and "get" API functions in the Status and Statistics modules can be mapped to the SNMP Get and Set operations. The interrupt-driven mode of the DCT can be linked to the TRAP mechanism in the SNMP agent. This allows Network Management Systems (NMSs) to manage the switch remotely.

The QSE Driver Control Task (DCT)

The QSE DCT wakes up when notified by the ISR of any alarms, such as PARITY_FAIL and BP_ACK_FAIL. The DCT can then execute a callback function provided by the application layer to deal with the situation. Alternatively, it can signal the application task of the occurrence of this event.

The DCT runs continually after the QSE devices are initialized and running. The DCT starts a timer, then waits for an asynchronous event notification in an infinite loop. The DCT can receive an event notification from the following sources:

- The timer started by the DCT. The timer times out at periodic intervals, and upon timeout it signals the DCT. The DCT wakes up and collects counts maintained by the QSE device in its internal memories. The DCT also clears any interrupts that may have been masked by the last execution of the ISR routine.
- The ISR, which signals the DCT when an interrupt occurs. The DCT in turn passes this information to the application, either by executing a callback function or by signaling the application.

The following lists the pseudocode for the QSE Driver DCT.

```
/* QSE driver control task */
{
start a periodic timer

loop forever
     {
          wait on event (timer_expiry_event | message_received_event);
          if (event == timer_expiry_event) {
               for each PM73488
                    {
                         read device maintained counts and update their copies in the
                         driver database
                    }
          }
          if(event == message_received_event) {
               extract the message from the DCT's queue;
               process message to get message_type;
               switch(message_type) {
                    case QSE_SWTCH_INTERRUPT (All interrupts from the QSE Device)
                         call signal processing function which will notify application
                         using the call back function given by the application;
                         /* alternatively, send a message to application task */
                         break;
```

```
                    default:
                         return error;
                         break;
               } /* end of switch */
          } /* end of if message */
     } /* end of loop */
} /* end of task */
```

The Interrupt Service Routine (ISR)

The QSE Driver supplies an interrupt handler and a location to install it during initialization. The ISR performs the following functions:

- Traps hardware interrupts generated by the QSE device. These interrupts correspond to events, such as backpressure failure, parity failures, and port failures in the ports and cells.
- Acknowledges and masks the interrupts received to prevent a flood of device interrupts. These interrupts are unmasked at a later time by the DCT.
- Signals the DCT when these interrupts occur. The DCT in turn signals the application or executes callback functions provided by the application.
- Updates the DCB and DDB structures. For example, the ISR updates the alarm error counters in the DDB.

The ISR is invoked every time a QSE device raises its hardware interrupt signal. The ISR reads the interrupt and status register for each QSE device, determines the cause of the interrupt, and performs necessary actions. The ISR updates the appropriate DDB counters and signals the DCT. The ISR is designed to consume as little execution time as possible. The ISR can also be used in polling mode with interrupts masked, where periodic timer interrupts can be used to poll the interrupt status register to check for any interrupts. The ISR can be enabled by setting the QSE_USE_ISR flag to in the makefile. Some of the preprocessing (for example, checking the thresholds for certain types of interrupts) is performed in the signal processing function.

The following lists the pseudocode for the QSE Driver ISR.

```
/* ISR for the PM73488 Driver*/
{
     mask all interrupts;
     for each PM73488
     {
          read the interrupt status register;
          update the DDB counters for alarm interrupts;
          signal the DCT if necessary;
     }
     unmask interrupts
```

The Driver Data Structures

Overview

The following are the main global data structures of the QSE Driver:

- The QSE device Global Device Driver Database (GDDB)
- The QSE Device Control Block (DCB)
- The QSE Device Data Block (DDB)
- The Initialization Vector

Figure 5 shows the relationship between the GDDB, the DCB, and the DDB.

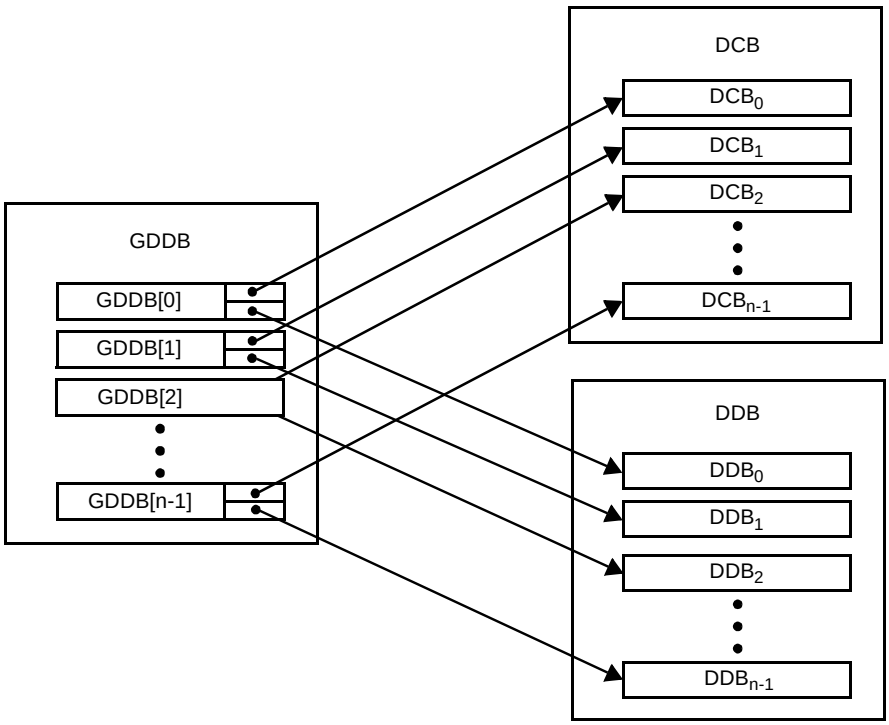

Figure 5. Relationship Between the GDDB, the DCB, and the DDB

Global Device Driver Database (GDDB)

Description: Applies to all QSE devices.

Data Type: tQSE_GDDB (struct)

Identifier: sQSE_GDDB sQseGddb[NUM_OF_QSEs]

The Global Device Driver Database (QSE_GDDB) maintains global parameters for the QSE Driver. This structure also maintains the pointers to the QSE device structures (the QSE_DCB and QSE_DDB), and to other data structures in the QSE Driver. The members of the structure are assigned values during power-on initialization. These values can be changed at run-time to support dynamic configuration of the QSE device. There is only one copy of this data structure in the QSE Driver. The database is declared as an array of the QSE_GDDB structure with NUM_OF_QSEs elements. Table 3 lists and describes the data types in the GDDB.

Table 3. Global Device Driver Database (GDDB)

Data Type	Field Name	Description
void *	ptrBaseAddress	Indicates the absolute address of start of QSE Driver.
UINT4	u4IntrptVector	Indicates the interrupt vector number of the QSE device.
UCHAR	ucStatus	Indicates the state of the QSE Driver (S0, S1, S2, and S3).
INT4	i4ScratchPad	Snapshots the previous return code.
tQSE_INITVECT	sInitvectQse	Specifies the QSE Driver initialization data.
tQSE_DCB *	ptrDcb	Points to the DCB. Refer to Table 4.
tQse_DDB *	ptrDdb	Points to DDB. Refer to Table 5 on page 17.

Device Control Block (DCB)

Description: Contains control information about the QSE device. Each element of the GDDB array (corresponding to each device) has a pointer to a DCB structure.

Data Type: tQSE_DCB

Identifier: QseGddb[ucDevId].ptrDcb (memory dynamically allocated)

Table 4 lists and describes the data types in the Device Control Block (DCB).

Table 4. Device Control Block (DCB)

Data Type	Field Name	Description
BOOLEAN	bExtRAM	Set to TRUE if external RAM is present. Otherwise, set to FALSE.
UINT4	ui4RAMSize	Size of the external RAM, if present. This value cannot exceed 32K.

Device Data Block (DDB)

Description: Stores statistics for individual devices being monitored by the QSE Driver. There is one DDB for each device being monitored.

Data Type: tQSE_DDB

Identifier: QseGddb[ucDevId].ptrDdb(memory dynamically allocated)

Table 5 lists and describes the data types in the DDB.

Table 5. Device Data Block (DDB)

Data Type	Field Name	Description
tQSE_LATCH_COUINTS	sLatchCounts	Accumulates the following information on each port on each device. 0 Backpressure acknowledgment failure. 1 Backpressure remote acknowledgment failure. 2 Input port failure. 3 Output port failure. 4 Parity error.
UINT4	ui4InMarkedCellCounts[32]	Counts all the tagged cells that arrive in all the ports in the QSE device.
UINT4	ui4OutMarkedCellCounts[32]	Counts all the tagged cells that leave all the ports in the QSE device.

Initialization Vector

Description: Contains the QSE Driver's initialization data, as configured by the user in the `QseDriverInit` function.

Data Type: tQSE_INITVECT

Identifier: tQSE_INITVECT *ptrInitVector

Table 6 lists and describes the data types in the QSE_INITVECT structure. The data types UINT8, UINT16, and UINT64 are all structures that store the number of bytes specified in the data types itself.

Table 6. Initialization Vector (QSE_INITVECT)

Data Type	Field Name	Description
UINT2	ui2RevisionReg	Revision number of the QSE device.
UINT2	ui2ChipMode	QSE device's configuration (single or double switch, chip hardware reset, parity check, external RAM).
UINT8	ui8McGrpIdx	Multicast group to be modified or read.
UINT16	ui16McGrpData	The multicast group mask written into RAM in the address given in the Group Index register for multicast operation.
UINT8	ui2McGrpOp	Read/write operation to be performed on the group index.
UINT2	ui2UcMcFairReg	Unicast/multicast behavior for cells of same priority.
UINT16	ui16InPortEnable	Enable/disable input ports and associated interrupts.
UINT16	ui16OutPortEnable	Enable/disable output ports and associated interrupts.
UINT16	ui16ParityErrPresent	Flags the port numbers where parity error occurred.
UINT16	ui16ParityErrLatch	Latches the parity error since last read.
UINT16	ui16SeInPortFailPresent	Flags the input ports that are failed. (Refer to the *PM73488 QSE Long Form Data Sheet* for a description of different failures.)
UINT16	ui16SeOutPortFailPresent	Flags the output ports that are failed.
UINT16	ui16SeInPortFailLatch	Latches the failed input ports since last read.
UINT16	ui16OutPortFailLatch	Latches the failed output ports since last read.
UINT16	ui16BpAckFailPresent	Flags the absence of pattern on the ports.
UINT16	ui16BpAckFailLatch	Latches the ports without patterns since last read.
UINT16	ui16BpAckRemoteFailPresent	Flags backpressure acknowledgment failures on the ports.
UINT16	ui16BpAckRemoteFailLatch	Latches the backpressure failed ports since last read.
tSWTCH_CNTL_STAT_REGS	asSwtchCntlStatRegs	Control/status registers.

OS Extensions

The PMC OS Extensions module is an operating system wrapper that is designed to provide a consistent interface to the underlying OS. The PMC OS Extensions module provides the following functionalities:

- Message queues
- Periodic timers
- Event notification
- Task management
- Memory management
- Debug logging

The PMC OS Extensions module separates the RTOS porting into a separate module (see Figure 6). The porting section calls the PMC OS Extensions module interface to perform RTOS actions. The only modifications required in the porting section are for device I/O and system configuration.

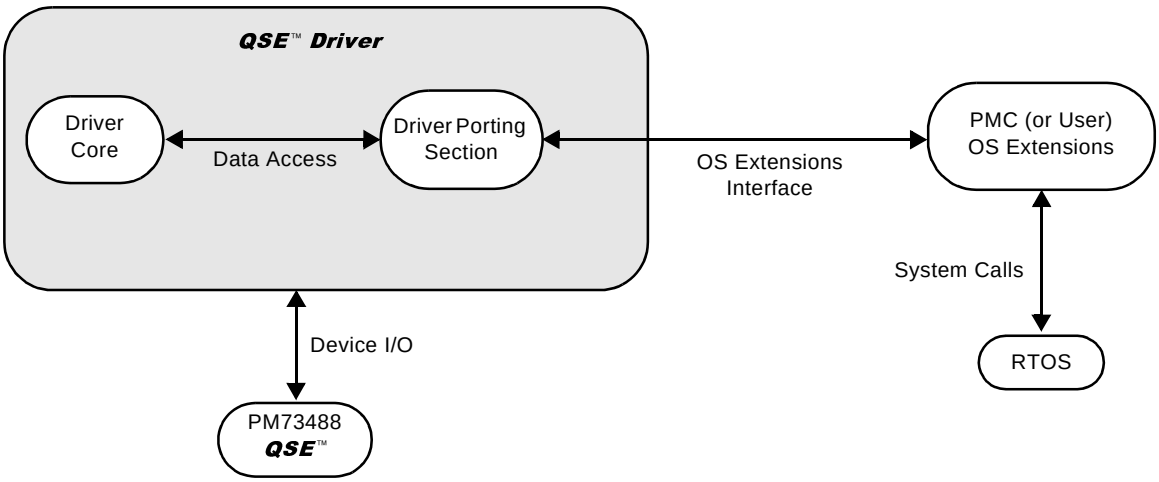

Figure 6. Model of a QSE Device and its Associated QSE Driver

Using the PMC OS Extensions module with the QSE Driver is optional. Customers may replace the PMC OS Extensions functions with their own OS-specific wrappers.

Chapter 4
Porting Guidelines

ABOUT THIS CHAPTER

This chapter describes how to port the QSE Driver onto a specific hardware and OS platform. Porting elements are in the form of constants in the qseport.h file and functions in the qseport.c file. The porting steps include developing additional code and defining the various macros and preprocessor constants used by the driver code. For easy porting, the changes required for a driver are grouped into two files: the qseport.h file, and the qseport.c file. Global changes required by all drivers are in two separate files: the gport.h file and the gport.c file.

In addition to the driver porting files, OS extensions must be ported for the target OS. PMC-Sierra provides OS Extensions, which is a wrapper that provides a consistent interface to the OS and is used in the QSE Driver (for more information on OS Extensions, refer to "OS Extensions" on page 19). Since the QSE Driver uses only a subset of the functions available in OS Extensions, porting only those functions is sufficient. The files that should be ported for use with the QSE Driver are:

- oextport.c
- oextport.h

NOTE: PMC-Sierra recommends that you do *not* modify the core files during porting.

HOW THE SOURCE CODE IS ORGANIZED

The code for the QSE Driver is organized into the C language files shown in Figure 7.

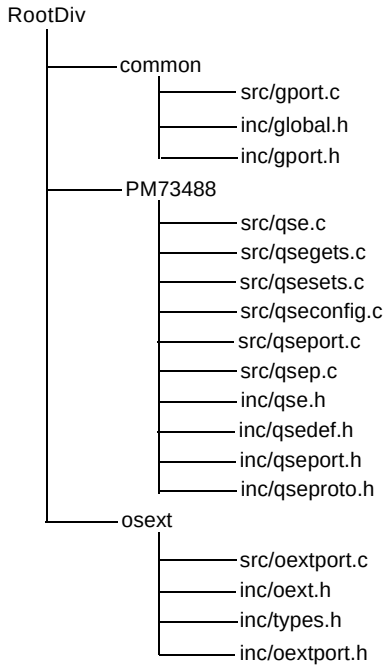

Figure 7. QSE Driver Source Files

The qseproto.h header file is necessary for the API functions to operate. This is the only header file that should be included by the application. It includes data structures, definitions, and prototypes of the QSE Driver API.

PORTING STEPS

To port the QSE Driver to a specific environment, complete the steps described in this section.

Step 1: Modify the gport.c File

Verify the implementations included in this file function correctly on the target system. By default, `EnablePreemption` and `DisablePreemption` use OS Extensions to control task preemption. This requires the actual porting to be done in OS Extensions.

Step 2: Modify the gport.h File

The gport.h file should not require porting, since it references OS Extensions for all OS-dependent operations. If required, task-specific constants for the QSE Drivers can be changed here. For most installations, this file will need no modifications.

Step 3: Port OS Extensions

The PMC-Sierra OS Extensions module encapsulates all OS-specific operations required by the QSE Driver. The PMC-Sierra OS Extensions provides operations for managing tasks, queues, timers, events, semaphores, memory, and debug logging. Queues, tasks, and semaphores are "named" in OS Extensions. A character string is assigned to each queue or task to uniquely identify it. For more information on the PMC-Sierra OS Extensions, refer to "OS Extensions" on page 19.

You can port either the PMC-Sierra OS Extensions module or your own OS extensions. The following substeps (3a, 3b, and 3c) describe what changes are required in each file if you choose to use the PMC-Sierra OS Extensions. If you choose to port your own OS extensions, you can skip step 3b.

Step 3a: Modify the types.h File

The types.h file defines the system- and compiler-specific type definitions required by OS Extensions. The types are identified by the number after the type. For example, UINT4 defines a 4-byte (32-bit) unsigned integer. Substitute the compiler types that yield the desired types as defined in this file.

Step 3b: Modify the oextport.h File

This step is required only if you are using the PMC OS Extensions module.

This file contains the preprocessor constants used by each different class of OS Extensions calls. Most values defined in this file do not need to be changed, unless doing so eases the porting of the oextport.c file. For example, the constant EV_COND_OR is used by `qx_receive` to indicate the wait condition is a logical OR of the event flags. The value of the EV_COND_OR constant in oextport.h can be changed to represent the actual value of this flag in the underlying OS call used in porting `qx_receive`. This allows more efficient operation, since the value of EV_COND_OR does not need to be translated to the value used by the actual OS call. Placing these constants in this file gives the developer more flexibility in porting OS Extensions.

NOTE: The name of the constants in this file should *not* be changed since doing so would alter the interface to OS Extensions.

Step 3c: Code the oextport.c File

The oextport.c file contains the code that implements the actual OS Extensions function calls. Most of the code in this file will have to be ported specifically for the target OS. The QSE Driver uses only a subset of the function calls defined in the oextport.c and oextport.h files. The following list summarizes the function calls required by the QSE Driver and gives a brief description of each function call.

NOTE: If you choose to port your own OS extensions (that is, you are not using the PMC-Sierra OS Extensions module), substitute code that implements your OS extension calls.

- Events
  - `evx_send` - Send a set of events to a task.
  - `evx_receive` - Wait for a set of events governed by a logical operation and a timeout.
- Memory
  - `mx_create` - Allocate a memory block of a given size.
  - `mx_delete` - Free a memory block allocated with `mx_create`.
  - `mx_set_value` - Set a constant byte value in a memory region.
- Queues
  - `qx_create` - Create a named queue with a given size, and optionally specify an event that will be sent to a task when a message arrives in the queue.
  - `qx_get_buffer` - Allocate a message to send with `qx_send`.
  - `qx_ident` - Return the queue identifier associated with a named queue.
  - `qx_receive` - Receive a message sent to a queue with an optional timeout period.
  - `qx_return_buffer` - Free a message after receiving it with `qx_receive`.
  - `qx_send` - Send a message to a queue, and optionally specify a named queue for the response.
- Timers
  - `tmx_evevery` - Send a set of events to a task at the specified interval.
  - `tmx_wkafter` -Wakes the task up the specified time period.
- Tasks
  - `tx_mode` - Set the current mode of the task.
  - `tx_start` - Create and start a new task with the given stack size, priority, arguments, and starting point.
- Debug Logging
  - `x_trace` - Send a message to the debug log with a key and debug level.

IMPORTANT! The interface to the PMC OS Extensions module must not change during porting. Do *not* modify the function names, their parameters, or the constant names used by the functions while porting. Doing so will make it more difficult to port future versions of the QSE Driver. If a parameter or constant does not make sense for the target system, then leave it as defined and ignore it.

Step 4: Assign Proper Values to the Device Specification Constants in the qseport.h File

Following are the constants defined in the qseport.h file. Define them as specified by the hardware environment.

`#define NUM_OF_QSE_DEVICES`
The number of QSE devices this driver is going to control.

`#define QSE_BASE_ERROR_CODE`
Each block of error codes has a base address that is added to the defined error codes. This way, the defined error base can be used to indicate from which driver portion the error originates. The QSE device uses `-100` as the current error base.

`#define QSE_REV_NO`
The QSE Driver is designed to work with certain revisions of the QSE device. These two constants describe the revision boundaries for which this QSE Driver was designed and tested. The current revision for the PM73488 QSE is 0x0001.

Step 5: Assign Proper Values to the Driver Task-Related Constants in the qseport.h File

`#define QSE_DRIVER_TASK_NAME`
The QSE Driver task name.
Default: `QSET`

`#define QSE_DRIVER_TASK_PRIORITY`
The QSE Driver task priority.
Default: `40`

`#define QSE_DRIVER_TASK_STACK`
The QSE Driver task stack size, in bytes.
Default: `8192`

`#define QSE_DRIVER_TASK_MODE`
The QSE Driver task mode. The value set allows for preemption only.
Default: `4`

Step 6: Assign Proper Values to the Queue-Related Constants in the qseport.h File

`#define QSE_DRIVER_QUEUE_NAME`
The QSE Driver queue name.
Default: `QSEQ`

`#define QSE_DRIVER_QUEUE_DEPTH`
The QSE Driver queue depth.
Default: `40`

Step 7: Modify the qseport.c File

Determine the desired configuration of the `QseDriverInit` function within the QSE Driver code, and change the function accordingly. The `QseDriverInit` function should basically initialize the GDDB entries for all the devices. Some typical entries to be initialized are:

- The physical base address of the device.
- The default parameter values with which `QsePowerOnInit` will initialize the devices.
- The interrupt vector number to be used by the device (if any) so the ISR defined by the QSE Driver is executed as a part of the interrupt servicing.
- The configuration parameters of the QSE Driver software.
- The external RAM configurations.

The `QseStartTimer` function sends periodic events to the DCT to allow the DCT to perform time-dependent operations. The implementation of `QseStartTimer` uses the `tmx_evevery` function in OS Extensions to send the periodic events to the DCT. Because it uses OS Extensions, the `QseStartTimer` function should not require modification during porting.

The `QseSigFn` function is used to inform the DCT, if present, of events. The implementation of `QseSigFn` uses the `qx_get_buffer` and `qx_send` functions in OS Extensions to send messages. The `QseSigFn` function is designed to be called from an ISR external to the QSE Driver code. In certain OS environments, the functions that can be called from within an ISR can be very limited. In such cases, it may be useful to pass all the signals to a dedicated task that will then signal the applications in an appropriate manner.

Step 8: Define the Hardware Write and Read Functions in the qseport.c File

Implement the following functions in the qseport.c file. These functions will be affected by the endianness of the hardware platform and your compiler option.

`void QseIn (UNIT4 *pui4Addr, UNIT4 *pui4Data)`

This function is used to write data to the QSE device registers. Internal and external RAM blocks are also written through the QSE device registers. This function can be modified according to system requirements (for example: byte ordering).

`void QseOut (UNIT4 *pui4Addr, UNIT4 pui4Data)`

This function is used to read data from the QSE device registers, internal RAM blocks, and external RAM blocks. This function can be modified according to system requirements (for example, byte ordering).

Step 9: Code and Install the Interrupt Handler

The ISR is expected to be executed when the device interrupts the processor. Program the OS or the CPU so the device ISR is executed once for every interrupt caused by the device. Typically, you can do this by coding an interrupt handler that appropriately handles the interrupt hardware of the CPU (for example, code an ISR that issues acknowledgments to the interrupt controller hardware and masks lower priority interrupts), and then calls the ISR provided by the QSE Driver. Then, at the end of the `QseDriverInit` function, install this interrupt handler so it executes when an interrupt for this device occurs.

Step 10: Compile and Link the Source Code Files into Library/Object Modules

Generate the QSE Driver libraries by compiling and linking all the source and header files. Follow the conventions required by the target OS to generate the QSE Driver library. For example, apply the proper compile and link switches. The following compile switches are defined for building the QSE Driver library

`QSE_CSW_SKIP_EXT_RAM_TEST`
When this macro is defined, the external RAM test is skipped while executing `QsePowerOnSelfTest`.

`QSE_CSW_USE_ISR`
Enables the ISR routine.

`QSE_CSW_CONTINUE_ON_ERROR`
This macro causes the entire self-test process to run until it has logged all errors. If this macro is disabled, the self-test will stop at the first error.

`QSE_CSW_VERSION_B`
This macro includes the features related to the PM73488 version B chip in the code.

After setting the appropriate compile switches, build the QSE Driver library using your make utility. Generate a target executable by compiling and linking the QSE Driver library with your application code.

Chapter 5
QSE Driver API

ABOUT THIS CHAPTER

This chapter describes the functions within the QSE Driver API.

QSE DRIVER API FUNCTIONS

The QSE Driver provides configuration, status, and statistics functions to configure, control, and monitor QSE devices. These functions provide a high-level and an easy-to-use interface to the QSE device. This section describes the primary functions provided by the QSE Driver to the application layer. The functions are grouped according to the QSE device functionality they support.

Initialization Functions

The initialization functions perform initial device setup for the QSE devices in the switch fabric.

QseDriverInit

Description: The QSE Driver provides a single initialization function, `QseDriverInit`, which is part of the driver's porting section. Within this function, the user specifies the set of desired initialization parameters in the form of an initialization vector. The `QseDriverInit` function initializes the QSE Driver's data structures based on the initialization vector. It then calls `QsePowerOnInit`, `QsePowerOnSelfTest`, and `QseFinalInit` to move the driver from state S0 to S3, the operational state. The DCT is started to handle interrupts. The device configuration (DEVICE_MODE) structure type for each device is shown in Table 7.

Table 7. QSE Driver Initialization Function Structure (DEVICE_MODE)

Structure Variables	Description
bRamStatus	If TRUE, external SRAM is present. Otherwise, it is absent.
bParityCheck	If TRUE, parity check is enabled. Otherwise, it is disabled.

Invocation:
```
INT4 QseDriverInit(UINT2 ui2NumDev, DEVICE_MODE *pasDevMode)
```

Inputs:

Argument	*Description*
ui2NumDev	Indicates the number of QSE devices controlled by the QSE Driver.
pasDevMode	Configuration data structure array for all the QSE devices and the switches controlled by the QSE Driver.

Outputs: None.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_EXTERNAL_RAM
QSE_ERR_INVALID_INIT_VECTOR
QSE_ERR_MEMORY_ALLOC_FAIL
QSE_ERR_READBACK_REG_TEST
QSE_ERR_SELF_TEST_PREP_FAIL
```

QsePowerOnInit

Description: The `QsePowerOnInit` function (which is called by the `QseDriverInit` function, refer to "QseDriverInit" on page 27) checks the device state (which should be S0), reads and validates the QSE device revision number, and sets the device state to S1.

Invocation: `QsePowerOnInit (UINT2 ui2DevId)`

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.

Outputs: None.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_ABSENT
QSE_ERR_INVALID_DEVICE_NO
QSE_ERROR_DEVICE_IN_WRONG_STATE
```

QsePowerOnSelfTest

Description: The `QsePowerOnSelfTest` function configures the QSE device with initialization vectors formed by `QseDriverInit`. The `QsePowerOnSelfTest` function reads back the register values to determine if the register writes were successful. It checks the internal and external RAMs of the QSE device. It is usually called within the `QseDriverInit` function.

Invocation: `QsePowerOnSelfTest (UINT2 ui2DevId)`

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.

Outputs: None.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_EXTERNAL_RAM
QSE_ERR_INVALID_DEVICE_NO
QSE_ERR_INVALID_INIT_VECTOR
QSE_ERR_READBACK_REG_TEST
QSE_ERR_SELF_TEST_PREP_FAIL
```

QseDelete

Description: The `QseDeleteDev` function deletes the data structure related to the device ID specified.

Invocation: `QseDelete (UINT2 ui2DevId)`

Inputs:

Argument	*Description*
`ui2DevId`	Identifies the QSE device.

Outputs: None.

Returns:
`QSE_SUCCESS`
`QSE_ERR_INVALID_DEVICE_NO`

QseFinalInit

Description: The `QseFinalInit` function initializes the device as specified by the `ptrInitVector` parameter. If the `ptrInitVector` is `NULL`, the current configuration of the device is not disturbed. It allocates and initializes other driver data structures and installs times for the overall functioning of the QSE Driver. The `QseFinalInit` function is usually called within the `QseDriverInit` function.

Invocation: `QseFinalInit (INT2 i2DevId, tQSE_INITVECT *ptrInitVector)`

Inputs:

Argument	*Description*
`i2DevId`	Identifies a particular QSE device.
`ptrInitVector`	Identifies the initialization vector for configuring the QSE device.

Outputs: None.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`
`QSE_ERR_INVALID_INIT_VECTOR`
`QSE_ERR_MEMORY_ALLOC_FAIL`

Configuration Functions

Configuration functions translate user-supplied configuration parameters to appropriate device register values, and set the relevant registers with these values. They also allow configuration data residing in the device to be retrieved at any time. The following sections describe the primary configuration functions.

QseSetRowCol

Description: The `QseSetRowCol` function sets the row and column number of the switch in the fabric. A QSE device can determine its position in the switch fabric using these numbers.

Invocation:
```
QseSetRowCol (UINT2 ui2DevId, UINT2 ui2RowNum, UINT2 ui2ColNum)
```

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`ui2RowNum`	Identifies a row number of the switch in the fabric.
`ui2ColNum`	Identifies a column number of the switch in the fabric.

Outputs: None.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_INVALID_COL_NUM
QSE_ERR_INVALID_DEVICE_NO
QSE_ERR_INVALID_ROW_NUM
QSE_WRN_ALREADY_SAME_VALUE
```

QseSoftReset

Description: The `QseSoftReset` function sets and resets the software reset bit in the control register. When SRAM writing is finished, if this bit is in the reset mode, the SRAM memory write will be faster than when this bit is in the set mode. (Refer to the *PM73488 QSE Long Form Data Sheet* for more information.)

Invocation:
```
QseSoftReset (UINT2 ui2DevId, BOOLEAN bFlag)
```

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`bFlag`	Flag bit that sets or resets the software reset bit in the control register. `1` Software reset. `0` Software set.

Outputs: None.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_INVALID_DEVICE_NO
QSE_WRN_ALREADY_SAME_VALUE
```

QseSetAckPayload

Description: The `QseSetAckPayload` function sets the acknowledgment payload for parity error and congestion notification. The default value for parity error is `ONACK` and for congestion notification is `MNACK`. The payloads are 4-bits wide.

Invocation:
```
QseSetAckPayload (UINT2 ui2DevId, UINT2 ui2ParityAck,
UINT2 ui2CongAck)
```

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`ui2ParityAck`	Parity error acknowledgment payload. Default: `ONACK`.
`ui2CongAck`	Congestion notification acknowledgment payload. Default: `MNACK`.

Outputs: None.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_INVALID_DEVICE_NO
QSE_WRN_ALREADY_SAME_VALUE
```

QseSetDeadGangAckPayload

Description: The `QseSetDeadGangAckPayload` function sets the acknowledgment payload to send when an entire gang is dead. The payloads are 4-bits wide.

Invocation:
```
QseSetDeadGangAckPayload (UINT2 ui2DevId,UINT2 ui2AckPayload)
```

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`ui2AckPayload`	Gang dead acknowledgment payload.

Outputs: None.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_INVALID_DEVICE_NO
QSE_WRN_ALREADY_SAME_VALUE
```

QseSetPhaseAligner

Description: The `QseSetPhaseAligner` function turns the phase aligner of the QSE device on or off. The default setting is on (`TRUE`).

Invocation: `QseSetPhaseAligner (UINT2 ui2DevId, ON_OFF eFlag)`

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`eFlag`	Turns the phase aligner on or off. `TRUE` Turns the phase aligner on. `FALSE` Turns the phase aligner off.

Outputs: None.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`
`QSE_WRN_ALREADY_SAME_VALUE`

QseSetBpDly

Description: The `QseSetBpDly` function can set the backpressure to either early backpressure or optimal backpressure.

Invocation: `QseSetBpDly (UINT2 ui2DevId, BP_MODE eBpMode)`

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`eBpMode`	Indicates if the backpressure is set to early or optimal. `1` Backpressure is set to early. `0` Backpressure is set to optimal.

Outputs: None.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`
`QSE_WRN_ALREADY_SAME_VALUE`

QseSetCellStartOffset

Description: The `QseSetCellStartOffset` function sets the window for clock cycles within which the cells should start arriving. The default value is 4. The maximum value is 8 clock cycles.

Invocation: `QseSetCellStartOffsset (UINT2 ui2DevId, UINT2 ui2Clk)`

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`ui2Clk`	Indicates the clock cycle window size. The maximum value is 8. The default value is 4.

Outputs: None.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`
`QSE_WRN_ALREADY_SAME_VALUE`

QseSetParityErrIntMask

Description: The `QseSetParityErrIntMask` function sets the parity interrupt mask bit for the port specified.

Invocation:
```
QseSetParityErrIntMask (UINT2 ui2DevId, UINT2 ui2PortNo,
BOOLEAN bStatus)
```

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`ui2PortNo`	Identifies the port number for which the parity interrupt mask is to be set.
`bStatus`	Mask bit that masks or unmasks the interrupt mask. `TRUE` Masks the interrupt. `FALSE` Unmasks the interrupt mask.

Outputs: None.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`
`QSE_ERR_INVALID_PORT_NO`

QseSetInPortEnable

Description: The `QseSetInPortEnable` function enables or disables the input port specified.

Invocation:
```
QseSetInPortEnable (UINT2 ui2DevId, UINT2 ui2PortNo,
BOOLEAN bStatus)
```

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`ui2PortNo`	Indicates the port number to be enabled or disabled.
`bStatus`	Enables or disables the input port. `TRUE` Enables the input port. `FALSE` Disables the input port.

Outputs: None.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_INVALID_DEVICE_NO
QSE_ERR_INVALID_PORT_NO
QSE_WRN_ALREADY_SAME_VALUE
```

QseSetOutPortEnable

Description: The `QseSetOutPortEnable` function enables or disables the output port specified.

Invocation:
```
QseSetOutPortEnable (UINT2 ui2DevId, UINT2 ui2PortNo,
BOOLEAN bStatus)
```

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`ui2PortNo`	Indicates the port number to be enabled or disabled.
`bStatus`	Enables or disables the output port. `TRUE` Enables the output port. `FALSE` Disables the output port.

Outputs: None.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_INVALID_DEVICE_NO
QSE_ERR_INVALID_PORT_NO
QSE_WRN_ALREADY_SAME_VALUE
```

QseSetExtMcRAMParityInt

Description: The `QseSetExtMcRAMParityInt` function enables or disables the parity interrupt for the external multicast RAM.

Invocation: `QseSetExtMcRAMParityInt (UINT2 ui2DevId, BOOLEAN bMode)`

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`bMode`	Enables or disables the parity interrupt for the external multicast RAM. `TRUE` Enables the parity interrupt when a parity error occurs. `FALSE` Disables the parity interrupt.

Outputs: None.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`
`QSE_WRN_ALREADY_SAME_VALUE`

QseSetIntrMask

Description: The `QseSetIntrMask` enables and disables all the interrupts on the QSE device, depending on the interrupt mask that is passed.

Invocation: `QseSetIntrMask (UINT2 ui2DevId, BOOLEAN bMask)`

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`bMask`	Enables or disables the interrupts. `1` Enables the interrupts. `0` Disables the interrupts.

Outputs:

Argument	*Description*
`TRUE`	Indicates the interrupt was set previously.
`FALSE`	Indicates the interrupt was not set previously.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`
`QSE_WRN_ALREADY_SAME_VALUE`

QseSetShortTags

Description: The `QseSetShortTags` function sets the tag rotation mode. When enabled, the QSE will rotate only the first five nibbles of the eight nibbles available for the tag. If disabled, the QSE will use all eight nibbles for tagging.

Invocation: `QseSetShortTags (UINT2 ui2DevId,BOOLEAN bMode)`

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`bMode`	Enables or disables the tag rotation mode. `TRUE` Enables the 5-nibble tag rotation mode. `FALSE` Enables the 8-nibble tag rotation mode.

Outputs: None.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`
`QSE_WRN_ALREADY_SAME_VALUE`

QseSetExtMcRAMParityCheck

Description: The `QseSetExtMcRAMParityCheck` function enables or disables the multicast RAM parity check bit in the extended chip mode register.

Invocation: `QseSetExtMcRAMParityCheck (UINT2 ui2DevId,BOOLEAN bMode)`

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`bMode`	Enables or disables the multicast RAM parity check bit in the extended chip mode register. `TRUE` Enables the multicast RAM parity check bit in the extended chip mode register. `FALSE` Disables the multicast RAM parity check bit in the extended chip mode register.

Outputs: None.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`
`QSE_WRN_ALREADY_SAME_VALUE`

Switch Management Functions

QseAddMCGrp

Description: The `QseAddMCGrp` function adds multicast group mask into the RAM at the given index. When a cell arrives with this index, the cell will be duplicated on all the ports where the multicast group mask bits are high. The QSE device should be set in software reset mode when calling this function.

Invocation:
```
QseAddMCGrp (UINT2 ui2DevId, UINT4 ui4Index, UINT4 ui4Gang,
BOOLEAN bAutoIncr)
```

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`ui4Index`	Index in the RAM into which the gang mode is written. Should be set to `-1` if `bAutoIncr` is `TRUE`.
`ui4Gang`	Multicast group associated with the index (for example, gang 1, 2, or 4).
`bAutoIncr`	Indicates if the index will be automatically incremented. `TRUE` The index will be automatically incremented. `FALSE` The `ui4Gang` parameter will be set in the index pointed to by `ui4index`.

Outputs: None.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_INVALID_DEVICE_NO
```

NOTE: The index (`ui4Index`) should be initialized by setting `bAutoIncr` to `FALSE` before it can be used to automatically increment the index.

QseClearMCGrp

Description: The `QseClearMCGrp` function clears the multicast group mask corresponding to the index from the RAM. The QSE device should be set in software reset mode when calling this function.

Invocation:
```
QseClearMCGrp (UINT2 ui2DevId, UINT4 ui4Index)
```

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`ui4Index`	Identifies the index of the multicast group in the RAM to be deleted.

Outputs: None.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_INVALID_DEVICE_NO
QSE_ERR_INVALID_INDEX
```

QseSetFairness

Description: The `QseSetFairness` function sets the fairness given to unicast cells over multicast cells that is used by the arbiter in the QSE device when there is conflict for ports among unicast and multicast cells with the same priority.

Invocation: `QseSetFairness (UINT2 ui2DevId, UINT2 ui2Fairness)`

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`ui2Fairness`	Indicates the weight that will be used to give preference for unicast cells over multicast cells.

Outputs: None.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`
`QSE_WRN_ALREADY_SAME_VALUE`

QseSetBufferResv

Description: The `QseSetBufferResv` function can reserve the internal buffer usage for different priority multicast cells to any of the four settings (listed below) for 32 ports.

Invocation: `QseSetBufferResv (UINT2 ui2DevId, BUFFER_MODE eBufResv)`

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`eBufResv`	Indicates the buffer reservation setting, which is one of the following: `MODE1`: • Four buffers are reserved for high priority cells. • Four buffers are reserved for high or medium priority cells. • All other buffers can be used by any cell. `MODE2`: • Four buffers are reserved for high priority cells. • All other buffers can be used by any cell. `MODE3`: • Eight buffers are reserved for high or medium priority cells. • All other buffers can be used by any cell. `MODE4`: • All buffers can be used by any cell.

Outputs: None.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`
`QSE_WRN_ALREADY_SAME_VALUE`

QseSetUCDropMode

Description: The `QseSetUCDropMode` function allows the higher layer software to set the QSE device to either drop all unicast cells or send ONACK back, if aggregate ports are off or dead. The default value is ONACK all dropped unicast cells.

Invocation: `QseSetUCDropMode (UINT2 ui2DevId, BOOLEAN bFlag)`

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`bFlag`	Sets the unicast drop mode of the switch. `TRUE` The device drops unicast cells. `FALSE` The device sends ONACK if ports are not functional.

Outputs: None.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`
`QSE_WRN_ALREADY_SAME_VALUE`

QseSetMaxPendingCells

Description: The `QseSetMaxPendingCells` function sets the maximum number of pending cells in a port. There can be a maximum of either three or four pending cells.

Invocation: `QseSetMaxPendingCells (UINT2 ui2DevId, MAX_CELL_MODE eMaxMode)`

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`eMaxMode`	Indicates the maximum cells that can be pending. It can take either three or four cells at the maximum.

Outputs: None.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`
`QSE_WRN_ALREADY_SAME_VALUE`

QseSetMCAggrMode

Description: The `QseSetMCAggrMode` function sets the aggregate mode for multicast traffic. The consecutive `eAggrIn` and `eAggrOut` output ports are considered a single input and output by the switch.

Invocation:
```
QseSetMCAggrMode (UINT2 ui2DevId, AGGR_IN_MODE eAggrIn,
AGGR_OUT_MODE eAggrOut)
```

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`eAggrIn`	Number of consecutive ports to be treated as a single input.
`eAggrOut`	Number of consecutive ports to be treated as a single output.

Outputs: None.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_INVALID_DEVICE_NO
QSE_WRN_ALREADY_SAME_VALUE
```

QseSetUCAggrMode

Description: The `QseSetUCAggrMode` function sets the aggregate mode for unicast traffic. The consecutive `eAggrOut` output ports are considered a single output by the switch.

Invocation:
```
QseSetUCAggrMode (UINT2 ui2DevId, AGGR_OUT_MODE eAggrOut)
```

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`eAggrOut`	Number of consecutive ports to be treated as a single output.

Outputs: None.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_INVALID_DEVICE_NO
QSE_WRN_ALREADY_SAME_VALUE
```

QseSetMgiMsb

Description: The `QseSetMgiMsb` function sets the Most Significant Bit (MSB) of the multicast group index to `1` or `0`.

Invocation: `QseSetMgiMsb (UINT2 ui2DevId, UINT2 ui2MsbIndex)`

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`ui2MsbIndex`	Identifes the MSB of the multicast group index.

Outputs: None.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`

Status and Statistics Functions

QseGetDeadGangAckPayload

Description: The `QseGetDeadGangAckPayload` function retrieves the acknowledgment payload that is sent when the entire gang of ports are dead.

Invocation:
```
QseGetDeadGangAckPayload (UINT2 ui2DevId, UINT2 *pui2AckPayload)
```

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.

Outputs:

Argument	*Description*
`pui2AckPayload`	Gang dead acknowledgment payload.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_INVALID_DEVICE_NO
QSE_WRN_ALREADY_SAME_VALUE
```

QseGetInPortEnable

Description: The `QseGetInPortEnable` function is used to read the input port enable register. Each of the 32 bits corresponds to the input port number specified by the bit position.

Invocation:
```
QseGetInPortEnable (UINT2 ui2DevId, UINT2 ui2PortNo,
UINT4 *pui4Status)
```

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`ui2PortNo`	Identifies the port number in the switch.

Outputs:

Argument	*Description*
`pui4Status`	Returns the status of all the ports.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_INVALID_DEVICE_NO
QSE_WRN_ALREADY_SAME_VALUE
```

QseGetOutPortEnable

Description: The `QseGetOutPortEnable` function is used to read the output port enable register. Each of the 32 bits corresponds to the output port number given by the bit position.

Invocation:
```
QseGetOutPortEnable (UINT2 ui2DevId, UINT2 ui2PortNo,
UINT4 *pui4Status)
```

Inputs:

Argument	*Description*
`ucDevId`	Identifies a particular QSE device.
`ui2PortNo`	Identifies the port number in the switch.

Outputs:

Argument	*Description*
`pui4Status`	Returns the status of the output port.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_INVALID_DEVICE_NO
```

QseGetParityErrPresent

Description: The `QseGetParityErrPresent` function returns the parity error status on all the input ports in the last cell time.

Invocation:
```
QseGetParityErrPresent (UINT2 ui2DevId, UINT4 *pui4Status)
```

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.

Outputs:

Argument	*Description*
`pui4Status`	Returns the parity error status of all the ports.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_INVALID_DEVICE_NO
```

QseGetInMarkedCellCount

Description: The `QseGetInMarkedCellCount` function returns the number of tagged cells that went through the specified input port.

Invocation:
```
QseGetInMarkedCellCount (UINT2 ui2DevId, UINT2 ui2PortNo,
UINT4 *pui4Count)
```

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`ui2PortNo`	Identifies the port number in the QSE device.

Outputs:

Argument	*Description*
`pui4Count`	Returns the number of tagged cells.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_INVALID_DEVICE_NO
```

QseGetOutMarkedCellCount

Description: The `QseGetOutMarkedCellCount` function returns the number of tagged cells that went through the specified output port.

Invocation:
```
QseGetOutMarkedCellCount (UINT2 ui2DevId, UINT2 ui2PortNo,
UINT4 *pui4Count)
```

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.
`ui2PortNo`	Identifies the port number in the QSE device.

Outputs:

Argument	*Description*
`pui4Count`	Returns the number of tagged cells.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_INVALID_DEVICE_NO
```

QseGetBpAckFail

Description: The `QseGetBpAckFail` function returns `TRUE` if there a backpressure acknowledgment failure in the port specified during the last cell time.

Invocation:
```
QseGetBpAckFail (UINT2 ui2DevId,UINT2 ui2PortNo,
BOOLEAN *pbStatus)
```

Inputs:	*Argument*	*Description*
	`ui2DevId`	Identifies a particular QSE device.
	`ui2PortNo`	Identifies the port number in the QSE device.

Outputs:	*Argument*	*Description*
	`pbStatus`	Returns the status of the port number identified.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_INVALID_DEVICE_NO
```

QseGetBpRemoteAckFailPresent

Description: The `QseGetBpRemoteAckFailPresent` function returns the backpressure remote fail error status of the port specified since the last cell time.

Invocation:
```
QseGetBpRemoteAckFailPresent (UINT2 ui2DevId, UINT2 ui2PortNo,
UINT4 *pui4Status)
```

Inputs:	*Argument*	*Description*
	`ui2DevId`	Identifies a particular QSE device.
	`ui2PortNo`	Port number for which the status returned.

Outputs:	*Argument*	*Description*
	`pui4Status`	Returns the remote failure status of the port specified since the last cell time.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_INVALID_DEVICE_NO
QSE_ERR_INVALID_PORT_NO
```

QseGetParityErrLatch

Description: The `QseGetParityErrLatch` function returns the parity error status on all the input ports since the last call to this function.

Invocation: `QseGetParityErrLatch (UINT2 ui2DevId,UINT4 *pui4Status)`

Inputs:	*Argument*	*Description*
	`ui2DevId`	Identifies a particular QSE device.

Outputs:	*Argument*	*Description*
	`pui4Status`	Returns the status of all the ports. The bit position corresponds to the port number in the QSE device.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`

QseGetRowCol

Description: The `QseGetRowCol` function gets the row and column number of the switch in the switch fabric.

Invocation: `QseGetRowCol (UINT2 ui2DevId,UINT2 *pui2Row, UINT4 *pui2Col)`

Inputs:	*Argument*	*Description*
	`ui2DevId`	Identifies a particular QSE device.

Outputs:	*Argument*	*Description*
	`pui2RowNum`	Returns the row number of this switch.
	`pui2ColNum`	Returns the column number of this switch.

Returns:
`QSE_SUCCESS`
`QSE_ERR_INVALID_DEVICE_NO`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_ROW_NUM`
`QSE_ERR_INVALID_COL_NUM`

QseGetFairness

Description: The `QseGetFairness` function gets the fairness weight given for unicast cells over multicast cells by the arbiter.

Invocation: `QseSetFairness (UINT2 ui2DevId, UINT2* pui2Fairness)`

Inputs:	*Argument*	*Description*
	`ui2DevId`	Identifies a particular QSE device.

Outputs:	*Argument*	*Description*
	`pui2Fairness`	Indicates the weight being used to give preference for unicast cells over multicast cells.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`

QseGetBufferResv

Description: The `QseGetBufferResv` function retrieves the buffer reservation setting for the switch specified.

Invocation: `QseGetBufferResv (UINT2 ui2DevId, BUFFER_MODE *peBufResv)`

Inputs:	*Argument*	*Description*
	`ui2DevId`	Identifies a particular QSE device.

Outputs:	*Argument*	*Description*
	`peBufResv`	Indicates the buffer reservation setting. Refer to "QseSetBufferResv" on page 38 for the possible buffer reservation settings.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`

QseGetCellStartOffset

Description: The `QseGetCellStartOffset` function retrieves the window for clock cycles within which the cells should start arriving.

Invocation: `QseGetCellStartOffsset (UINT2 ui2DevId, UINT2 *pui2Clk)`

Inputs:	*Argument*	*Description*
	`ui2DevId`	Identifies a particular QSE device.

Outputs:	*Argument*	*Description*
	`pui2Clk`	Indicates the clock window size. The maximum value is eight clock cycles.

Returns: `QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`

QseGetUCAggrMode

Description: The `QseGetUCAggrMode` function retrieves the aggregate mode for unicast traffic. The consecutive `peAggrOut` output ports are considered a single output by the switch.

Invocation: `QseGetUCAggrMode (UINT2 ui2DevId,AGGR_OUT_MODE *peAggrOut)`

Inputs:	*Argument*	*Description*
	`ui2DevId`	Identifies a particular QSE device.

Outputs:	*Argument*	*Description*
	`peAggrOut`	The number of consecutive ports to be treated as a single output.

Returns: `QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`

QseGetAckPayload

Description: The `QseGetAckPayload` function retrieves the acknowledgment payload sent when there is a parity error and congestion notification.

Invocation:
```
QseGetAckPayload (UINT2 ui2DevId, UINT2 *pui2ParityAck,
UINT2 *pui2CongAck)
```

Inputs:

Argument	*Description*
`ui2DevhId`	Identifies a particular QSE device.

Outputs:

Argument	*Description*
`pui2ParityAck`	Parity error acknowledgment payload.
`pui2CongAck`	Congestion notification acknowledgment payload.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_INVALID_DEVICE_NO
```

QseGetUCDropMode

Description: The `QseGetUCDropMode` function retrieves the unicast drop mode parameters from the switch.

Invocation:
```
QseSetUCDropMode (UINT2 ui2DevId, BOOLEAN *pbMode)
```

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.

Outputs:

Argument	*Description*
`pbMode`	Indicates the unicast drop mode of the switch. `TRUE` The device drops unicast cells. `FALSE` The device sends ONACK.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_INVALID_DEVICE_NO
```

QseGetParityErrIntMask

Description: The `QseGetParityErrIntMask` function retrieves the parity error interrupt mask for the input port specified.

Invocation:
```
QseGetParityErrIntMask (UINT2 ui2DevId, UINT2 ui2PortNo,
BOOLEAN *bStatus)
```

Inputs:	*Argument*	*Description*
	`ui2DevId`	Identifies a particular QSE device.
	`ui2PortNo`	Port number for which to obtain the interrupt mask.

Outputs:	*Argument*	*Description*
	`pbStatus`	Enables or disables the parity error interrupt.
		`TRUE` Enables the parity interrupt.
		`FALSE` Disables the parity interrupt.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`

QseGetPhaseAligner

Description: The `QseGetPhaseAligner` function retrieves the status of the phase aligner of the switch.

Invocation:
```
QseGetPhaseAligner (UINT2 ui2DevId, ON_OFF* peFlag)
```

Inputs:	*Argument*	*Description*
	`ui2DevId`	Identifies a particular QSE device.

Outputs:	*Argument*	*Description*
	`peFlag`	Indicates if the phase aligner is on or off.
		`TRUE` Indicates the phase aligner is on.
		`FALSE` Indicates the phase aligner is off.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`

QseGetBpDly

Description: The `QseGetBpDly` function sets the backpressure to either early backpressure or optimal backpressure.

Invocation: `QseGetBpDly (UINT2 ui2DevId, BP_MODE* peBpMode)`

Inputs:	*Argument*	*Description*
	`ui2DevId`	Identifies a particular QSE device.

Outputs:	*Argument*	*Description*
	`peBpMode`	Indicates early or optimal backpressure setting.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`

QseGetIntrMask

Description: The `QseGetIntrMask` retrieves the status of the interrupt mask.

Invocation: `QseGetIntrMask (UINT2 ui2DevId, BOOLEAN* pbFlag)`

Inputs:	*Argument*	*Description*
	`ui2DevId`	Identifies a particular QSE device.

Outputs:	*Argument*	*Description*
	`pbMask`	Enables or disables the interrupts. `1` Enables the interrupts. `0` Disables the interrupts.

Returns:
`QSE_SUCCESS`
`QSE_ERR_INVALID_DEVICE_NO`
`QSE_ERR_DEVICE_IN_WRONG_STATE`

QseGetMCAggrMode

Description: The `QseSetMCAggrMode` function retrieves the aggregate mode for multicast traffic.

Invocation:
```
QseGetMCAggrMode (UINT2 ui2DevId, AGGR_IN_MODE* peAggrIn,
AGGR_OUT_MODE* peAggrOut)
```

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.

Outputs:

Argument	*Description*
`peAggrIn`	Indicates the number of consecutive ports treated as a single input.
`peAggrOut`	Indicates the number of consecutive ports treated as a single output.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_INVALID_DEVICE_NO
```

QseGetMaxPendingCells

Description: The `QseGetMaxPendingCells` function retrieves the maximum pending cells in a port.

Invocation:
```
QseGetMaxPendingCells (UINT2 ui2DevId, MAX_CELL_MODE* peMaxMode)
```

Inputs:

Argument	*Description*
`ui2DevId`	Identifies a particular QSE device.

Outputs:

Argument	*Description*
`peMaxMode`	The maximum cells that can be pending. There can be a maximum of either three or four cells.

Returns:
```
QSE_SUCCESS
QSE_ERR_DEVICE_IN_WRONG_STATE
QSE_ERR_INVALID_DEVICE_NO
```

QseGetBpRemoteFailLatch

Description: The `QseGetBpRemoteFailLatch` function returns the error status on all the input ports since the last call of this function.

Invocation: `QseGetBpRemoteFailLatch (UINT2 ui2DevId, UINT4 *pui4Status)`

Inputs:	*Argument*	*Description*
	`ui2DevId`	Identifies a particular QSE device.

Outputs:	*Argument*	*Description*
	`pui4Status`	Returns the backpressure remote fail status of the all the ports since the last call.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`

QseGetBpAckFailLatch

Description: The `QseGetBpAckFailLatch` function returns the error status on all the input ports since the last read of this function.

Invocation: `QseGetBpAckFailLatch (UINT2 ui2DevId,UINT4 *pui4Status)`

Inputs:	*Argument*	*Description*
	`ui2DevId`	Identifies a particular QSE device.

Outputs:	*Argument*	*Description*
	`pui4Status`	Returns the backpressure remote fail status of all the ports since the last call.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`

QseGetInPortFailLatch

Description: The `QseGetInPortFailLatch` function returns the error status on all the input ports since the last call of this function.

Invocation: `QseGetInPortFailLatch (UINT2 ui2DevId,UINT4 *pui4Status)`

Inputs:	*Argument*	*Description*
	`ui2DevId`	Identifies a particular QSE device.

Outputs:	*Argument*	*Description*
	`pui4Status`	Returns the input status failure status of the all the ports since the last call.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`

QseGetInPortFailPresent

Description: The `QseGetInPortFailLatch` function returns the error status of the port specified since the last cell time.

Invocation:
```
QseGetInPortFailPresent (UINT2 ui2DevId, UINT2 ui2PortNo,
UINT4 *pui4Status)
```

Inputs:	*Argument*	*Description*
	`ui2DevId`	Identifies a particular QSE device.
	`ui2PortNo`	Port number for which the status was returned.

Outputs:	*Argument*	*Description*
	`pui4Status`	Returns the input failure status of the port specified since the last cell time.

Returns:
`QSE_SUCCESS`
`QSE_ERR_DEVICE_IN_WRONG_STATE`
`QSE_ERR_INVALID_DEVICE_NO`
`QSE_ERR_INVALID_PORT_NO`

Appendix A
Error Codes

Table 8 lists the error codes used by the QSE Driver. These error codes are declared in the qsedef.h file.

Table 8. QSE Driver Error Codes

Error Code	Description
QSE_ERR_ADDR_ALIAS_TEST	Indicates a word during a memory alias test was read back incorrectly. The memory test failed.
QSE_ERR_DEVICE_ABSENT	Indicates the QSE device cannot be detected.
QSE_ERR_DEVICE_IN_WRONG_STATE	Indicates the QSE device cannot execute the next state machine transaction. The QSE device is in an incorrect state.
QSE_ERR_EXTERNAL_RAM	Indicates an external RAM error.
QSE_ERR_INVALID_COL_NUM	Indicates the column number specified for the QSE device is invalid.
QSE_ERR_INVALID_DEVICE_NO	Indicates the QSE Driver is not compatible with the QSE device's revision number.
QSE_ERR_INVALID_INIT_VECTOR	Indicates the initialization vector validation failed.
QSE_ERR_INVALID_INDEX	Indicates the number of multicast connections allowed is exceeded.
QSE_ERR_INVALID_ROW_NUM	Indicates the row number specified for the QSE device is invalid.
QSE_ERR_MEMORY_ALLOC_FAIL	Indicates the memory allocation function (`alloc`) failed. Sufficient memory is not available for the requested function.
QSE_ERR_MEM_PATTERN_TEST1	Indicates a word during a memory test with the sequence: write, write, write…, read, read, read… was read back incorrectly. The memory test failed.
QSE_ERR_MEM_PATTERN_TEST2	Indicates a word during a memory test with the sequence: write, read, write, read… was read back incorrectly. The memory test failed.
QSE_ERR_READBACK_REG_TEST	Indicates the register values of the device have been read and compared with the initialization vector. The two values are not the same.
QSE_ERR_SCRATCHPAD_FAIL	Indicates the scratchpad memory has errors.
QSE_ERR_SELF_TEST_PREP_FAIL	Indicates the self-test failed.
QSE_WRN_ALREADY_SAME_VALUE	Indicates that an attempt is being made to set the device to the same state as before.

CONTACTING PMC-SIERRA, INC.

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

Tel: (604) 415-6000

Fax: (604) 415-6200

Document Information:	document@pmc-sierra.com
Corporate Information:	info@pmc-sierra.com
Application Information:	apps@pmc-sierra.com
	(604) 415-4533
Web Site:	http://www.pmc-sierra.com

PMC-980876 (P1) Issue date: November 1998