PM5313

SPECTRA-622

SONET/SDH PAYLOAD EXTRACTOR/ALIGNER FOR 622 MBIT/S

DRIVER MANUAL

DOCUMENT ISSUE 2

ISSUED NOVEMBER, 2000

ABOUT THIS MANUAL AND SPECTRA-622

This manual describes the SPECTRA-622 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 the devices. It also describes in general terms how to modify and port the driver to your software and hardware platform.

Audience

This manual was written for people who need to:

- Evaluate and test the SPECTRA-622 devices
- Modify and add to the SPECTRA-622 driver's functions
- Port the SPECTRA-622 driver to a particular platform.

References

For more information about the SPECTRA-622 driver, see the driver's release notes. For more information about the SPECTRA-622 device, see the documents listed in Table 1 and any related errata documents.

Table 1: Related Documents

Document Name	**Document Number**
SPECTRA-622 Telecom Standard Product Data Sheet	PMC-1981162
PM5313 SPECTRA-622 SONET/SDH Payload Extractor/Aligner for 622 Mbit/s Interfaces Short Form Data Sheet	PMC-1981271

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

Revision History

Issue No.	Issue Date	Details of Change
Issue 1	December 1999	Document created
Issue 2	November 2000	1) Modified the alarm, status and statistics architecture (structures and APIs): a) removed MSB and DSB structures as well as `spectraClearStats()` API since statistics are no longer accumulated inside the driver. b) Added `SPE_STATUS_XX` and `SPE_CNT_XX` structures to add granularity. c) replaced `spectraGetStats()` API with `spectraGetCntXX()` and `spectraGetStatusXX()` APIs. 2) Modified “normal mode” initialization profile in section 4.2: a) replaced `serialMode`, `stm1Mode` and `ds3Mode` fields with 3 new fields: `lineSideMode`, `sysSideMode` and `clock77` to allow for a better initialization of the IO interface. b) replaced `master[4][3]` field with `sts12c` and `sts3c[4]` for easier configuration of concatenated payloads. 3) Added `spectraTOCReadS1` to read the received S1 byte. 4) Removed `spectraRPPSDiagPJ` and `spectraTPPSDiagPJ` APIs since the feature is not available in hardware. 5) Fixed incorrect descriptions throughout the document: a) added missing `cfgCnt` field in DDB structure. b) added missing `tppsIllreq[4][3]` in ISR mask structure. c) removed `rppsCdiff[4][3]` and `tppsBlkBip[4][3]` from `CFG_CNT`. d) added missing au3 parameter description in `spectraDPGMGenRegen`. e) fixed function table description of `spectraISR`. f) valid states for `spectraDiagTestReg` now show as PRESENT only. 6) Fixed various typos and formatting issues.

Legal Issues

PMC-1991254 (R2), ref 990876 (R3)

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

LIST OF FIGURES

LIST OF TABLES

1 DRIVER PORTING QUICK START

This section summarizes how to port the SPECTRA-622 device driver to your hardware and operating system (OS) platform. For more information about porting the SPECTRA-622 driver, see section 8 (page 126).

Note: Because each platform and application is unique, this manual can only offer guidelines for porting the SPECTRA-622 driver.

The code for the SPECTRA-622 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 SPECTRA-622 driver to your platform:

Step 1: Port the driver's RTOS interface (page 127):

- Data types
- OS-specific services
- Utilities and interrupt services that use OS-specific services

Step 2: Port the driver's hardware interface (page 129)

- Port low-level device read-and-write macros.
- Define hardware system-configuration constants.

Step 3: Port the driver's application-specific elements (page 130):

- Define the task-related constants.
- Code the callback functions.

Step 4: Build the driver (page 130).

2 DRIVER FUNCTIONS AND FEATURES

This section describes the main functions and features supported by the SPECTRA-622 driver.

Table 1: Driver Functions and Features

Function	Description
Open / Close Driver Module (page 56)	Opening the Driver Module allocates all the memory needed by the driver and initializes all Module level data structures. Closing the Driver Module shuts down the driver module gracefully after deleting all devices that are currently registered with the driver, and releases all the memory allocated by the driver.
Start / Stop Driver Module (page 57)	Starting the Driver Module involves allocating all RTOS resources needed by the driver such as timers and semaphores (except for memory, which is allocated during the Open call). Closing the Driver Module involves de-allocating all RTOS resources allocated by the driver without changing the amount of memory allocated to it.
Add / Delete Device (page 61)	Adding a device involves verifying that the device exists, associating a device Handle to the device, and storing context information about it. The driver uses this context information to control and monitor the device. Deleting a device involves shutting down the device and clearing the memory used for storing context information about this device.
Device Initialization (page 63)	The initialization function resets then initializes the device and any associated context information about it. The driver uses this context information to control and monitor the SPECTRA-622 device.
Activate / De-Activate Device (page 64)	Activating a device puts it into its normal mode of operation by enabling interrupts and other global registers. A successful device activation also enables other API invocations. On the contrary, de-activating a device removes it from its operating state, disables interrupts and other global registers.

Read / Write Device Registers (page 66)	These functions provide a ‘raw’ interface to the device. Device registers that are both directly and indirectly accessible are available for both inspection and modification via these functions. If applicable, block reads and writes are also available.
Interrupt Servicing / Polling (page 93)	Interrupt Servicing is an optional feature. The user can disable device interrupts and instead poll the device periodically to monitor status and check for alarm/error conditions. Both polling and interrupt driven approaches detect a change in device status and report the status to a Deferred Processing Routine (DPR). The DPR then invokes application callback functions based on the status information retrieved. This allows the driver to report significant events that occur within the device to the application.
Statistics Collection (page 97)	Functions are provided to retrieve a snapshot of the various counts that are accumulated by the SPECTRA-622 device. Routines should be invoked often enough to avoid letting the counters to rollover.

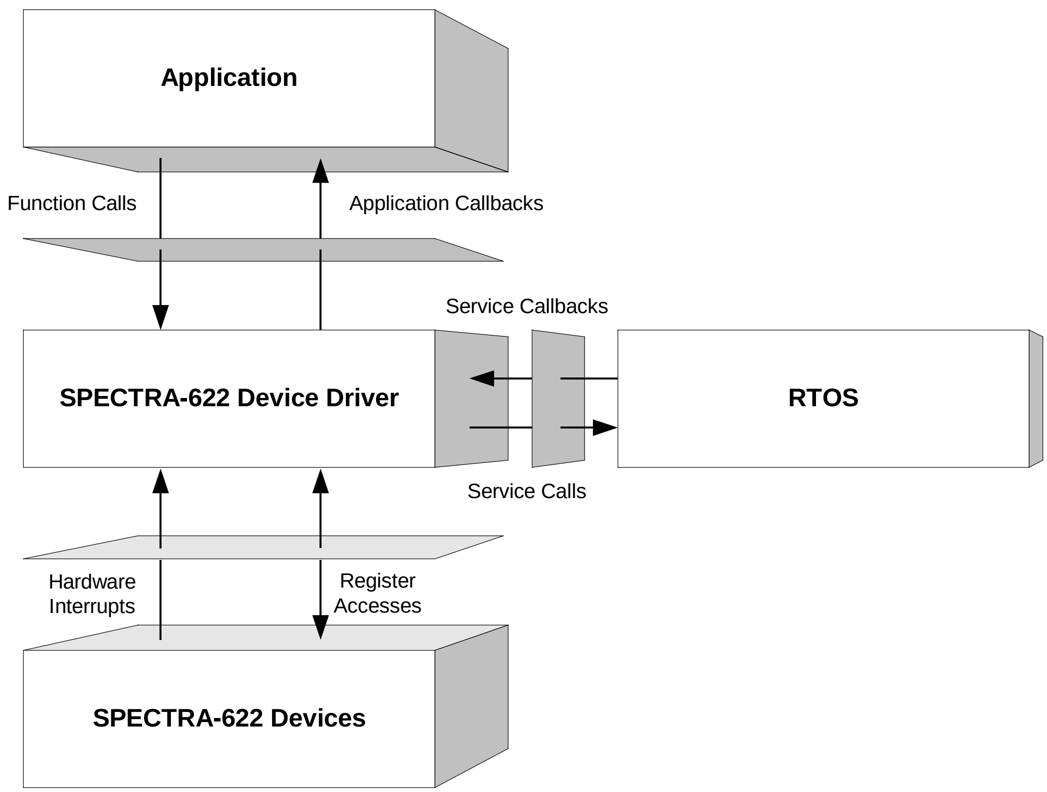
3 SOFTWARE ARCHITECTURE

This section describes the software architecture of the SPECTRA-622 device driver. This includes a discussion of the driver's external interfaces and its main components.

3.1 Driver External Interfaces

Figure 1 illustrates the external interfaces defined for the SPECTRA-622 device driver.

Figure 1: Driver Interfaces



Application Programming Interface

The driver's API is a collection of high level functions that can be called by application programmers to configure, control, and monitor the SPECTRA-622 device, such as:

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

- Diagnosing the device

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

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

Real-Time OS Interface

The driver's RTOS interface module provides functions that let the driver use RTOS services. The SPECTRA-622 driver requires the memory, interrupt, and preemption services from the RTOS. The RTOS interface functions perform the following tasks for the SPECTRA-622 device and driver:

- Allocate and de-allocate memory
- Manage buffers for the ISR and DPR
- Disable and enable preemption

The RTOS interface also includes service callbacks. These are functions installed by the driver using RTOS service calls, such as installing the ISR handler and the DPR task. These service callbacks are invoked when an interrupt occurs or the DPR is scheduled.

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

Driver Hardware Interface

The SPECTRA-622 hardware interface provides functions that read from and write to device-registers. The hardware interface also provides a template for an ISR that the driver calls when the device raises a hardware interrupt. You must modify this function based on the interrupt configuration of your system.

3.2 Main Components

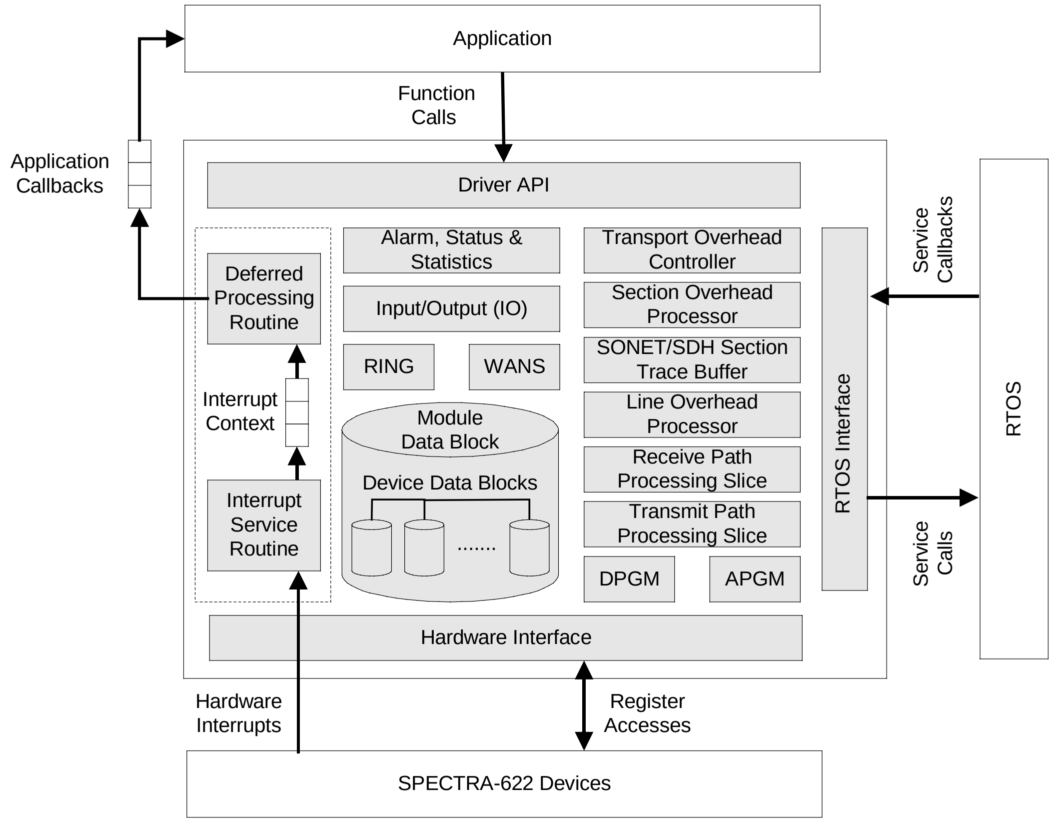
Figure 2 illustrates the top-level architectural components of the SPECTRA-622 device driver. This applies in both polled and interrupt driven operation. In polled operation the ISR is called periodically. In interrupt operation the interrupt directly triggers the ISR.

The driver includes the following main components:

- Module and Device(s) Data-Blocks
- Interrupt-Processing Routine
- Deferred-Processing Routine

- Alarm, Status and Statistics
- Input/Output
- Transport Overhead Controller
- Section Overhead Processor
- SONET/SDH Section Trace Buffer
- Line Overhead Processor
- Receive Path Processing Slice
- Transmit Path Processing Slice
- Ring Control Ports
- WAN Synchronization Controller
- DROP Bus PRBS Generator and Monitor
- ADD Bus PRBS Generator and Monitor

Figure 2: Driver Architecture



Alarms, Status and Statistics

The Alarms, Status and Statistics is responsible for monitoring alarms, tracking devices status information and retrieving statistical counts for each device registered with (added to) the driver.

Input / Output (IO)

The Input / Output section is responsible for configuring the line-side and system-side device interfaces. On the line-side, functions are provided to control the 622.08 Mbps clock/data interface. On the system-side, in Telecom Bus mode functions are provided to control the Add/Drop Telecom Bus data interfaces and the Time Slot Interchange (TSI). In DS3 mode, functions are provided to control the DS3 data interface.

Transport Overhead Controller (TOC)

The Transport Overhead Controller is responsible for configuring the transport overhead processing on both receive and transmit sides. Functions are provided to directly write the Z0 and S1 bytes.

Receive / Transmit Section Overhead Processor (RSOP/TSOP)

The Receive / Transmit Section Overhead Processor is responsible for configuring and monitoring the processing of the section overhead on both receive and transmit sides. Functions are provided to monitor the received section overhead, to enable/disable Line AIS insertion and to enable/disable insertion of section errors for diagnostics.

SONET / SDH Section Trace Buffer (SSTB)

The SONET / SDH Section Trace Buffer is responsible for configuring and monitoring the section trace message (J0). Functions are provided to monitor the received section trace message and set the transmit section message,

Receive / Transmit Line Overhead Processor (RLOP/TLOP)

The Receive / Transmit Line Overhead Processor is responsible for configuring and monitoring the processing of the line overhead on both receive and transmit sides. Functions are provided to monitor the received line overhead, to configure and monitor the RASE (Receive APS Synchronization Extractor), and enable/disable the insertion of line errors for diagnostics. Functions are provided to directly read/write the K1 and K2 bytes.

Receive Path Processing Slice (RPPS)

The Receive Path Processing Slice functions are provided to configure and monitor the RTAL (Receive Telecombus Aligner) and tandem connection, to monitor the received path overhead and path trace message (J1), and to configure the DS3 mapper (D3MD) in DS3 mode.

Transmit Path Processing Slice (TPPS)

The Transmit Path Processing Slice functions are provided to configure and monitor the TTAL (Transmit Telecombus Aligner) and tandem connection, to configure the path overhead, to enable/disable the insertion of path overhead (J1) errors for diagnostics, and to configure the DS3 mapper (D3MA) in DS3 mode. Functions are provided to directly write the J1, C2, F2, Z3, Z4 and Z5 bytes.

Ring Control Ports (RING)

Ring Control Ports functions are provided to enable/disable the generation of the rx/tx ring control port signals.

WAN Synchronization Controller (WANS)

The WAN Synchronization Controller functions are provided to enable/disable the generation of the WAN synchronization signals.

DROP Bus PRBS Generator and Monitor (DPGM)

The DROP Bus PRBS Generator and Monitor functions are provided to enable / disable the insertion of a pseudo random byte sequence inside the payload.

ADD Bus PRBS Generator and Monitor (APGM)

The ADD bus PRBS Generator and Monitor functions are provided to enable / disable the insertion of a pseudo random byte sequence inside the payload.

Module Data Block (MDB)

The Module Data Block (MDB) is the top-layer data structure, created by the SPECTRA-622 device driver to keep track of its initialization and operating parameters, modes and dynamic data. The MDB is allocated via an RTOS call, when the driver module is opened and contains all the device structures

Device Data Blocks (DDB)

The Device Data Blocks (DDB) are contained in the MDB and they are allocated when the module is opened. They are initialized by the SPECTRA-622 device driver for each device that is registered, to keep track of that device's initialization and operating parameters, modes and dynamic data. There is a limit on the number of devices that can be registered with the driver module. This number is set when the driver module is opened.

Interrupt Service Routine

The SPECTRA-622 driver provides an ISR called `spectraISR` that checks if there are any valid interrupt conditions present for the device. This function can be used by a system-specific interrupt-handler function to service interrupts raised by the device.

The low-level interrupt-handler function that traps the hardware interrupt and calls `spectraISR` is system and RTOS dependent. Therefore, it is outside the scope of the driver. Example implementations of an interrupt handler and functions that install and remove it are provided as a reference on page 117. You can customize these example implementations to suit your specific needs.

Deferred Processing Routine

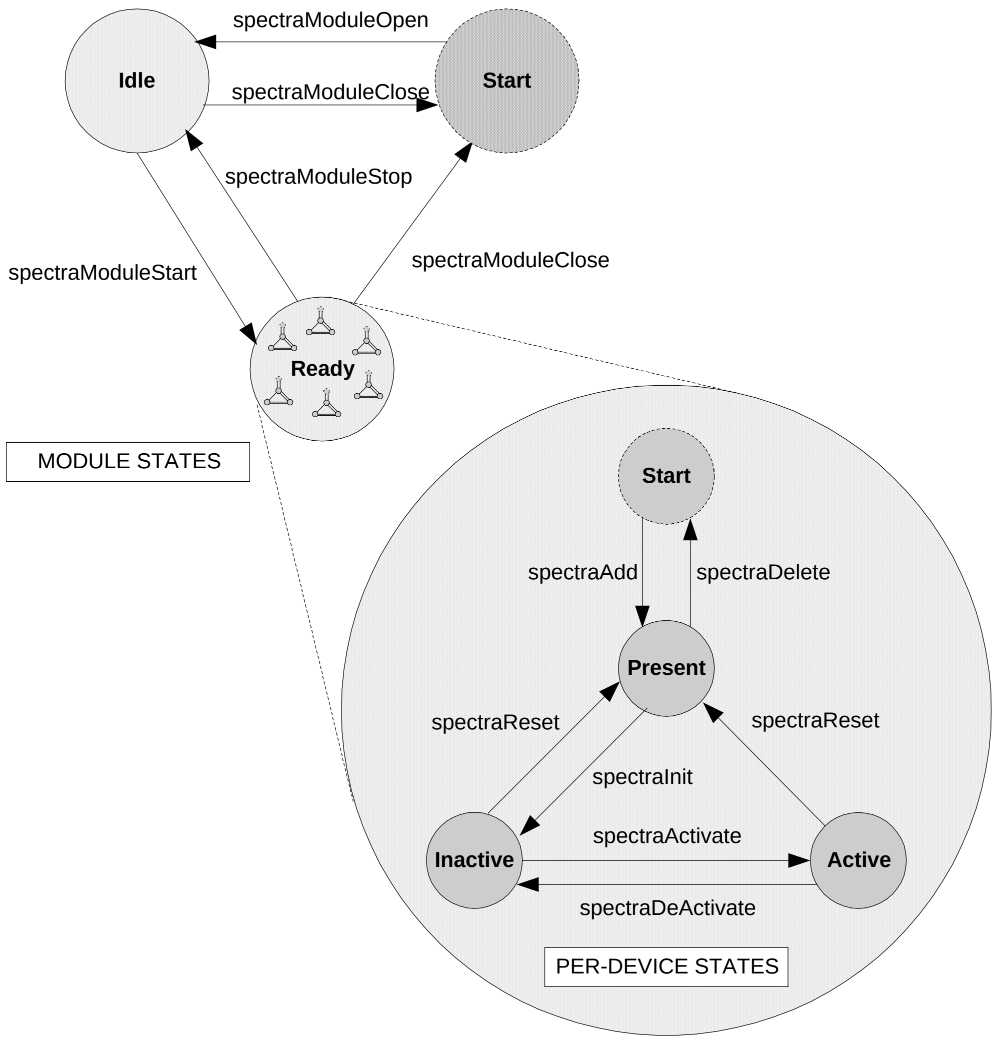
The DPR provided by the SPECTRA-622 driver (`spectraDPR`) clears and processes interrupt conditions for the device. Typically, a system specific function, which runs as a separate task within the RTOS, executes the DPR.

See page 26 for a detailed explanation of the DPR and interrupt-servicing model.

3.3 Software States

Figure 3 shows the software state diagram for the SPECTRA-622 driver. State transitions occur on the successful execution of the corresponding transition functions shown. State information helps maintain the integrity of the MDB and DDB(s) by controlling the set of operations allowed in each state.

Figure 3: Driver Software States



Module States

The following is a description of the SPECTRA-622 module states. See sections 5.1 and 5.2 for a detailed description of the API functions that are used to change the module state.

Start

The driver Module has not been initialized. In this state the driver does not hold any RTOS resources (memory, timers, etc); has no running tasks, and performs no actions.

Idle

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

Ready

This is the normal operating state for the driver Module. This means that all RTOS resources have been allocated and the driver is ready for Devices to be added. The driver Module remains in this state while Devices are in operation.

Device States

The following is a description of the SPECTRA-622 per-device states. The state that is mentioned here is the software state as maintained by the driver, and not as maintained inside the device itself. See sections 5.4, 5.5 and 5.6 for a detailed description of the API functions that are used to change the per-device state.

Start

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

Present

The Device has been successfully added. A Device Data Block (DDB) has been associated to the Device and updated with the user context, and a device handle has been given to the USER. In this state, the device performs no actions.

Inactive

In this state the Device is configured but all data functions are de-activated including interrupts and alarms, status and statistics functions.

Active

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

3.4 Processing Flows

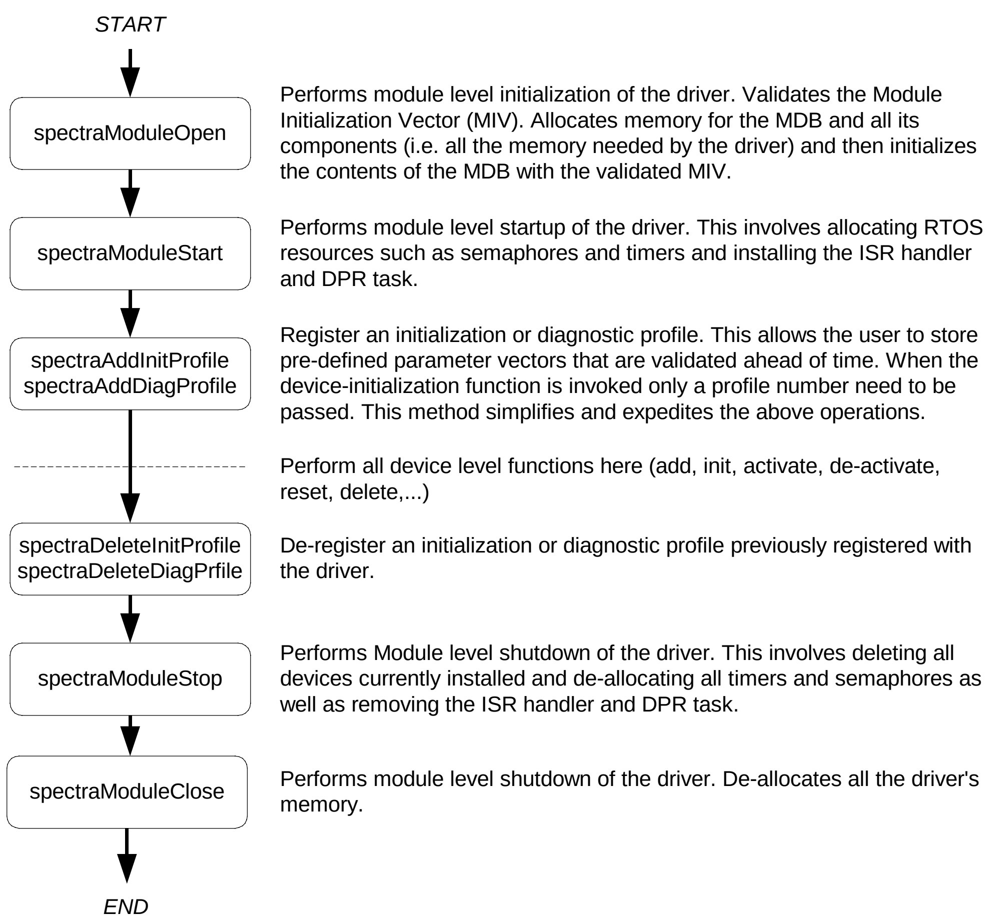
This section describes the main processing flows of the SPECTRA-622 driver modules.

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

Module Management

The following diagram illustrates the typical function call sequences that occur when initializing or shutting down the SPECTRA-622 driver module.

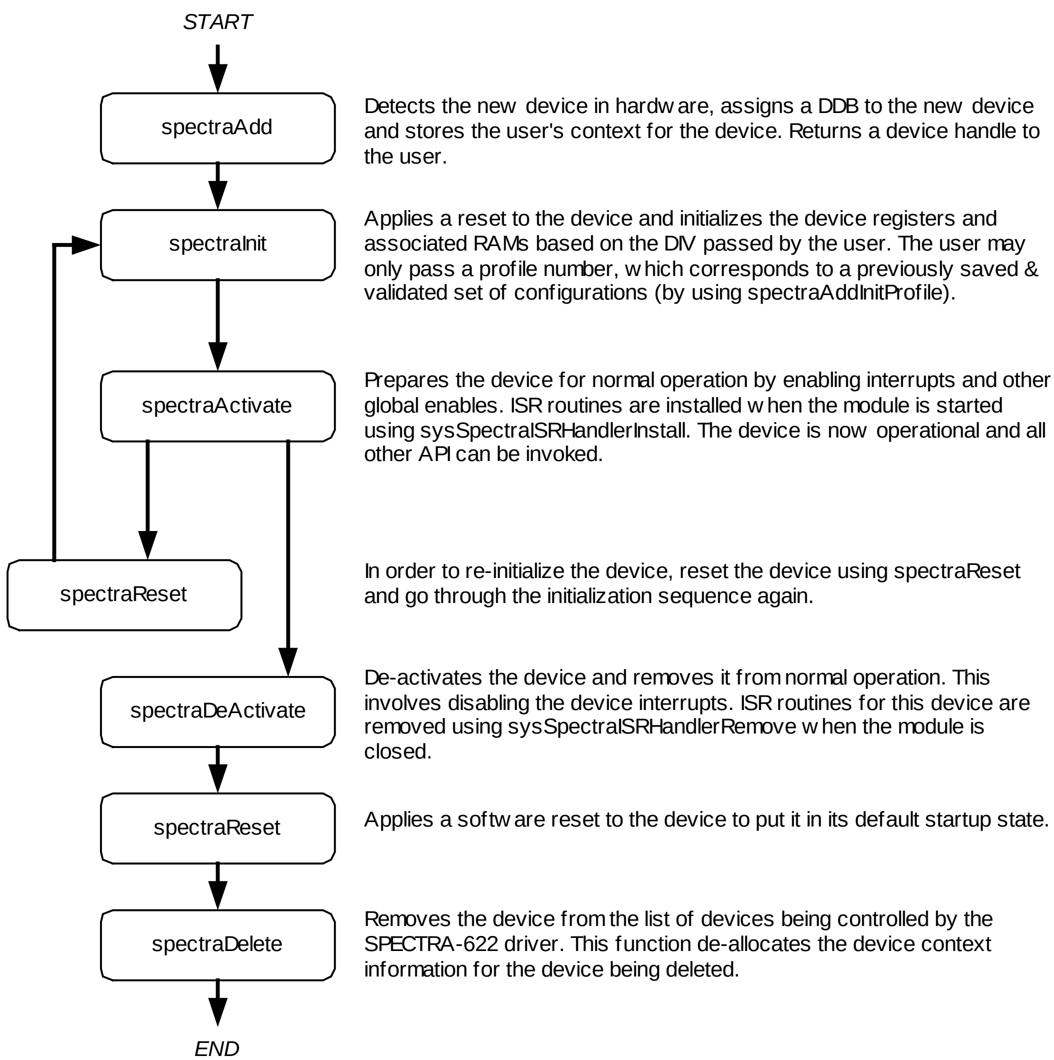
Figure 4: Module Management Flow Diagram



Device Management

The following diagram shows the functions and process that the driver uses to add, initialize, re-initialize, and delete the SPECTRA-622 device.

Figure 5: Device Management Flow Diagram



3.5 Interrupt Servicing

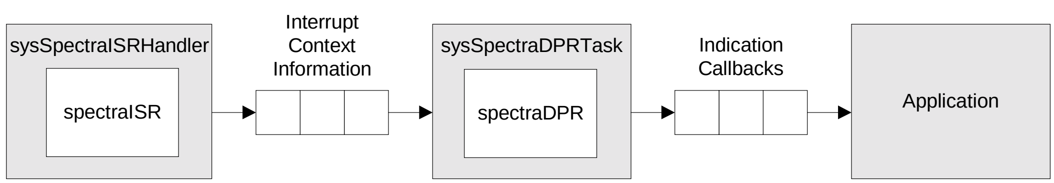
The SPECTRA-622 driver services device interrupts using an interrupt service routine (ISR) that traps interrupts and a deferred processing routine (DPR) that actually processes the interrupt conditions and clears them. This lets the ISR execute quickly and exit. Most of the time-consuming processing of the interrupt conditions is deferred to the DPR by queuing the necessary interrupt-context information to the DPR task. The DPR function runs in the context of a separate task within the RTOS.

Note: Since the DPR task processes potentially serious interrupt conditions, you should set the DPR task's priority higher than the application task interacting with the SPECTRA-622 driver.

The driver provides the system-independent functions, `spectraISR` and `spectraDPR`. You must fill in the corresponding system-specific functions, `sysSpectraISRHandler` and `sysSpectraDPRTask`. The system-specific functions isolate the system-specific communication mechanism (between the ISR and DPR) from the system-independent functions, `spectraISR` and `spectraDPR`.

Figure 6 illustrates the interrupt service model used in the SPECTRA-622 driver design.

Figure 6: Interrupt Service Model



Note: Instead of using an interrupt service model, you can use a polling service model in the SPECTRA-622 driver to process the device's event-indication registers (see page 28).

Calling spectraISR

An interrupt handler function, which is system dependent, must call `spectraISR`. But first, the low-level interrupt-handler function must trap the device interrupts. You must implement this function to fit your own system. As a reference, an example implementation of the interrupt handler (`sysSpectraISRHandler`) appears on page 118. You can customize this example implementation to suit your needs.

The interrupt handler that you implement (`sysSpectraISRHandler`) is installed in the interrupt vector table of the system processor. It is called when one or more SPECTRA-622 devices interrupt the processor. The interrupt handler then calls `spectraISR` for each device in the active state that has interrupt processing enabled.

The `spectraISR` function reads from the master interrupt-status registers and the miscellaneous interrupt-status registers of the SPECTRA-622. If at least one valid interrupt condition is found then `spectraISR` fills an Interrupt Service Vector (ISV) with this status information as well as the current device Handle. The `spectraISR` function also clears and disables all the device's interrupts detected. The `sysSpectraISRHandler` function is then responsible to send this ISV buffer to the DPR task.

Note: Normally you should save the status information for deferred interrupt processing by implementing a message queue.

Calling spectraDPR

The `sysSpectraDPRTask` function is a system specific function that runs as a separate task within the RTOS. You should set the DPR task's priority higher than the application task(s) interacting with the SPECTRA-622 driver. In the message-queue implementation model, this task has an associated message queue. The task waits for messages from the ISR on this message queue. When a message arrives, `sysSpectraDPRTask` calls the DPR (`spectraDPR`) with the received ISV.

Then `spectraDPR` processes the status information and takes appropriate action based on the specific interrupt condition detected. The nature of this processing can differ from system to system. Therefore, `spectraDPR` calls different indication callbacks for different interrupt conditions.

Typically, you should implement these callback functions as simple message posting functions that post messages to an application task. However, you can implement the indication callback to perform processing within the DPR task context and return without sending any messages. In this case, ensure that the indication function does not call any API functions that change the driver's state, such as `spectraDelete`. Also, ensure that the indication function is non-blocking because the DPR task executes while SPECTRA-622 interrupts are disabled. You can customize these callbacks to suit your system. See page 109 for example implementations of the callback functions.

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

You must implement the two system specific functions, `sysSpectraISRHandler` and `sysSpectraDPRTask`. When the driver calls `sysSpectraISRHandlerInstall`, the application installs `sysSpectraISRHandler` in the interrupt vector table of the processor. The `sysSpectraDPRTask` function is spawned as a task by the application. The `sysSpectraISRHandler Install` function also creates the communication channel between `sysSpectraISRHandler` and `sysSpectraDPRTask`. This communication channel is most commonly a message queue associated with the `sysSpectraDPRTask`.

Similarly, during removal of interrupts, the driver removes `sysSpectraISRHandler` from the microprocessor’s interrupt vector table and deletes the task associated with `sysSpectraDPRTask`.

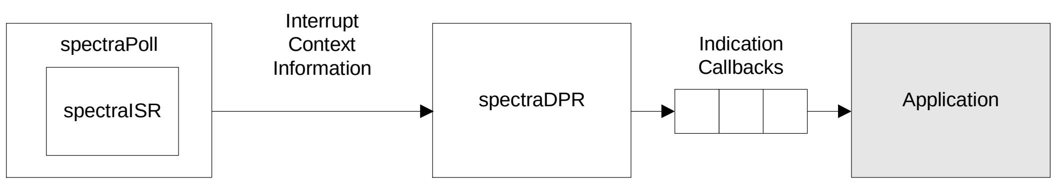
As a reference, this manual provides example implementations of the interrupt installation and removal functions on page 117. You can customize these prototypes to suit your specific needs.

Calling spectraPoll

Instead of using an interrupt service model, you can use a polling service model in the SPECTRA-622 driver to process the device’s event-indication registers.

Figure 7 illustrates the polling service model used in the SPECTRA-622 driver design.

Figure 7: Polling Service Model



In polling mode, the application is responsible for calling `spectraPoll` often enough to service any pending error or alarm conditions. When `spectraPoll` is called, the `spectraISR` function is called internally.

The `spectraISR` function reads from the master interrupt-status registers and the miscellaneous interrupt-status registers of the SPECTRA-622. If at least one valid interrupt condition is found then `spectraISR` fills an Interrupt Service Vector (ISV) with this status information as well as the current device Handle. The `spectraISR` function also clears and disables all the device’s interrupts detected. In polling mode, this ISV buffer is passed to the DPR task by calling `spectraDPR` internally.

4 DATA STRUCTURES

4.1 Constants

The following Constants are used throughout the driver code:

- `<SPECTRA-622 ERROR CODES>` error codes used throughout the driver code, returned by the API functions and used in the global error number field of the MDB and DDB. See Appendix A on page 131.
- `SPE_MAX_DEVS` defines the maximum number of devices that can be supported by this driver. This constant must not be changed without a thorough analysis of the consequences to the driver code.
- `SPE_MOD_START`, `SPE_MOD_IDLE`, `SPE_MOD_READY` are the three possible Module states (stored in `stateModule`).
- `SPE_START`, `SPE_PRESENT`, `SPE_ACTIVE`, `SPE_INACTIVE` are the four possible Device states (stored in `stateDevice`).

4.2 Structures Passed by the Application

These structures are defined for use by the application and are passed as argument to functions within the driver. These structures are the Module Initialization Vector (MIV), the Device Initialization Vector (DIV) and the ISR mask.

Module Initialization Vector: MIV

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

- `maxDevs` is used to inform the Driver how many Devices will be operating concurrently during this session. The number is used to calculate the amount of memory that will be allocated to the driver. The maximum value that can be passed is `SPE_MAX_DEVS`.

Table 2: Module Initialization Vector: sSPE_MIV

Field Name	Field Type	Field Description
pmdb	sSPE_MDB *	(pointer to) MDB
maxDevs	UINT2	Maximum number of devices supported during this session
maxInitProfs	UINT2	Maximum number of initialization profiles
maxDiagProfs	UINT2	Maximum number of diagnostic profiles

Device Initialization Vector: DIV

Passed via the `spectraInit` call, this structure contains all the information needed by the driver to initialize a SPECTRA-622 device. This structure is also passed via the `spectraAddInitProfile` call when used as an initialization profile.

- `valid` indicates that this initialization profile has been properly initialized and may be used by the USER. This field should be ignored when the DIV is passed directly.
- `pollISR` is a flag that indicates the type of interrupt servicing the driver is to use. The choices are ‘polling’ (`SPE_POLL_MODE`), and ‘interrupt driven’ (`SPE_ISR_MODE`). When configured in polling the Interrupt capability of the Device is NOT used, and the USER is responsible for calling `devicePoll` periodically. The actual processing of the event information is the same for both modes.
- `cbackIO`, `cbackTOC`, `cbackSOP`, `cbackSSTB`, `cbackLOP`, `cbackRPPS`, `cbackTPPS`, `cbackWANS`, `cbackDPGM` and `cbackAPGM` are used to pass the address of application functions that will be used by the DPR to inform the application code of pending events. If these fields are set as NULL, then any events that might cause the DPR to ‘call back’ the application will be processed during ISR processing but ignored by the DPR.

Table 3: Device Initialization Vector: sSPE_DIV

Field Name	Field Type	Field Description
valid	UINT2	Indicates that this profile is valid
initMode	SPE_MODE	Mode used for Initialization: `SPE_NORM`, `SPE_COMP` or `SPE_FRM`

Field Name	Field Type	Field Description
pinitData	UINT1*	(pointer to) initialization data. Depending on the specified mode of initialization, this is in fact a pointer to sSPE_INIT_DATA_NORM, sSPE_INIT_DATA_COMP or sSPE_INIT_DATA_FRM.
pollISR	sSPE_POLL	Indicates the type of ISR / polling to do
cbackIO	sSPE_CBACK	Address for the callback function for IO Events
cbackTOC	sSPE_CBACK	Address for the callback function for TOC Events
cbackSOP	sSPE_CBACK	Address for the callback function for SOP Events
cbackSSTB	sSPE_CBACK	Address for the callback function for SSTB Events
cbackLOP	sSPE_CBACK	Address for the callback function for LOP Events
cbackRPPS	sSPE_CBACK	Address for the callback function for RPPS Events
cbackTPPS	sSPE_CBACK	Address for the callback function for TPPS Events
cbackWANS	sSPE_CBACK	Address for the callback function for WANS Events
cbackDPGM	sSPE_CBACK	Address for the callback function for DPGM Events
cbackAPGM	sSPE_CBACK	Address for the callback function for APGM Events

Initialization Profile: INIT_PROF

Initialization Profile Top-Level Structure

Passed via the spectraAddInitProfile call, this structure contains all the information needed by the driver to initialize and activate a SPECTRA-622 device. This is in fact the same structure as sSPE_DIV.

Table 4: Initialization Profile: sSPE_INIT_PROF

Field Name	Field Type	Field Description
valid	UINT2	Indicates that this profile is valid
initMode	SPE_MODE	Mode used for Initialization: SPE_NORM, SPE_COMP or SPE_FRM
pinitData	UINT1*	(pointer to) initialization data. Depending on the specified mode of initialization, this is in fact a pointer to sSPE_INIT_DATA_NORM, sSPE_INIT_DATA_COMP or sSPE_INIT_DATA_FRM.
pollISR	sSPE_POLL	Indicates the type of ISR / polling to do
cbackIO	sSPE_CBACK	Address for the callback function for IO Events
cbackTOC	sSPE_CBACK	Address for the callback function for TOC Events
cbackSOP	sSPE_CBACK	Address for the callback function for SOP Events
cbackSSTB	sSPE_CBACK	Address for the callback function for SSTB Events
cbackLOP	sSPE_CBACK	Address for the callback function for LOP Events
cbackRPPS	sSPE_CBACK	Address for the callback function for RPPS Events
cbackTPPS	sSPE_CBACK	Address for the callback function for TPPS Events
cbackWANS	sSPE_CBACK	Address for the callback function for WANS Events
cbackDPGM	sSPE_CBACK	Address for the callback function for DPGM Events
cbackAPGM	sSPE_CBACK	Address for the callback function for APGM Events

Initialization Data in Normal Mode (SPE_NORM)

In Normal mode (NORM), the user only specifies the main modes of operation of the device. Most of the device's register bits are left in their default state (after a software reset). This structure is pointed to by `pinitData` inside the initialization profile.

Table 5: Initialization Data: sSPE_INIT_DATA_NORM

Field Name	Field Type	Field Description
lineSideMode	UINT2	selects between serial mode, parallel mode, dual mode with serial input, and dual mode with parallel input on the line side
sysSideMode	UINT2	selects between mode selected via DMODE[1:0] inputs, telecom mode, ds3 mode, dual mode w/ telecom bus input, and dual mode w/ds3 input.
clock77	UINT2	selects between stm4 and stm1 telecom bus mode on the system side
sts12c	UINT2	selects the master/slave slices for sts-12/12c mode
sts3c[4]	UINT2	selects the master/slave slices for sts-3/3c mode
ringEna	UINT2	enables the ring control ports
wansEna	UINT2	enables the phase comparison in the wan synchronization controller

Initialization Data in Compatibility Mode (SPE_COMP)

In Compatibility mode (COMP), the user provides a list of data blocks to write directly to the device registers. There are `numBlocks` blocks provided by the USER. The block number [`i`] is fully defined by:

- `ppblock`[i], which points to the data to write to the device's registers
- `ppmask`[i], which points to a data mask to specify which bits are to be modified
- `psize`[i], the block size
- `pstartReg`[i], which is the register number at which the driver should start writing the data.

This structure is pointed to by `pinitData` inside the initialization profile.

Table 6: Initialization Data: sSPE_INIT_DATA_COMP

Field Name	Field Type	Field Description
numBlocks	UINT2	Number of provided blocks
ppblk[]	UINT1*	(pointer to) an array of pointer to a data block
ppmask[]	UINT1*	(pointer to) an array of pointer to a mask
pblkSize[]	UINT2	(pointer to) an array of block size
pstartReg[]	UINT2	array of register numbers

Initialization Data in Flat Register Mode (SPE_FRM)

In Flat Register Mode (FRM), for each of the hardware blocks (IO, TOC, SOP, SSTB, LOP, RPPS, TPPS, RING and WANS), the user needs to fill a structure that holds a mapping of all the configuration bits for this hardware block. They are used to fully configure the SPECRTA-622 device. This structure is pointed to by `pinitData` inside the initialization profile. The reader is referred to the code for the definitions of the configuration blocks (`sSPE_CFG_XXX`).

Table 7: Initialization Data: sSPE_INIT_DATA_FRM

Field Name	Field Type	Field Description
cfgIO	sSPE_CFG_IO	Input / Output (IO) configuration block
cfgTOC	sSPE_CFG_TOC	Receive / Transmit Transport Overhead Controller (TOC) configuration block
cfgSOP	sSPE_CFG_SOP	Receive / Transmit Section Overhead Processor (RSOP/TSOP) configuration block
cfgSSTB	sSPE_CFG_SSTB	Sonet/SDH Section Trace Buffer (SSTB) configuration block
cfgLOP	sSPE_CFG_LOP	Receive / Transmit Line Overhead Processor (RLOP/TLOP) configuration block
cfgRPPS[4][3]	sSPE_CFG_RPPS	Receive Path Processing Slice (RPPS) configuration block
cfgTPPS[4][3]	sSPE_CFG_TPPS	Transmit Path Processing Slice (TPPS) configuration block

Field Name	Field Type	Field Description
cfgRING	sSPE_CFG_RING	Ring Control Port (RING) configuration block
cfgWANS	sSPE_CFG_WANS	WAN Synchronization controller (WANS) configuration block

Diagnostic Profile: DIAG_PROF

Diagnostic Profile Top-Level Structure

Passed via the `spectraAddDiagProfile` call, this structure contains all the information needed by the driver to initiate a specific diagnostic on the SPECTRA-622 device.

- `diagMode` is a flag that tells the Driver which diagnostic mode is used to configure the device. There are three different ways to configure a device for diagnostics, each corresponding to a different mode:
  - Normal Mode (`SPE_NORM`): the user only specifies the main modes of operation of the DPGM and APGM. Most of the device's register bits are left in their default state (after a software reset).
  - Compatibility mode (`SPE_COMP`): the user provides a list of data blocks to write directly to the APGM and DPGM registers.
  - Flat Register Mode (`SPE_FRM`): for each of the 12 DPGM and APGM hardware blocks, the user needs to fill a structure (`sSPE_CFG_DPGM` and `sSPE_CFG_APGM`) that holds a mapping of all the configuration bits for this hardware block.

Table 8: Diagnostic Profile: sSPE_DIAG_PROF

Field Name	Field Type	Field Description
valid	UINT2	Indicates that this profile is valid
diagMode	SPE_MODE	Mode used for diagnostic: `SPE_NORM`, `SPE_COMP` or `SPE_FRM`
pdiagData	UINT1*	(pointer to) diagnostic data. Depending on the specified mode of diagnostic, this is in fact a pointer to `sSPE_INIT_DATA_NORM`, `sSPE_INIT_DATA_COMP` or `sSPE_INIT_DATA_FRM`.

Diagnostic Data in Normal Mode: SPE_NORM

In Normal mode (NORM), the user only specifies the main modes of operation of the DPGM and APGM. Most of the register bits are left in their default state (after a software reset). This structure is pointed to by `pdiagData` inside the diagnostic profile.

Table 9: Diagnostic Data: sSPE_DIAG_DATA_NORM

Field Name	Field Type	Field Description
dpgmGenEna[4][3]	UINT1	Enables the Generator of the DROP Bus PRBS Generator and Monitor (DPGM)
dpgmMonEna[4][3]	UINT1	Enables the Monitor of the DROP Bus PRBS Generator and Monitor (DPGM)
apgmGenEna[4][3]	UINT1	Enables the Generator of the ADD Bus PRBS Generator and Monitor (APGM)
apgmMonEna[4][3]	UINT1	Enables the Monitor of the ADD Bus PRBS Generator and Monitor (APGM)

Diagnostic Data in Compatibility: Mode SPE_COMP

In Compatibility mode (COMP), the user provides a list of data blocks to write directly to the DPGM and APGM registers. There are `numBlocks` blocks provided by the USER. The block number `[i]` is fully defined by:

- `ppblock[i]`, which points to the data to write to the device's registers
- `ppmask[i]`, which points to a data mask to specify which bits are to be modified
- `psize[i]`, the block size
- `pstartReg[i]`, which is the register number at which the driver should start writing the data.

This structure is pointed to by `pdiagData` inside the diagnostic profile.

Table 10: Diagnostic Data: sSPE_DIAG_DATA_COMP

Field Name	Field Type	Field Description
numBlocks	UINT2	number of provided blocks
ppblk[]	UINT1*	array of pointer to a data block
ppmask[]	UINT1*	array of pointer to a mask
pblkSize[]	UINT2	array of block size

Field Name	Field Type	Field Description
pstartReg[]	UINT2	array of register numbers

Diagnostic Data in FRM Mode: SPE_FRM

In Flat Register Mode (FRM), for each of the 12 DPGM and APGM hardware blocks, the user needs to fill a structure that holds a mapping of all the configuration bits for this hardware block. They are used to fully configure the DPGM and APGM. This structure is pointed to by `pdiagData` inside the diagnostic profile. The reader is referred to the code for the definitions of the configuration blocks (`sSPE_CFG_XXX`).

Table 11: Diagnostic Data: sSPE_DIAG_DATA_FRM

Field Name	Field Type	Field Description
cfgDPGM[4][3]	sSPE_CFG_DPGM	DROP Bus PRBS Generator and Monitor (DPGM) configuration block
cfgAPGM[4][3]	sSPE_CFG_APGM	ADD Bus PRBS Generator and Monitor (APGM) configuration block

ISR Enable/Disable Mask

Passed via the `spectraSetMask`, `spectraGetMask` and `spectraClearMask` calls, this structure contains all the information needed by the driver to enable and disable any of the interrupts in the SPECTRA-622.

Table 12: ISR Mask: sSPE_MASK

Field Name	Field Type	Field Description
ioScpife[4]	UINT1	Serial control port falling edge
ioScpire[4]	UINT1	Serial control port raising edge
ioDool	UINT1	Data out of lock (DOOL)
ioCrsiRool	UINT1	Reference out of lock (ROOL)
ioLos	UINT1	Loss of signal (LOS)
ioCspiRool	UINT1	Reference out of lock
ioApe[4]	UINT1	Add bus parity error

Field Name	Field Type	Field Description
tocLos	UINT1	Loss of signal (LOS)
tocLof	UINT1	Loss of frame (LOF)
tocLais	UINT1	Line alarm indication signal (LAIS)
tocLrdi	UINT1	Line remote defect indication (LRDI)
tocOof	UINT1	Out of frame (OOF)
tocRdool	UINT1	Receive data out of lock (RDOOL)
tocTrool	UINT1	Transmit reference out of lock (TROOL)
sopOof	UINT1	Out of frame
sopLof	UINT1	Loss of frame
sopLos	UINT1	Loss of signal
sopBipe	UINT1	Bip-8 (B1) error
sstbRtim	UINT1	Receive section trace identifier (Mode1) mismatch
sstbRtiu	UINT1	Receive section trace identifier (Mode 1) unstable
lopSf	UINT1	Signal fail (SF)
lopSd	UINT1	Signal degrade (SD)
lopLrdi	UINT1	Line remote defect indication
lopLais	UINT1	Line alarm indication signal
lopBipe	UINT1	Bip-8 (B2) error
lopLrei	UINT1	Line remote error indication
lopSdber	UINT1	Signal degrade threshold
lopSfber	UINT1	Signal fail threshold
lopZ1S1	UINT1	Change in the received synchronization status
lopCoaps	UINT1	Change in the receive APS code
lopPsbf	UINT1	Protection switch byte failure
rppsTim[4][3]	UINT1	Path trace identifier (Mode 1) mismatch
rppsTiu[4][3]	UINT1	Path trace identifier (Mode 1) unstable
rppsLom1[4][3]	UINT1	Loss of multiframe
rppsLop1[4][3]	UINT1	Loss of pointer

Field Name	Field Type	Field Description
rppsPslm[4][3]	UINT1	Path signal label mismatch
rppsPslu[4][3]	UINT1	Path signal label unstable
rppsPais1[4][3]	UINT1	Path alarm indication signal
rppsPrdi1[4][3]	UINT1	Path remote defect indication
rppsPerdi[4][3]	UINT1	Path enhanced remote defect indication
rppsTiu2[4][3]	UINT1	Path trace identifier mode 2 unstable
rppsPaisCon[4][3]	UINT1	Path alarm indication signal concatenation
rppsLopCon[4][3]	UINT1	Loss of pointer concatenation
rppsNewPtr[4][3]	UINT1	Reception of new_point
rppsPrei[4][3]	UINT1	Path remote error indication
rppsBipe[4][3]	UINT1	Bip-8 error
rppsPrdi2[4][3]	UINT1	Path remote defect indication
rppsPais2[4][3]	UINT1	Path alarm indication signal
rppsAu3PaisCon[4][3]	UINT1	AU3 concatenation path AIS
rppsLop2[4][3]	UINT1	Loss of pointer
rppsAu3LopCon[4][3]	UINT1	AU3 concatenation Loss of pointer
rppsErdi[4][3]	UINT1	Path enhanced remote defect indication
rppsNdf[4][3]	UINT1	Detection of an NDF_enable
rppsPse[4][3]	UINT1	Positive pointer adjustment event
rppsNse[4][3]	UINT1	Negative pointer adjustment event
rppsInvNdf[4][3]	UINT1	Invalid NDF code
rppsDiscopa[4][3]	UINT1	Change of pointer alignment event
rppsIllreq[4][3]	UINT1	Illegal pointer justification request
rppsComa[4][3]	UINT1	Change of multiframe alignment
rppsLom2[4][3]	UINT1	Loss of multiframe
rppsDpje[4][3]	UINT1	DROP bus pointer justification event
rppsEse[4][3]	UINT1	Elastic store error
rppsIsf[4][3]	UINT1	Incoming signal failure
rppsRtim[4][3]	UINT1	Receive path trace identifier (Mode 1) mismatch

Field Name	Field Type	Field Description
rppsRtiu[4][3]	UINT1	Receive path trace identifier (Mode 1) unstable
rppsRpslm[4][3]	UINT1	Receive path signal label mismatch
rppsRpslu[4][3]	UINT1	Receive path signal label unstable
rppsUfl[4][3]	UINT1	Elastic store underflow
rppsOfl[4][3]	UINT1	Elastic store overflow
tppsLom1[4][3]	UINT1	Loss of multiframe
tppsLop1[4][3]	UINT1	Loss of pointer
tppsPais1[4][3]	UINT1	Path alarm indication signal
tppsPaisCon[4][3]	UINT1	Path alarm indication signal concatenation
tppsLopCon[4][3]	UINT1	Loss of pointer concatenation
tppsPje[4][3]	UINT1	Pointer justification event
tppsEse[4][3]	UINT1	Elastic store error
tppsIsf[4][3]	UINT1	Incoming signal failure
tppsNewPtr[4][3]	UINT1	Reception of a new_point indication
tppsPrei[4][3]	UINT1	Path remote error indication
tppsBipe[4][3]	UINT1	Bip-8 error
tppsPais2[4][3]	UINT1	Path alarm indication signal
tppsAu3PaisCon[4][3]	UINT1	AU3 concatenation path alarm indication signal
tppsLop2[4][3]	UINT1	Loss of pointer
tppsAu3LopCon[4][3]	UINT1	AU3 concatenation loss of pointer
tppsNdf[4][3]	UINT1	Detection of an NDF_enable indication
tppsPse[4][3]	UINT1	Positive pointer adjustment event
tppsNse[4][3]	UINT1	Negative pointer adjustment event
tppsInvNdf[4][3]	UINT1	Invalid NDF code
tppsDiscopa[4][3]	UINT1	Change of pointer alignment event
tppsIllreq[4][3]	UINT1	Illegal pointer justification request
tppsComa[4][3]	UINT1	Change of multiframe alignment
tppsLom2[4][3]	UINT1	Loss of multiframe

Field Name	Field Type	Field Description
tppsUfl[4][3]	UINT1	Elastic store underflow
tppsOfl[4][3]	UINT1	Elastic store overflow
wansInt	UINT1	Beginning of a phase averaging period
dpgmGenSig[4][3]	UINT1	DROP generator signal
dpgmMonSig[4][3]	UINT1	DROP monitor signal
dpgmMonErr[4][3]	UINT1	DROP monitor byte error
dpgmMonSync[4][3]	UINT1	DROP monitor synchronize
apgmGenSig[4][3]	UINT1	ADD generator signal
apgmMonSig[4][3]	UINT1	ADD monitor signal
apgmMonErr[4][3]	UINT1	ADD monitor byte error
apgmMonSync[4][3]	UINT1	ADD monitor synchronize

4.3 Structures in the Driver's Allocated Memory

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

Module Data Block: MDB

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

Table 13: Module Data Block: sSPE_MDB

Field Name	Field Type	Field Description
errModule	INT4	Global error Indicator for module calls
valid	UINT2	Indicates that this structure has been initialized
maxDevs	UINT2	Maximum number of devices supported
numDevs	UINT2	Number of devices currently registered
maxInitProfs	UINT2	Maximum number of initialization profiles
maxDiagProfs	UINT2	Maximum number of diagnostic profiles

Field Name	Field Type	Field Description
stateModule	SPE_MOD_STATE	Module state; can be one of the following: SPE_MOD_START, SPE_MOD_IDLE or SPE_MOD_READY
pddb	sSPE_DDB *	(array of) Device Data Blocks (DDB) in context memory
pinitProfs	sSPE_INIT_PROF *	(array of) initialization profiles
pdiagProfs	sSPE_DIAG_PROF *	(array of) diagnostic profiles

Device Data Block: DDB

The DDB is the top-level structure for each Device. It contains configuration data about the Device level code and pointers to configuration data about Device level sub-blocks.

Table 14: Device Data Block: sSPE_DDB

Field Name	Field Type	Field Description
errDevice	INT4	Global error indicator for device calls
valid	UINT2	Indicates that this structure has been initialized
baseAddr	UINT1*	Base address of the Device
usrCtxt	sSPE_USR_CTXT	Stores the user's context for the device. It is passed as an input parameter when the driver invokes an application callback
profileNum	UINT2	Profile number used at initialization
stateDevice	SPE_DEV_STATE	Device State; can be one of the following: SPE_START, SPE_PRESENT, SPE_INACTIVE or SPE_ACTIVE
cfgIO	sSPE_CFG_IO	Input / Output (IO) configuration block
cfgTOC	sSPE_CFG_TOC	Receive / Transmit Transport Overhead Controller (TOC) configuration block

Field Name	Field Type	Field Description
cfgSOP	sSPE_CFG_SOP	Receive / Transmit Section Overhead Processor (RSOP/TSOP) configuration block
cfgSSTB	sSPE_CFG_SSTB	Sonet/SDH Section Trace Buffer (SSTB) configuration block
cfgLOP	sSPE_CFG_LOP	Receive / Transmit Line Overhead Processor (RLOP/TLOP) configuration block
cfgRPPS[4][3]	sSPE_CFG_RPPS	Receive Path Processing Slice (RPPS) configuration block
cfgTPPS[4][3]	sSPE_CFG_TPPS	Transmit Path Processing Slice (TPPS) configuration block
cfgRING	sSPE_CFG_RING	Ring Control Port (RING) configuration block
cfgWANS	sSPE_CFG_WANS	WAN Synchronization controller (WANS) configuration block
cfgDPGM[4][3]	sSPE_CFG_DPGM	DROP Bus PRBS Generator and Monitor (DPGM) configuration block
cfgAPGM[4][3]	sSPE_CFG_APGM	ADD Bus PRBS Generator and Monitor (APGM) configuration block
cfgCnt	sSPE_CFG_CNT	Counter configuration structure
pollISR	SPE_POLL	Indicates the current type of ISR / polling
cbackIO	sSPE_CBACK	Address for the callback function for IO Events
cbackTOC	sSPE_CBACK	Address for the callback function for TOC Events
cbackSOP	sSPE_CBACK	Address for the callback function for SOP Events
cbackSSTB	sSPE_CBACK	Address for the callback function for SSTB Events
cbackLOP	sSPE_CBACK	Address for the callback function for LOP Events
cbackRPPS	sSPE_CBACK	Address for the callback function for RPPS Events

Field Name	Field Type	Field Description
cbackTPPS	sSPE_CBACK	Address for the callback function for TPPS Events
cbackWANS	sSPE_CBACK	Address for the callback function for WANS Events
cbackDPGM	sSPE_CBACK	Address for the callback function for DPGM Events
cbackAPGM	sSPE_CBACK	Address for the callback function for APGM Events
mask	sSPE_MASK	Interrupt Enable Mask

Input / Output (IO) Status

Table 15: Input/Output Status: sSPE_STATUS_IO

Field Name	Field Type	Field Description
refclkActive	UINT1	Monitors for low to high transitions on the REFCLK reference clock input.
rool	UINT1	Monitors the transmit reference out of lock status to report if the synthesis phase lock loop is unable to lock to the reference clock on REFLCK.
dckAct	UINT1	Monitors for low to high transitions on the DCK input.
ackActiv	UINT1	Monitors for low to high transitions on the ACK input.
insLRDI	UINT1	Reports the value of the SENDLRDI bit position in the transmit ring control port.
insLAIS	UINT1	Reports the value of the SENDLAIS bit position in the transmit ring control port.
rlos	UINT1	The loss of transition status indicates the receive power is lost or at least 95 consecutive ones or zeros have been received.

Field Name	Field Type	Field Description
`rrool`	`UINT1`	Monitors the recovered reference out of lock status to report if the clock recovery phase locked loop is unable to lock to the reference clock on REFCLK.
`rdool`	`UINT1`	Monitors the recovered data out of lock status to report if the clock recovery phase locked loop is unable to recover and lock to the input data stream.
`ds3tdatAct`	`UINT1`	Monitors for low to high transitions on the sampled DS3TDAT input for the TPPS.
`ds3tiClkAct`	`UINT1`	Monitors for low to high transitions on the DS3TICLK input for the TPPS.
`addControlAct[4]`	`UINT1`	Monitors for low to high transitions on the corresponding APL[n], AC1J1V1[n] and ADP[n] inputs`.addControlActiv[n]` is non-zero when rising edges have been observed on all these signals.
`addDataAct[4]`	`UINT1`	Monitors for low to high transitions on the corresponding AD[7:0] (#1), AD[15:8] (#2), AD[23:16] (#3) or AD[31:24] (#4) bus when configured for byte Telecom ADD bus mode. addDataActiv[n] is non-zero when rising edges have been observed on all the required signals in the corresponding Telecom ADD bus.
`scpi[4]`	`UINT1`	Status of the associated SCPI[3:0] input pins.

Section Overhead Processor (SOP) Status

Table 14: Section Overhead Processor Status: sSPE_STATUS_SOP

Field Name	Field Type	Field Description
`los`	`UINT1`	The LOSV bit is set high when loss of signal is declared. LOS is removed when two valid framing words (A1, A2) are detected, and during the intervening time (125 µs), no violating period of all zeros patterns is observed.

Field Name	Field Type	Field Description
lof	UINT1	The LOFV bit is set high when loss of frame is declared. LOFV is set high and loss of frame declared when an out-of-frame state persists for 3 ms. LOF is removed when an in frame state persists for 3 ms.
oof	UINT1	The OOFV bit is set high when out of frame is declared. OOFV is set high and out-of frame declared while the SPECTRA-622 is unable to find a valid framing pattern (A1, A2) in the incoming stream. OOF is removed when a valid framing pattern is detected.
tiu	UINT1	Monitors the receive section trace identifier unstable status, which is dependent on the Trace Identifier Mode. In Mode 1, the bit is set high when 8 trace messages mismatching against their immediate predecessor message have been received without a persistent message being detected. In Mode 2, RTIUV is set low during the stable state which is declared after having received the same 16 byte trace message 3 consecutive times.
tim	UINT1	Monitors the receive section trace identifier mismatch status to report if the accepted message differs from the expected message.

Line Overhead Processor (LOP) Status

Table 15: Line Overhead Status: sSPE_STATUS_LOP

Field Name	Field Type	Field Description
sfber	UINT1	Indicate the signal failure threshold crossing alarm state.
sdber	UINT1	Indicates the signal degrade threshold crossing alarm state.
psbf	UINT1	Indicates the protection switching byte failure alarm state.
lrdi	UINT1	Indicates when the line Remote Defect Indication (RDI) is detected.
lais	UINT1	Indicates when the line Alarm Indication Signal (AIS) is detected.

Receive Path Processing Slice (RPPS) Status

Table 16: Receive Path Status: sSPE_STATUS_RPPS

Field Name	Field Type	Field Description
ptiu	UINT1	Monitors the receive path trace identifier unstable status bit (RTIUV), which is dependent on the Trace Identifier Mode. In Mode 1, the bit is set high when 8 trace messages mismatching against their immediate predecessor message have been received without a persistent message being detected. In Mode 2, RTIUV is set low during the stable state which is declared after having received the same 16 byte trace message 3 consecutive times.
ptim	UINT1	Monitors the receive path trace identifier mismatch status bit (RTIMV) in Trace Identifier Mode 1 to report if the accepted message differs from the expected message.
au3paisc	UINT1	Indicates reception of path AIS alarm in the concatenation indication in the receive STS-1 (STM-0/AU3) or equivalent stream.

Field Name	Field Type	Field Description
au3plopc	UINT1	Indicates entry to LOPCON_state for the receive STS-1 (STM-0/AU3) or equivalent stream in the RPOP pointer interpreter.
pais	UINT1	Indicates reception of path AIS alarm in the receive stream.
lop	UINT1	Indicates entry to the LOP_state in the RPOP pointer interpreter state machine.
prdi	UINT1	Indicates reception of path RDI alarm in the receive stream.
erdiv	UINT1	Reflect the current filtered value of the enhanced RDI codepoint (G1 bits 5, 6, & 7) for the receive SONET/SDH stream. Filtering is controlled using rdi10 in the RPPS configuration block.
lom	UINT1	Reports the current state of the multiframe framer monitoring the receive stream.
isf	UINT1	Reports an incoming signal fail alarm.
uneq	UINT1	Monitors the unequipped status bit (UNEQV), which is dependent on the PSL Mode. In Mode 1, this bit is set high when the accepted path signal label indicates that the path connection is unequipped. When in PSL Mode 2, the UNEQV is set high upon the reception of five consecutive frames with an unequipped (00h) label.
pslm	UINT1	Monitors the receive path signal label mismatch status bit (RPSLMV), which is dependent on the PSL Mode. In Mode 1, this bit reports the match/mismatch status between the expected and the accepted path signal label. In Mode 2, this bit reports the match/mismatch status between the expected and the received path signal label.
pslu	UINT1	Monitors the receive path signal label unstable status bit (RPSLUV) and is independent on the PSL Mode. This bit reports the stable/unstable status of the path signal label in the receive stream.

Field Name	Field Type	Field Description
dropGenSig	UINT1	Indicates if the partial pseudo random sequence (PRBS) begin generated is correctly aligned with the partial PRBS begin generated in the master generator.
dropMonSig	UINT1	Indicates if the partial pseudo random sequence (PRBS) being monitored for is correctly aligned with the partial PRBS being monitored for by the master generator.
dropMonSync	UINT1	Reports when the monitor is out of synchronization

Transmit Path Processing Slice (TPPS) Status

Table17: Transmit Path Status: sSPE_STATUS_TPPS

Field Name	Field Type	Field Description
isf	UINT1	Reports an incoming signal failure detected.
au3lopc	UINT1	Indicates entry to LOPCON_state for the transmit STS-1 (STM-0/AU3) or equivalent stream in the TPIP pointer interpreter.
au3paisc	UINT1	Indicates reception of path AIS alarm in the concatenation indication in the transmit STS-1 (STM-0/AU3) or equivalent stream.
lop	UINT1	Indicates entry to the LOP_state in the TPIP pointer interpreter state machine.
pais	UINT	Indicates reception of path AIS alarm in the receive stream.
rdi	UINT1	Indicates remote defect indication detected in transmit stream.
lom	UINT1	Reports the current state of the multiframe framer monitoring the receive stream.
addGenSig	UINT1	Indicates if the partial pseudo random sequence (PRBS) begin generated is correctly aligned with the partial PRBS begin generated in the master generator.

Field Name	Field Type	Field Description
addMonSig	UINT1	Indicates if the partial pseudo random sequence (PRBS) being monitored for is correctly aligned with the partial PRBS being monitored for by the master generator
addMonSync	UINT1	Reports when the monitor is out of synchronization
addMonSync	UINT1	Reports when the monitor is out of synchronization

Statistic Counter Configuration (CFG_CNT)

This structure contains all the fields needed to configure the device counters. It is also passed via the `spectraCfgStats` function call.

Table 16: Counters Config: sSPE_CFG_CNT

Field Name	Field Type	Field Description
sopBlkBip	UINT1	Enables the accumulating of section block BIP errors. When non-zero, one or more errors in the section BIP-8 byte (B1) results in a single error accumulated in the B1 error counter. When zero, all errors in the B1 byte are accumulated in the B1 error counter.
lopBlkRei	UINT1	Controls the accumulation of REI's. When non-zero, and the REI has a value between 1 and 4, the REI event counter is incremented for each set REI bit. If the REI has value greater than 4, and is valid, the REI counter is only incremented by 4. When zero, the REI event counter is incremented for each and every REI bit that occurs during that frame. The counter may be incremented up to 96 times. The REI counter is not incremented for invalid REI codewords.

Field Name	Field Type	Field Description
lopBlkBip	UINT1	Controls the accumulation of B2 errors. When non-zero, the B2 error event counter is incremented only once per frame whenever one or more B2 bit errors occur during that frame. When zero, the B2 error event counter is incremented for each B2 bit error that occurs during that frame (the counter can be incremented up to 96 times per frame).
rppsMonrs[4][3]	UINT1	When non-zero, selects the receive side pointer justification events counters to monitor the receive stream directly. When zero, the counters accumulates pointer justification events on the DROP bus.
rppsBlkBip[4][3]	UINT1	When non-zero, indicates that path BIP-8 errors are to be reported and accumulated on a block basis. A single BIP error is accumulated and reported to the return transmit path overhead processor if any of the BIP-8 results indicates a mismatch. When zero, BIP-8 errors are accumulated on a bit basis.
rppsBlkRei[4][3]	UINT1	When non-zero, block REI indicates that path REI counts are to be reported and accumulated on a block basis. A single REI error is accumulated if the received REI code is between 1 and 8 inclusive. When zero, REI errors are accumulated literally.

Statistic Counters (CNT)

This structure, as well as its component structures, is being used by the statistics collection APIs to retrieve the device counts. The user can either collect all statistics at once by using `spectraGetCnt`, or collect statistics from individual blocks using `spectraGetCntSOP`, `spectraGetCntLOP`, `spectraGetCntRPPS`, `spectraGetCntTPPS`, and/or `spectraGetCntPJ`.

Table 17: Statistic Counters: sSPE_STAT_CNT

Field Name	Field Type	Field Description
cntSOP	sSPE_STAT_CNT_SOP	Statistics counters of the Section Overhead (SOH)
cntLOP	sSPE_STAT_CNT_LOP	Statistics counters of the Line Overhead (LOH)
cntRPPS[4][3]	sSPE_STAT_CNT_RPPS	Statistics counters of the Receive Path Overhead (RPOH)
cntTPPS[4][3]	sSPE_STAT_CNT_TPPS	Statistics counters of the Transmit Path Overhead (TPOH)
cntPJ[4][3]	sSPE_STAT_CNT_PJ	Statistics counters of the Pointer Justifications

Section Overhead (SOP) Statistics Counters

Table 18: Section Overhead Statistics Counters: sSPE_STAT_CNT_SOP

Field Name	Field Type	Field Description
sopBip	UINT4	Section BIP errors counter

Line Overhead (LOP) Statistics Counters

Table 19: Line Overhead Statistic Counters: sSPE_STAT_CNT_LOP

Field Name	Field Type	Field Description
lopBip	UINT4	Line BIP errors counter
lopRei	UINT4	Line REI error counter

Receive Path Overhead (RPOH) Statistics Counters

Table 20: SPECTRA-622 Receive Path Processing Statistics Counters: sSPE_STAT_CNT_RPPS

Field Name	Field Type	Field Description
rppsBip	UINT4	Path BIP error counter
rppsRei	UINT4	Path REI error counter
rppsDPGMPrse	UINT4	Number of PRBS byte errors detected since the last accumulation interval. Errors are only accumulated in the synchronized state and each PRBS data byte can have a maximum of 1 errors.

Transmit Path Overhead (TPOH) Statistics Counters

Table 21: Transmit Path Processing Statistics Counters: STAT_CNT_TPPS

Field Name	Field Type	Field Description
tppsAPGMPrse	UINT4	Number of PRBS byte errors detected since the last accumulation interval. Errors are only accumulated in the synchronized state and each PRBS data byte can have a maximum of 1 errors.

Pointer Justification Statistics Counters

Table 22: Pointer Justification Statistics Counters: STAT_CNT_PJ

Field Name	Field Type	Field Description
rppsPosJust	UINT4	Positive RPPS pointer justification event counter
rppsNegJust	UINT4	Negative RPPS pointer justification event counter
tppsPosJust	UINT4	Positive TPPS pointer justification event counter
tppsNegJust	UINT4	Negative TPPS pointer justification event counter

4.4 Structures Passed Through RTOS Buffers

Interrupt Service Vector: ISV

This block is used in two ways. First it is used to determine the size of buffer required by the RTOS for use in the driver. Second it is the template for data that is captured during ISR processing and sent to the Deferred Processing Routine (DPR).

Table 23: Interrupt Service Vector: sSPE_ISV

Field Name	Field Type	Field Description
deviceHandle	sSPE_HNDL	Handle to the device in cause
mask	sSPE_MASK	sSPE_MASK

Deferred Processing Vector: DPV

This block is used in two ways. First it is used to determine the size of buffer required by the RTOS for use in the driver. Second it is the template for data that is assembled by the DPR and sent to the application code.

Note: the application code is responsible for returning this buffer to the RTOS buffer pool.

Table 24: Deferred Processing Vector: sSPE_DPV

Field Name	Field Type	Field Description
event	SPE_DPR_EVENT	Event being reported
cause	UINT2	Reason for the Event

4.5 Global Variable

Most variables within the driver are not meant to be used by the application code. There is one, however, that can be of great use to the application code:

`spectraMdb`: A global pointer to the Module Data Block (MDB). This global variable is to be considered read only by the application.

- `errModule`: This structure element is used to store an error code that specifies the reason for an API function's failure. The field is only valid when the function in question returns a `SPE_FAILURE` value.
- `stateModule`: This structure element is used to store the Module state.
- `pddb[ ]`: An array of pointers to the individual Device Data Blocks. The USER is cautioned that a DDB is only valid if the '`valid`' flag is set. Note that the DDBs are in no particular order.
  - `errDevice`: This structure element is used to store an error code that specifies the reason for an API function's failure. The field is only valid when the function in question returns a `SPE_FAILURE` value.
  - `stateDevice`: This structure element is used to store the Device state.

5 Application Programming Interface

This section provides a detailed description of each function that is a member of the SPECTRA-622 driver Application Programming Interface (API).

5.1 Module Initialization

Opening the Driver Module: spectraModuleOpen

This function performs module level initialization of the device driver. This involves allocating all of the memory needed by the driver and initializing the Module Data Block (MDB) with the passed Module Initialization Vector (MIV).

Prototypes	`INT4 spectraModuleOpen(sSPE_MIV *pmiv, sSPE_MDB** ppmdb)`	
Inputs	`pmiv`	: (pointer to) Module Initialization Vector
	`ppmdb`	: (pointer to) pointer to the Module Data Block
Outputs	`ppmdb`	: pointer to the Module Data Block
Returns	Success = `SPE_SUCCESS` Failure = `<SPECTRA-622 ERROR CODE>`	
Valid States	START	
Side Effects	Changes MODULE state to IDLE	

Closing the Driver Module: spectraModuleClose

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

Prototype	`INT4 spectraModuleClose(void)`
Inputs	None

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = `<SPECTRA-622 ERROR CODE>`

Valid States ALL STATES

Side Effects Changes MODULE state to START

5.2 Module Activation

Starting the Driver Module: spectraModuleStart

This function performs module level startup of the driver. This involves allocating semaphores and timers, initializing buffers and installing the ISR handler and DPR task. Upon successful return of this function the driver is ready to add devices.

Prototype `INT4 spectraModuleStart(void)`

Inputs None

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = `<SPECTRA-622 ERROR CODE>`

Valid States IDLE

Side Effects Changes MODULE state to READY

Stopping the Driver Module: spectraModuleStop

This function performs module level shutdown of the driver. This involves deleting all devices being controlled by the driver and removing the ISR handler and DPR task.

Prototype	`INT4 spectraModuleStop(void)`
Inputs	None
Outputs	None
Returns	Success = `SPE_SUCCESS` Failure = < `SPECTRA-622 ERROR CODE`>
Valid States	READY
Side Effects	Changes MODULE state to IDLE

5.3 Profile Management

Initialization Profile

Creating an Initialization Profile: spectraAddInitProfile

This function creates an initialization profile that is stored by the driver. A device can now be initializaed by simply passing an initialization profile number.

Prototype	`INT4 spectraAddInitProfile(sSPE_INIT_PROF *pProfile, UINT2 *pProfileNum)`	
Inputs	`pProfile`	: (pointer to) initialization profile being added
	`pProfileNum`	: (pointer to) profile number to be assigned by the driver
Outputs	`pProfileNum`	: profile number assigned by the driver
Returns	Success = `SPE_SUCCESS` Failure = <`SPECTRA-622 ERROR CODE`>	
Valid States	IDLE, READY	
Side Effects	None	

Retrieving an Initialization Profile: spectraGetInitProfile

This function retrieves the contents of the initialization profile.

Prototype `INT4 spectraGetInitProfile(UINT2 profileNum, sSPE_INIT_PROF *pProfile)`

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

Outputs `pProfile` : contents of the corresponding profile

Returns
Success = `SPE_SUCCESS`
Failure = `<SPECTRA-622 ERROR CODE>`

Valid States IDLE, READY

Side Effects None

Deleting an Initialization Profile: spectraDeleteInitProfile

This function deletes an initialization profile given its profile number.

Prototype `INT4 spectraDeleteInitProfile(UINT2 profileNum)`

Inputs `profileNum` : initialization profile number

Outputs None

Returns
Success = `SPE_SUCCESS`
Failure = `<SPECTRA-622 ERROR CODE>`

Valid States IDLE, READY

Side Effects None

Diagnostic Profile

Creating a Diagnostic Profile: spectraAddDiagProfile

This function creates a diagnostic profile that is stored by the driver. Passing the diagnostic profile number starts a diagnostic.

Prototype	`INT4 spectraAddDiagProfile(sSPE_DIAG_PROF *pProfile, UINT2 *pProfileNum)`	
Inputs	`pProfile`	: (pointer to) diagnostic profile being added
	`pProfileNum`	: (pointer to) profile number
Outputs	`pProfileNum`	: profile number assigned by the driver
Returns	Success = `SPE_SUCCESS` Failure = `<SPECTRA-622 ERROR CODE>`	
Valid States	IDLE, READY	
Side Effects	None	

Retrieving a Diagnostic Profile: spectraGetDiagProfile

This function retrieves the contents of a diagnostic profile.

Prototype	`INT4 spectraGetDiagProfile(UINT2 profileNum, sSPE_DIAG_PROF *pProfile)`	
Inputs	`profileNum`	: diagnostic profile number
	`pProfile`	: (pointer to) diagnostic profile
Outputs	`pProfile`	: contents of the corresponding profile
Returns	Success = `SPE_SUCCESS` Failure = `<SPECTRA-622 ERROR CODE>`	

Valid States IDLE, READY

Side Effects None

Deleting a Diagnostic Profile: spectraDeleteDiagProfile

This function deletes a diagnostic profile.

Prototype `INT4 spectraDeleteDiagProfile(UINT2 profileNum)`

Inputs `profileNum` : diagnostic profile number

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = `<SPECTRA-622 ERROR CODE>`

Valid States IDLE, READY

Side Effects None

5.4 Device Addition and Deletion

Adding a Device: spectraAdd

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

Prototype `sSPE_HNDL spectraAdd(void *usrCtxt, UINT1 *baseAddr, INT4 **pperrDevice)`

Inputs

`usrCtxt`	: user context for this device
`baseAddr`	: base address of the device
`pperrDevice`	: (pointer to) an area of memory

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

Returns Device handle (to be used as an argument to most of the SPECTRA-622 APIs) or NULL pointer in case of an error

Valid States READY

Side Effects Changes the DEVICE state to PRESENT

Deleting a Device: spectraDelete

This function is used to remove the specified device from the list of devices being controlled by the SPECTRA-622 driver. Deleting a device involves clearing the DDB for that device and releasing its associated device handle.

Prototype `INT4 spectraDelete(sSPE_HNDL deviceHandle)`

Inputs	`deviceHandle`	: device Handle (from `spectraAdd`)

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = `<SPECTRA-622 ERROR CODE>`

Valid States PRESENT, ACTIVE, INACTIVE

Side Effects None

5.5 Device Initialization

Initializing a Device: spectraInit

This function initializes the Device Data Block (DDB) that is associated with that device during `spectraAdd`. It applies a reset to the device and configures it according to the DIV passed by the Application. If the DIV is passed as a NULL the profile number is used. A profile number of zero indicates that all the register bits are to be left in their default state (after a soft reset). Note that the profile number is ignored UNLESS the passed DIV is NULL.

Prototype	`INT4 spectraInit(sSPE_HNDL deviceHandle, sSPE_DIV *pdiv, UINT2 profileNum)`	
Inputs	`deviceHandle`	: device Handle (from `spectraAdd`)
	`pdiv`	: (pointer to) Device Initialization Vector
	`profileNum`	: profile number (ignored if `pdiv` is NULL)
Outputs	None	
Returns	Success = `SPE_SUCCESS` Failure = `<SPECTRA-622 ERROR CODE>`	
Valid States	PRESENT	
Side Effects	Changes DEVICE state to INACTIVE	

Updating the Configuration of a Device: spectraUpdate

Updates the configuration of the device as well as the Device Data Block (DDB) associated with that device according to the DIV passed by the Application. The only difference between `spectraUpdate` and `spectraInit` is that no soft reset will be applied to the device.

Prototype	`INT4 spectraInit(sSPE_HNDL deviceHandle, sSPE_DIV *pdiv, UINT2 profileNum)`	
Inputs	`deviceHandle`	: device Handle (from `spectraAdd`)
	`pdiv`	: (pointer to) Device Initialization Vector
	`profileNum`	: profile number (ignored if `pdiv` is NULL)

Outputs None

Returns Success = SPE_SUCCESS
Failure = <SPECTRA-622 ERROR CODE>

Valid States PRESENT

Side Effects Changes DEVICE state to INACTIVE

Resetting a Device: spectraReset

This function applies a software reset to the SPECTRA-622 device. It also resets all the DDB contents (except for the user context). This function is typically called before re-initializing the device.

Prototype `void spectraReset(sSPE_HNDL deviceHandle)`

Inputs `deviceHandle` : device Handle (from `spectraAdd`)

Outputs None

Returns None

Valid States ACTIVE, INACTIVE

Side Effects Changes DEVICE state to PRESENT

5.6 Device Activation and De-Activation

Activating a Device: spectraActivate

This function restores the state of a device after it has been deactivated. Interrupts may be re-enabled after deactivation.

Prototype	`INT4 spectraActivate(sSPE_HNDL deviceHandle)`
Inputs	`deviceHandle` : device Handle (from `spectraAdd`)
Outputs	None
Returns	Success = `SPE_SUCCESS` Failure = <`SPECTRA-622 ERROR CODE`>
Valid States	Inactive
Side Effects	Change the DEVICE state to ACTIVE

DeActivating a Device: spectraDeActivate

This function de-activates the device from operation. In the process, interrupts are masked and the device is put into a quiet state via enable bits.

Prototype	`INT4 spectraDeActivate(sSPE_HNDL deviceHandle)`
Inputs	`deviceHandle` : device Handle (from `spectraAdd`)
Outputs	None
Returns	Success = `SPE_SUCCESS` Failure = <`SPECTRA-622 ERROR CODE`>
Valid States	ACTIVE
Side Effects	Changes the DEVICE state to INACTIVE

5.7 Device Reading and Writing

Reading from a Device Register: spectraRead

This function can be used to read a register of a specific SPECTRA-622 device by providing the register number. This function derives the actual address location based on the device handle and register number inputs. It then reads the contents of this address location using the system specific macro, `sysSpectraRead`.

Note: A failure to read returns a zero and any error indication is written to the DDB.

Prototype	`UINT1 spectraRead(sSPE_HNDL deviceHandle, UINT2 regNum)`	
Inputs	`deviceHandle`	: device Handle (from `spectraAdd`)
	`regNum`	: register number
Outputs	ERROR code written to the DDB	
Returns	Success = the register value Failure = `0x00`	
Valid States	ALL DEVICE STATES	
Side Effects	May affect registers that change after a read operation	

Writing to a Device: spectraWrite

This function can be used to write to a register of a specific SPECTRA-622 device by providing the register number. The function derives the actual address location based on the device handle and register number inputs. It then writes the contents of this address location using the system specific macro `sysSpectraWrite`.

Note: A failure to write returns a zero and any error indication is written to the DDB.

Prototype	`UINT1 spectraWrite(sSPE_HNDL deviceHandle, UINT2 regNum, UINT1 value)`	
Inputs	`deviceHandle`	: device Handle (from `spectraAdd`)
	`regNum`	: register number
	`value`	: value to be written

Outputs ERROR code written to the DDB

Returns Success = previous value
Failure = `0x00`

Valid States ALL DEVICE STATES

Side Effects May change the configuration of the Device

Reading a Block of Registers: spectraReadBlock

This function can be used to read a register block of a specific SPECTRA-622 device by providing the starting register number, and the size to read. The function derives the actual start address location based on the device handle and starting register number inputs. It then reads the contents of this data block using multiple calls to the system specific macro and `sysSpectraRead`.

Note: Any error indication is written to the DDB. It is the USER's responsibility to allocate enough memory for the block read.

Prototype `void spectraReadBlock(sSPE_HNDL deviceHandle, UINT2 startRegNum, UINT2 size, UINT1 *pblock)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`startRegNum`	: starting register number
`size`	: size of the block to read
`pblock`	: (pointer to) the block to read

Outputs ERROR code written to the DDB

`pblock`	: (pointer to) the block read

Returns None

Valid States ALL DEVICE STATES

Side Effects May affect registers that change after a read operation

Writing a Block of Registers: spectraWriteBlock

This function can be used to write to a register block of a specific SPECTRA-622 device by providing the starting register number and the block size. The function derives the actual starting address location based on the device handle and starting register number inputs. It then writes the contents of this data block using multiple calls to the system specific macro and `sysSpectraWrite`. A bit from the passed block is only modified in the device's registers if the corresponding bit is set in the passed mask.

Note: Any error indication is written to the DDB

Prototype `void spectraWriteBlock(sSPE_HNDL deviceHandle, UINT2 startRegNum, UINT2 size, UINT1 *pblock, UINT1 *pmask)`

Inputs

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

Outputs ERROR code written to the DDB

Returns None

Valid States ALL DEVICE STATES

Side Effects May change the configuration of the Device

5.8 Transport Overhead Controller (TOC)

Modifying the Z0 Byte: spectraTOCWriteZ0

This function writes the Z0 byte into the transmit transport overhead.

Prototype `INT4 spectraTOCWriteZ0(sSPE_HNDL deviceHandle, UINT1 Z0)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`Z0`	: Z0 byte to write

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Modifying the S1 Byte: spectraTOCWriteS1

This function writes the S1 byte into the transmit transport overhead.

Prototype `INT4 spectraTOCWriteS1(sSPE_HNDL deviceHandle, UINT1 S1)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`S1`	: S1 byte to write

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Reading the S1 Byte: spectraTOCReadS1

This function reads the S1 byte received in the transport overhead of the received stream.

Prototype `INT4 spectraTOCReadS1(sSPE_HNDL deviceHandle, UINT1 *pS1)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`pS1`	: (pointer to) S1 byte

Outputs `pS1` : S1 byte read

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

5.9 Receive / Transmit Section Overhead Processor (RSOP/TSOP)

Forcing Out-of-Frame: spectraSOPForceOOF

When the enable flag is set, this function forces the Receive section overhead processor out-of-frame. When the enable flag is not set, the function resumes normal processing.

Prototype `INT4 spectraSOPForceOOF(sSPE_HNDL deviceHandle, UINT2 enable)`

Inputs `deviceHandle` : device Handle (from `spectraAdd`)
`enable` : flag to start/stop forcing OOF

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Inserting Line AIS: spectraSOPInsertLineAIS

When the enable flag is set, this function forces a Line-AIS insertion. When the enable flag is not set, the function resumes normal processing.

Prototype `INT4 spectraSOPInsertLineAIS(sSPE_HNDL deviceHandle, UINT2 enable)`

Inputs `deviceHandle` : device Handle (from `spectraAdd`)
`enable` : flag to start/stop Line-AIS insertion

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = `<SPECTRA-622 ERROR CODE>`

Valid States ACTIVE, INACTIVE

Side Effects None

Forcing Errors in the A1 Byte: spectraSOPDiagFB

This function enables the insertion of a single bit error continuously in the most significant bit (bit 1) of the A1 section overhead framing byte. A1 bytes are set to 76H instead of F6H.

Prototype `INT4 spectraSOPDiagFB(sSPE_HNDL deviceHandle, UINT2 enable)`

Inputs `deviceHandle` : device Handle (from `spectraAdd`)
`enable` : flag to start/stop error insertion

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = `<SPECTRA-622 ERROR CODE>`

Valid States ACTIVE, INACTIVE

Side Effects None

Forcing Errors in the B1 Byte: spectraSOPDiagB1

This function enables insertion of bit errors continuously in the B1 section overhead byte. The B1 byte value is inverted.

Prototype `INT4 spectraSOPDiagB1(sSPE_HNDL deviceHandle, UINT2 enable)`

Inputs
`deviceHandle` : device Handle (from `spectraAdd`)
`enable` : flag to start/stop error insertion

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Forcing Loss-Of-Signal: spectraSOPDiagLOS

This function enables the insertion of zeros in the transmit outgoing stream.

Prototype `INT4 spectraSOPDiagLOS(sSPE_HNDL deviceHandle, UINT2 enable)`

Inputs
`deviceHandle` : device Handle (from `spectraAdd`)
`enable` : flag to start/stop error insertion

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

5.10 SONET / SDH Section Trace Buffer (SSTB)

Retrieving and Setting the Section Trace Messages: spectraSectionTraceMsg

This function retrieves and sets the section trace message (J0) in the Sonet/SDH Section Trace Buffer.

Note: It is the USER's responsibility to ensure that the message pointer points to an area of memory large enough to hold the returned data.

Prototype `INT4 spectraSectionTraceMsg(sSPE_HNDL deviceHandle, UINT2 type, UINT1* pJ0)`

Inputs

`deviceHandle` : device Handle (from spectraAdd)

`type` : type of access
- 0 = write tx section trace
- 1 = read rx accepted section trace
- 2 = read rx captured section trace
- 3 = write rx expected section trace

`pJ0` : (pointer to) the section trace message

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = `<SPECTRA-622 ERROR CODE>`

Valid States ACTIVE, INACTIVE

Side Effects None

5.11 Receive / Transmit Line Overhead Processor (RLOP/TLOP)

Inserting Line Remote Defect Indication: spectraLOPInsertLineRDI

This function enables the insertion of a transmit line remote defect indication (RDI). The Line RDI is inserted by transmitting the code 110 in bit positions 6, 7, and 8 of the K2 byte.

Prototype	`INT4 spectraLOPInsertLineRDI(sSPE_HNDL deviceHandle, UINT2 enable)`	
Inputs	`deviceHandle`	: device Handle (from `spectraAdd`)
	`enable`	: flag to start/stop Line RDI insertion
Outputs	None	
Returns	Success = `SPE_SUCCESS`	
	Failure = <`SPECTRA-622 ERROR CODE`>	
Valid States	ACTIVE, INACTIVE	
Side Effects	None	

Forcing Errors in the B2: spectraLOPDiagB2

This function enables the insertion of bit errors continuously in each of the line BIP-8 bytes (B2 bytes). Each bit of every B2 is inverted.

Prototype	`INT4 spectraLOPDiagB2(sSPE_HNDL deviceHandle, UINT2 enable)`	
Inputs	`deviceHandle`	: device Handle (from `spectraAdd`)
	`enable`	: flag to start/stop B2 error insertion
Outputs	None	
Returns	Success = `SPE_SUCCESS`	
	Failure = <`SPECTRA-622 ERROR CODE`>	
Valid States	ACTIVE, INACTIVE	
Side Effects	None	

Reading the Received K1 and K2 Bytes: spectraLOPReadK1K2

This function reads the K1 and K2 bytes from the received line overhead.

Prototype `INT4 spectraLOPReadK1K2(sSPE_HNDL deviceHandle, UINT1 *pK1, UINT1 *pK2)`

Inputs
`deviceHandle` : device Handle (from `spectraAdd`)
`pK1` : (pointer to) K1 byte
`pK2` : (pointer to) K2 byte

Outputs
`pK1` : K1 byte read
`pK2` : K2 byte read

Returns
Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Writing the Transmitted K1 and K2 Bytes: spectraLOPWriteK1K2

This function writes the K1 and K2 bytes into the transmit line overhead.

Prototype `INT4 spectraLOPWriteK1K2(sSPE_HNDL deviceHandle, UINT1 K1, UINT1 K2)`

Inputs
`deviceHandle` : device Handle (from `spectraAdd`)
`K1` : K1 byte to write
`K2` : K2 byte to write

Outputs None

Returns
Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

5.12 Receive Path Processing Slice (RPPS)

Retrieving and Setting the Path Trace Messages: spectraPathTraceMsg

This function retrieves and sets the current path trace message (J1) in the Sonet/SDH Path Trace Buffer. Note: It is the USER's responsibility to make sure that the message pointer points to an area of memory large enough to hold the returned data.

Prototype `INT4 spectraPathTraceMsg(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2 au3, UINT2 type, UINT1* pJ1)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`type`	: type of access 0 = write tx path trace 1 = read rx accepted path trace 2 = read rx captured path trace 3 = write rx expected path trace
`pJ1`	: (pointer to) the path trace message

Outputs `pJ1` : updated path trace message

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Forcing Loss-Of-Pointer: spectraRPPSDiagLOP

This function forces the downstream pointer processing to enter the Loss of Pointer (LOP) state. It does so by inverting the new data flag (NDF) field of the payload pointer that is inserted in the DROP bus.

Prototype `INT4 spectraRPPSDiagLOP(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2 au3, UINT2 enable)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`enable`	: flag to start/stop NDF inversion

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Forcing Errors in the H4 Byte: spectraRPPSDiagH4

This function enables the inversion of the multiframe indicator (H4) byte in the DROP bus. An inversion forces an out-of-multiframe alarm in the downstream circuitry. This can only occur when the SPE (VC) is used to carry virtual tributary (VT) or tributary unit (TU) based payloads.

Prototype `INT4 spectraRPPSDiagH4(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2 au3, UINT2 enable)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`enable`	: flag to start/stop H4 inversion

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Forcing Tributary Path AIS: spectraRPPSInsertTUAIS

This function enables the insertion of tributary path AIS on the DROP bus for VT1.5 (TU11), VT2 (TU12), VT3 and VT6 (TU2) payloads. Columns in the DROP bus carrying tributary traffic are set to all ones. The pointer bytes (H1, H2, and H3), the path overhead column, and the fixed stuff columns remain unaffected. Note: This is not applicable for TU3 tributary payloads.

Prototype `INT4 spectraRPPSInsertTUAIS(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2 au3, UINT2 enable)`

Inputs

	`deviceHandle`	: device Handle (from `spectraAdd`)
	`stm1`	: STM-1 index
	`au3`	: AU-3 index
	`enable`	: flag to start/stop TUAIS

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Forcing DS3 AIS: spectraRPPSDs3AisGen

Forces generation of DS3 AIS. Note: Any data on the STS-1 (STM-0/AU3) SPE is then lost.

Prototype `INT4 spectraRPPSDs3AisGen(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2 au3, UINT2 enable)`

Inputs

	`deviceHandle`	: device Handle (from `spectraAdd`)
	`stm1`	: STM-1 index
	`au3`	: AU-3 index

	`enable`	: flag to start/stop DS3 AIS generation

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

5.13 Transmit Path Processing Slice (TPPS)

Forcing Path AIS: spectraTPPSInsertPAIS

This function enables the insertion of the path alarm indication signal (PAIS) in the transmit stream. The synchronous payload envelope and the pointer bytes (H1 – H3) are set to all ones.

Prototype

```
INT4 spectraTPPSInsertPAIS(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2
au3, UINT2 enable)
```

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`enable`	: flag to start/stop PAIS insertion

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Forcing Errors in the B3 Byte: spectraTPPSDiagB3

This function enables the inversion of the path BIP-8 byte (B3) in the transmit stream. The B3 byte is inverted causing the insertion of eight path BIP-8 errors per frame.

Prototype `INT4 spectraTPPSDiagB3(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2 au3, UINT2 enable)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`enable`	: flag to start/stop B3 inversion

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Forcing a Pointer Value: spectraTPPSForceTxPtr

This function enables the insertion of the pointer value passed in argument into the H1 and H2 bytes of the transmit stream. As a result, the upstream payload mapping circuitry and a valid SPE can continue functioning and generating normally.

Prototype `INT4 spectraTPPSForceTxPtr(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2 au3, UINT2 enable, UINT2 aptr)`

Inputs

`deviceHandle`	: device Handle (from spectraAdd)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`enable`	: flag to start/stop generation
`aptr`	: pointer value to insert in (H1,H2)

Outputs None

Returns Success = `SPE_SUCCESS`

Failure = <SPECTRA-622 ERROR CODE>

Valid States ACTIVE, INACTIVE

Side Effects None

Writing the New Data Flag Bits: spectraTPPSInsertNDF

This function enables the insertion of the passed new data flag bits (NDF[3:0]) in the NDF bit positions.

Prototype `INT4 spectraTPPSInsertNDF(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2 au3, UINT2 enable, UINT1 ndf)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`enable`	: flag to start/stop NDF insertion
`ndf`	: NDF value

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <SPECTRA-622 ERROR CODE>

Valid States ACTIVE, INACTIVE

Side Effects None

Writing the Path Remote Error Indication Count: spectraTPPSInsertPREI

This function inserts the path remote error indication count passed in argument inside the path status byte.

Prototype `INT4 spectraTPPSInsertPREI(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2 au3, UINT1 PREI)`

Inputs	`deviceHandle`	: device Handle (from `spectraAdd`)
	`stm1`	: STM-1 index
	`au3`	: AU-3 index
	`prei`	: PREI value

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Forcing Errors in the H4 Byte: spectraTPPSDiagH4

This function enables the inversion of the multiframe indicator (H4) byte in the TRANSMIT stream. This forces an out of multiframe alarm in the downstream circuitry when the SPE (VC) is used to carry virtual tributary (VT) or tributary unit (TU) based payloads.

Prototype

```
INT4 spectraTPPSDiagH4(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2
au3, UINT2 enable)
```

Inputs	`deviceHandle`	: device Handle (from `spectraAdd`)
	`stm1`	: STM-1 index
	`au3`	: AU-3 index
	`enable`	: flag to start/stop H4 inversion

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Forcing Tributary Path AIS: spectraRPPSInsertTUAIS

This function enables the insertion of tributary path AIS in the transmit stream for VT1.5 (TU11), VT2 (TU12), VT3 and VT6 (TU2) payloads. Columns in the transmit stream carrying tributary traffic are set to all ones. The pointer bytes (H1, H2, and H3); the path overhead column; and the fixed stuff columns are unaffected. Note: This is not applicable for TU3 tributary payloads.

Prototype `INT4 spectraTPPSInsertTUAIS(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2 au3, UINT2 enable)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`enable`	: flag to start/stop TUAIS insertion

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Forcing DS3 AIS: spectraTPPSDs3AisGen

This function forces the generation of a DS3 AIS. Note: Any data on the STS-1 (STM-0/AU3) SPE is then lost.

Prototype `INT4 spectraTPPSDs3AisGen(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2 au3, UINT2 enable)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`enable`	: flag to start/stop DS3 AIS generation

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Writing the J1 Byte: spectraTPPSWriteJ1

This function writes the J1 byte into the transmit path overhead

Prototype

```
INT4 spectraTPPSWriteJ1(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2
au3, UINT1 J1)
```

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`J1`	: J1 byte to write

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Writing the C2 Byte: spectraTPPSWriteC2

This function writes the C2 byte into the transmit path overhead.

Prototype

```
INT4 spectraTPPSWriteC2(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2
au3, UINT1 C2)
```

Inputs	`deviceHandle`	: device Handle (from `spectraAdd`)
	`stm1`	: STM-1 index
	`au3`	: AU-3 index
	`C2`	: C2 byte to write

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Writing the F2 Byte: spectraTPPSWriteF2

This function writes the F2 byte into the transmit path overhead.

Prototype
```
INT4 spectraTPPSWriteF2(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2
au3, UINT1 F2)
```

Inputs	`deviceHandle`	: device Handle (from `spectraAdd`)
	`stm1`	: STM-1 index
	`au3`	: AU-3 index
	`F2`	: F2 byte to write

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Writing the Z3 Byte: spectraTPPSWriteZ3

This function writes the Z3 byte into the transmit path overhead.

Prototype `INT4 spectraTPPSWriteZ3(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2 au3, UINT1 Z3)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`Z3`	: Z3 byte to write

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Writing the Z4 Byte: spectraTPPSWriteZ4

This function writes the Z4 byte into the transmit path overhead.

Prototype `INT4 spectraTPPSWriteZ4(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2 au3, UINT1 Z4)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`Z4`	: Z4 byte to write

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Writing the Z5 Byte: spectraTPPSWriteZ5

This function writes the Z5 byte into the transmit path overhead.

Prototype `INT4 spectraTPPSWriteZ5(sSPE_HNDL deviceHandle UINT2 stm1, UINT2 au3, UINT1 Z5)`

Inputs

`deviceHandle`	: device Handle(from `spectraAdd`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`Z5`	: Z5 byte to write

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

5.14 Ring Control Ports (RING)

Sending Line AIS Maintenance Signal: spectraRINGLineAISControl

This function forces a mate SPECTRA-622 to send the line AIS maintenance signal.

Prototype `INT4 spectraRINGLineAISControl(sSPE_HNDL deviceHandle, UINT2 enable)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`enable`	: flag to start/stop Line-AIS insertion

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Sending Line RDI Maintenance Signal: spectraRINGLineRDIControl

This function forces a mate SPECTRA-622 to send the line RDI maintenance signal.

Prototype `INT4 spectraRINGLineRDIControl(sSPE_HNDL deviceHandle, UINT2 enable)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`enable`	: flag to start/stop Line-RDI insertion

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

5.15 WAN Synchronization Controller (WANS)

Forcing Phase Reacquisitions: spectraWANSForceReac

This function forces a phase reacquisition of the Phase Detector.

Prototype `INT4 spectraWANSForceReac(sSPE_HNDL deviceHandle)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

5.16 DROP Bus and ADD Bus PRBS Monitor and Generator (DPGM & APGM)

Configuring Diagnostics: spectraDiagCfg

This function configures the DPGM and APGM for diagnostics in accordance with the profile passed by the Application. Note: The DPGM and APGM are both disabled by default unless this function is called. A profile number of zero indicates a NULL profile. All register bits are left unchanged.

Prototype `INT4 spectraDiagCfg(sSPE_HNDL deviceHandle, UINT2 profileNum)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`profileNum`	: profile number

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects May insert a pseudo random byte sequence inside the payload.

5.17 DPGM Functions

Forcing Generation of a New PRBS: spectraDPGMGenRegen

This function reinitializes the generator LFSR and regenerates the pseudo random bit sequence (PRBS) from the known reset state. The LFSR is dependent on the sequence number. This automatically forces all slaves to reset at the same time.

Prototype `INT4 spectraDPGMGenRegen(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2 au3)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`stm1`	: STM-1 index
`au3`	: AU-3 index

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Forcing Bit Errors: spectraDPGMGenForceErr

This function forces bit errors in the inserted pseudo random bit sequence (PRBS). Thereafter, the MSB of the PRBS is inverted, inducing a single bit error.

Prototype `INT4 spectraDPGMGenForceErr(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2 au3)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`stm1`	: STM-1 index
`au3`	: AU-3 index

Outputs None

Returns Success = `SPE_SUCCESS`

Failure = <SPECTRA-622 ERROR CODE>

Valid States ACTIVE, INACTIVE

Side Effects None

Forcing a Resynchronization: spectraDPGMonResync

This function forces the resynchronization of the monitor to the incoming pseudo random bit sequence (PRBS). The monitor will go out of synchronization and begin re-synchronizing the incoming PRBS payload. This will automatically force all slaves to resynchronize at the same time.

Prototype `INT4 spectraDPGMMonResync(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2 au3)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`stm1`	: STM-1 index
`au3`	: AU-3 index

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <SPECTRA-622 ERROR CODE>

Valid States ACTIVE, INACTIVE

Side Effects None

5.18 APGM Functions

Forcing Generation of a New PRBS: spectraAPGMGenRegen

This function re-initializes the generator LFSR and begins regenerating the pseudo random bit sequence (PRBS) from the known reset state. The LFSR is dependent on the sequence number. This automatically forces all slave to reset at the same time.

Prototype	`INT4 spectraAPGMGenRegen(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2 au3)`	
Inputs	`deviceHandle`	: device Handle (from `spectraAdd`)
	`stm1`	: STM-1 index
	`au3`	: AU-3 index
Outputs	None	
Returns	Success = `SPE_SUCCESS` Failure = `<SPECTRA-622 ERROR CODE>`	
Valid States	ACTIVE, INACTIVE	
Side Effects	None	

Forcing Bit Errors: spectraAPGMGenForceErr

This function forces bit errors in the inserted pseudo random bit sequence (PRBS). Thereafter, the MSB of the PRBS is inverted, inducing a single bit error.

Prototype	`INT4 spectraAPGMGenForceErr(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2 au3)`	
Inputs	`deviceHandle`	: device Handle (from `spectraAdd`)
	`stm1`	: STM-1 index
	`au3`	: AU-3 index
Outputs	None	
Returns	Success = `SPE_SUCCESS`	

Failure = <SPECTRA-622 ERROR CODE>

Valid States ACTIVE, INACTIVE

Side Effects None

Forcing a Resynchronization: spectraAPGMonResync

This function forces resynchronization of the monitor to the incoming pseudo random bit sequence (PRBS). This process will automatically force all slaves to resynchronize at the same time.

Prototype `INT4 spectraAPGMMonResync(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2 au3)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`stm1`	: STM-1 index
`au3`	: AU-3 index

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <SPECTRA-622 ERROR CODE>

Valid States ACTIVE, INACTIVE

Side Effects None

5.19 Interrupt Service Functions

Getting the Interrupt Mask: spectraGetMask

This function returns the contents of the interrupt mask registers of the SPECTRA-622 device.

Prototype `INT4 spectraGetMask(sSPE_HNDL deviceHandle, sSPE_MASK *pmask)`

Inputs `deviceHandle` : device Handle (from `spectraAdd`)
`pmask` : (pointer to) mask structure

Outputs ERROR code written to the DDB

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Setting the Interrupt Mask: spectraSetMask

This function sets the contents of the interrupt mask registers of the SPECTRA-622 device.

Prototype `INT4 spectraSetMask(sSPE_HNDL deviceHandle, sSPE_MASK *pmask)`

Inputs `deviceHandle` : device Handle (from `spectraAdd`)
`pmask` : (pointer to) mask structure

Outputs ERROR code written to the DDB

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects May change the operation of the ISR / DPR

Clearing the Interrupt Mask: spectraClearMask

This function clears the individual interrupt bits and registers in the SPECTRA-622 device. Any bits that are set in the passed structure are cleared in the associated registers.

Prototype `INT4 spectraClearMask(sSPE_HNDL deviceHandle, sSPE_MASK *pmask)`

Inputs
`deviceHandle` : device Handle (from `spectraAdd`)
`pmask` : (pointer to) mask structure

Outputs ERROR code written to the DDB

Returns
Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects May change the operation of the ISR / DPR

Polling Interrupt Status Registers: spectraPoll

Commands the Driver to poll the interrupt registers in the Device. The call will fail unless the device is initialized in polling mode.

Prototype `INT4 spectraPoll(sSPE_HNDL deviceHandle)`

Inputs
`deviceHandle` : device Handle (from `spectraAdd`)

Outputs None

Returns
Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States `SPE_ACTIVE`

Side Effects None

Interrupt Service Routine: spectraISR

This function reads the state of the interrupt registers in the SPECTRA-622 and stores them into an ISV. It performs whatever functions are needed to clear the interrupt. This routine is called by the application code from within `sysSpectraISRHandler`.

Prototype	`void *spectraISR(sSPE_HNDL deviceHandle)`	
Inputs	`deviceHandle`	: device Handle (from `spectraAdd`)
Outputs	None	
Returns	(pointer to) ISV buffer (to send to the DPR) or NULL (pointer)	
Valid States	ACTIVE	
Side Effects	None	

Deferred Processing Routine: spectraDPR

This function acts on data contained in an ISV. It creates a DPV that invokes application code callbacks (if defined and enabled), and possibly other performing linked actions. This function is called from within the application function `sysSpectraDPRTask`.

Prototype	`void spectraDPR(sSPE_ISV *pisv)`	
Inputs	`pisv`	: (pointer to) ISV buffer
Outputs	None	
Returns	None	
Valid States	ACTIVE	
Side Effects	None	

5.20 Alarm, Status and Statistics Functions

Configuring Statistical Counts: spectraCfgStats

This function configures all the statistical counts.

Prototype `INT4 spectraCfgStats(sSPE_HNDL deviceHandle, sSPE_CFG_CNT cfgCnt)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`cfgCnt`	: counters configuration block

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Statistics Collection Routine: spectraGetCnt

This function retrieves all the device counts.Note: It is the USER's responsibility to ensure that the structure points to an area of memory large enough to hold a copy of the counter structure.

Prototype `INT4 spectraGetCnt(sSPE_HNDL deviceHandle, sSPE_STAT_CNT *pcnt)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`pcnt`	: (pointer to) allocated memory

Outputs

`pcnt`	: current device counts

Returns Success = `SPE_SUCCESS`

Failure = <SPECTRA-622 ERROR CODE>

Valid States ACTIVE, INACTIVE

Side Effects None

Retrieving Counter for SOP Block: spectraGetCntSOP

This function retrieves the specified device counts block.

Note: It is the USER's responsibility to ensure that the structure points to an area of memory large enough to hold a copy of the counter structure.

Prototype

```
INT4 spectraGetCntSOP(sSPE_HNDL deviceHandle,
sSPE_STAT_CNT_SOP *pcntSOP)
```

Inputs

deviceHandle	: device Handle (from spectraAdd)
pcntSOP	: (pointer to) allocated memory

Outputs

pcntSOP	: current device counts

Returns

Success = SPE_SUCCESS
Failure = <SPECTRA-622 ERROR CODE>

Valid States ACTIVE, INACTIVE

Side Effects None

Retrieving Counter for LOP Block: spectraGetCntLOP

This function retrieves the specified device counts block.

Note: It is the USER's responsibility to ensure that the structure points to an area of memory large enough to hold a copy of the counter structure.

Prototype
```
INT4 spectraGetCntLOP(SPE_HNDL deviceHandle,
sSPE_STAT_CNT_LOP *pcntLOP)
```

Inputs

`deviceHandle` : device Handle (from `spectraAdd`)
`pcntLOP` : (pointer to) allocated memory

Outputs

`pcntLOP` : current device counts

Returns

Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States `ACTIVE, INACTIVE`

Side Effects None

Retrieving Counter for RPPS Block: spectraGetCntRPPS

This function retrieves the specified device counts block.

Note: It is the USER's responsibility to ensure that the structure points to an area of memory large enough to hold a copy of the counter structure.

Prototype
```
INT4 spectraGetCntRPPS(sSPE_HNDL deviceHandle,
UINT2 stm1, UINT2 au3, sSPE_STAT_CNT_RPPS *pcntRPPS)
```

Inputs

`deviceHandle` : device Handle (from `spectraAdd`)
`stm1` : STM-1 index
`au3` : AU-3 index
`pcntRPPS` : (pointer to) allocated memory

Outputs

`pcntRPPS` : current device counts

Returns

Success = `SPE_SUCCESS`

Failure = <SPECTRA-622 ERROR CODE>

Valid States ACTIVE, INACTIVE

Side Effects None

Retrieving Counter for TPPS Block: spectraGetCntTPPS

This function retrieves the specified device counts block.

Note: It is the USER's responsibility to ensure that the structure points to an area of memory large enough to hold a copy of the counter structure.

Prototype

```
INT4 spectraGetCntTPPS(sSPE_HNDL deviceHandle,
UINT2 stm1, UINT2 au3, sSPE_STAT_CNT_TPPS *pcntTPPS)
```

Inputs

deviceHandle	: device Handle (from spectraAdd)
stm1	: STM-1 index
au3	: AU-3 index
pcntTPPS	: (pointer to) allocated memory

Outputs

pcntTPPS	: current device counts

Returns

Success = SPE_SUCCESS
Failure = <SPECTRA- 622 ERROR CODE>

Valid States ACTIVE, INACTIVE

Side Effects None

Retrieving Counter for Pointer Justifications: spectraGetCntPJ

This function retrieves the specified device counts block.

Note: It is the USER's responsibility to ensure that the structure points to an area of memory large enough to hold a copy of the counter structure.

Prototype `INT4 spectraGetCntTPPS(sSPE_HNDL deviceHandle, UINT2 stm1, UINT2 au3, sSPE_STAT_CNT_PJ *pcntPJ)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`pcntPJ`	: (pointer to) allocated memory

Outputs `pcntPJ` : current device counts

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States `ACTIVE, INACTIVE`

Side Effects None

Retrieving Alarm Status: spectraGetStatus

This function retrieves the current alarm status by reading all the alarm status registers.

Note: It is the USER's responsibility to ensure that the structure points to an area of memory large enough to hold a copy of the counter structure.

Prototype `INT4 spectraGetStatus(sSPE_HNDL deviceHandle, sSPE_STATUS *palm)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`palm`	: (pointer to) allocated memory

Outputs `palm` : current alarm status

Returns Success = `SPE_SUCCESS`

Failure = <SPECTRA-622 ERROR CODE>

Valid States ACTIVE, INACTIVE

Side Effects None

Retrieving Alarm Status for IO block: spectraGetStatusIO

This function reads a given alarm status from the alarm status registers.

Prototype
```
INT4 spectraGetStatusIO(sSPE_HNDL deviceHandle,
sSPE_STATUS_IO *palmIO)
```

Inputs

deviceHandle	: device Handle (from spectraAdd)
palmIO	: (pointer to) allocated memory

Outputs

palmIO	: current alarm status

Returns

Success = SPE_SUCCESS
Failure = <SPECTRA-622 ERROR CODE>

Valid States ACTIVE

Side Effects None

Retrieving Alarm Status for SOP block: spectraGetStatusSOP

This function reads a given alarm status from the alarm status registers.

Prototype
```
INT4 spectraGetStatusSOP(sSPE_HNDL deviceHandle,
sSPE_STATUS_SOP *palmSOP)
```

Inputs

deviceHandle	: device Handle (from spectraAdd)

	`palmSOP`	: (pointer to) allocated memory
Outputs	`palmSOP`	: current alarm status

Returns

Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States `ACTIVE`

Side Effects None

Retrieving Alarm Status for LOP block: spectraGetStatusLOP

This function reads a given alarm status from the alarm status registers.

Prototype

```
INT4 spectraGetStatusLOP(sSPE_HNDL deviceHandle,
sSPE_STATUS_LOP *palmLOP)
```

Inputs	`deviceHandle`	: device Handle (from `spectraAdd`)
	`palmLOP`	: (pointer to) allocated memory
Outputs	`palmLOP`	: current alarm status

Returns

Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States `ACTIVE`

Side Effects None

Retrieving Alarm Status for RPPS block: spectraGetStatusRPPS

This function reads a given alarm status from the alarm status registers.

Prototype `INT4 spectraGetStatusRPPSP(sSPE_HNDL deviceHandle, sSPE_STATUS_RPPS *palmRPPS)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`palmRPPS`	: (pointer to) allocated memory

Outputs

`palmRPPS`	: current alarm status

Returns Success = `SPE_SUCCESS`
Failure = < `SPECTRA-622 ERROR CODE`>

Valid States `ACTIVE`

Side Effects None

Retrieving Alarm Status for TPPS block: spectraGetStatusTPPS

This function reads a given alarm status from the alarm status registers.

Prototype `INT4 spectraGetStatusTPPS(sSPE_HNDL deviceHandle, sSPE_STATUS_TPPS *palmTPPS)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`palmTPPS`	: (pointer to) allocated memory

Outputs

`palmTPPS`	: current alarm status

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States `ACTIVE`

Side Effects None

5.21 Device Diagnostics

Verifying Register Access: spectraTestReg

This function verifies the hardware access to the device registers by writing and reading back values.

Prototype `INT4 spectraTestReg(sSPE_HNDL deviceHandle)`

Inputs `deviceHandle` : device Handle (from `spectraAdd`)

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States PRESENT

Side Effects None

Clearing and Setting a Line Loopback: spectraLoopLine

This function clears and sets a Line Loopback (SLLE=1). The spectraLoopLine connects the high speed receive data and clock to the high speed transmit data and clock, and can be used for line side investigations (including clock recovery and clock synthesis). While in this mode, the entire receive path is operating normally. Note: It is up to the USER to perform any tests on the looped data.

Prototype `INT4 spectraLoopLine(sSPE_HNDL deviceHandle, UINT2 enable)`

Inputs `deviceHandle` : device Handle (from `spectraAdd`)

	`enable`	: sets loop if non-zero, else clears loop

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE

Side Effects Will inhibit the flow of active data

Clearing and Setting a Serial Loopback: spectraLoopSerialDiag

This function clears and sets a Serial Diagnostic Loopback (SDLE=1). It connects the high speed transmit data and clock to the high speed receive data and clock. While in this mode, the entire transmit path is operating normally and data is transmitted on the TXD+/- outputs. Note: It is up to the USER to perform any tests on the looped data.

Prototype `INT4 spectraLoopSerialDiag(sSPE_HNDL deviceHandle, UINT2 enable)`

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`enable`	: sets loop if non-zero, else clears loop

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE

Side Effects Will inhibit the flow of active data

Clearing and Setting a Parallel Loopback: spectraLoopParaDiag

This function clears and sets a parallel diagnostic loopback (PDLE=1). It connects the byte wide transmit data and clock to the byte wide receive data and clock. While in this mode, the entire transmit path is operating normally and data is transmitted on the TXD+/- outputs. Note: It is up to the USER to perform any tests on the looped data.

Prototype INT4 spectraLoopParaDiag(sSPE_HNDL deviceHandle, UINT2 enable)

Inputs

	`deviceHandle`	: device Handle (from `spectraAdd`)
	`enable`	: sets loop if non-zero, else clears loop

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE

Side Effects Will inhibit the flow of active data

Clearing and Setting a System-Side Loopback: spectraLoopSysSideLine

This function clears and sets a system-side line loopback (SLLBEN=1). It connects the STS-1 (STM-0/AU3) or equivalent receive stream from the Receive Telecom bus Aligner (RTAL) of the associated RPPS to the Transmit Telecom bus Aligner (TTAL) of the corresponding TPPS. This mode can be used for line side investigations (including clock recovery and clock synthesis) as well as path processing investigations. While in this mode, the entire receive path is operating normally. The SPECTRA-622 may be configured to support the system-side line loopback of up to twelve STS-1 (STM-0/AU3) or equivalent receive streams. Note: It is up to the USER to perform any tests on the looped data.

Prototype `INT4 spectraLoopSysSideLine`(sSPE_HNDL `deviceHandle`, `UINT2 stm1`, `UINT2 au3`, `UINT2 enable`)

Inputs

	`deviceHandle`	: device Handle (from `spectraAdd`)
	`stm1`	: STM-1 index
	`au3`	: AU-3 index

`enable` : sets loop if non-zero, else clears loop

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE

Side Effects Will inhibit the flow of active data

Clearing and Setting a DS3 Line Loopback: spectraLoopDS3Line

This function clears and sets a DS3 line loopback (DS3LLBEN=1). It connects the DS3 receive stream from the DS3 Mapper DROP side (D3MD) of the associated RPPS to the DS3 Mapper ADD side (D3MA) of the corresponding TPPS. The DS3ADDSEL bit in the SPECTRA-622 TPPS Path and DS3 Configuration register of the TPPS must be set high. This mode can be used for line side investigations (including clock recovery and clock synthesis) as well as DS3 stream processing investigations. While in this mode, the entire receive (DS3) path is operating normally. The SPECTRA-622 may be configured to support the DS3 line loopback of up to twelve DS3 receive streams. Note: It is up to the USER to perform any tests on the looped data.

Prototype

```
INT4 spectraLoopDS3Line(sSPE_HNDL deviceHandle, UINT2
stm1, UINT2 au3, UINT2 enable)
```

Inputs

`deviceHandle`	: device Handle (from `spectraAdd`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`enable`	: sets loop if non-zero, else clears loop

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = <`SPECTRA-622 ERROR CODE`>

Valid States ACTIVE

Side Effects Will inhibit the flow of active data

5.22 Callback Functions

The SPECTRA-622 driver has the capability to callback to functions within the USER code when certain events occur. These events and their associated callback routine declarations are detailed below. There is no USER code action that is required by the driver for these callbacks – the USER is free to implement these callbacks in any manner or else they can be deleted from the driver.

The names given to the callback functions are given as examples only. The addresses of the callback functions invoked by the `spectraDPR` function are passed during the `spectraInit` call (inside a DIV). However the USER shall use the exact same prototype.

Note: The Application is left responsible for releasing the passed DPV as soon as possible (to avoid running out of DPV buffers) by calling `sysSpectraDPVBufferRtn` either within the callback function or later inside the Application code.

Callbacks Due to IO Events: cbackSpectraIO

This callback function is provided by the USER and is used by the DPR to report significant IO section events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements.

Note: The USER should free the DPV buffer.

Prototype `void cbackSpectraIO(sSPE_USR_CTXT usrCtxt, sSPE_DPV *pdpv)`

Inputs

	`usrCtxt`	: user context (from `spectraAdd`)
	`pdpv`	: (pointer to) formatted event buffer

Outputs None

Returns None

Valid States ACTIVE

Side Effects None

Callbacks Due to TOC Events: cbackSpectraTOC

This callback function is provided by the USER and is used by the DPR to report significant TOC section events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements.

Note: The USER should free the DPV buffer.

Prototype `void cbackSpectraTOC(sSPE_USR_CTXT usrCtxt, sSPE_DPV *pdpv)`

Inputs
`usrCtxt` : user context (from `spectraAdd`)
`pdpv` : (pointer to) formatted event buffer

Outputs None

Returns None

Valid States ACTIVE

Side Effects None

Callbacks Due to SOP Events: cbackSpectraSOP

This callback function is provided by the USER and is used by the DPR to report significant SOP section events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements.

Note: The USER should free the DPV buffer.

Prototype `void cbackSpectraSOP(sSPE_USR_CTXT usrCtxt, sSPE_DPV *pdpv)`

Inputs
`usrCtxt` : user context (from `spectraAdd`)
`pdpv` : (pointer to) formatted event buffer

Outputs None

Returns None

Valid States ACTIVE

Side Effects None

Callbacks Due to SSTB Events: cbackSpectraSSTB

This callback function is provided by the USER and is used by the DPR to report significant SSTB section events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements.

Note: The USER should free the DPV buffer.

Prototype `void cbackSpectraSSTB(sSPE_USR_CTXT usrCtxt, sSPE_DPV *pdpv)`

Inputs

`usrCtxt`	: user context (from `spectraAdd`)
`pdpv`	: (pointer to) formatted event buffer

Outputs None

Returns None

Valid States ACTIVE

Side Effects None

Callbacks Due to LOP Events: cbackSpectraLOP

This callback function is provided by the USER and is used by the DPR to report significant LOP section events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements.

Note: The USER should free the DPV buffer.

Prototype	`void cbackSpectraLOP(sSPE_USR_CTXT usrCtxt, sSPE_DPV *pdpv)`
Inputs	`usrCtxt` : user context (from `spectraAdd`) `pdpv` : (pointer to) formatted event buffer
Outputs	None
Returns	None
Valid States	ACTIVE
Side Effects	None

Callbacks Due to RPPS Events: cbackSpectraRPPS

This callback function is provided by the USER and is used by the DPR to report significant RPPS section events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements.

Note: The USER should free the DPV buffer.

Prototype	`void cbackSpectraRPPS(sSPE_USR_CTXT usrCtxt, sSPE_DPV *pdpv)`
Inputs	`usrCtxt` : user context (from `spectraAdd`) `pdpv` : (pointer to) formatted event buffer
Outputs	None
Returns	None
Valid States	ACTIVE
Side Effects	None

Callbacks due to TPPS events: cbackSpectraTPPS

This callback function is provided by the USER and is used by the DPR to report significant TPPS section events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements.

Note: The USER should free the DPV buffer.

Prototype	`void cbackSpectraTPPS(sSPE_USR_CTXT usrCtxt, sSPE_DPV *pdpv)`	
Inputs	`usrCtxt`	: user context (from `spectraAdd`)
	`pdpv`	: (pointer to) formatted event buffer
Outputs	None	
Returns	None	
Valid States	ACTIVE	
Side Effects	None	

Callbacks Due to WANS Events: cbackSpectraWANS

This callback function is provided by the USER and is used by the DPR to report significant WANS section events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements.

Note: The USER should free the DPV buffer.

Prototype	`void cbackSpectraWANS(sSPE_USR_CTXT usrCtxt, sSPE_DPV *pdpv)`	
Inputs	`usrCtxt`	: user context (from `spectraAdd`)
	`pdpv`	: (pointer to) formatted event buffer
Outputs	None	

Returns None

Valid States ACTIVE

Side Effects None

Callbacks Due to DPGM Events: cbackSpectraDPGM

This callback function is provided by the USER and is used by the DPR to report significant DPGM section events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements.

Note: The USER should free the DPV buffer.

Prototype `void cbackSpectraDPGM(sSPE_USR_CTXT usrCtxt, sSPE_DPV *pdpv)`

Inputs

`usrCtxt`	: user context (from `spectraAdd`)
`pdpv`	: (pointer to) formatted event buffer

Outputs None

Returns None

Valid States ACTIVE

Side Effects None

Callbacks Due to APGM Events: cbackSpectraAPGM

This callback function is provided by the USER and is used by the DPR to report significant APGM section events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements.

Note: the USER should free the DPV buffer.

Prototype	`void cbackSpectraAPGM(sSPE_USR_CTXT usrCtxt, sSPE_DPV *pdpv)`	
Inputs	`usrCtxt`	: user context (from `spectraAdd`)
	`pdpv`	: (pointer to) formatted event buffer
Outputs	None	
Returns	None	
Valid States	ACTIVE	
Side Effects	None	

6 HARDWARE INTERFACE

The SPECRTA-622 driver interfaces directly with the USER's hardware. In this section, a listing of each point of interface is shown, along with a declaration and any specific porting instructions. It is the responsibility of the USER to connect these requirements into the hardware, either by defining a macro or by writing a function for each item listed. Care should be taken when matching parameters and return values.

6.1 Device I/O

Reading Registers: sysSpectraRead

This function serves as the most basic hardware connection by reading the contents of a specific register location. This Macro should be UINT1 oriented, and should be defined by the user to reflect the target system's addressing logic. There is no need for error recovery in this function.

Prototype `UINT1 sysSpectraRead(UINT1 *addr)`

Inputs `addr` : register location to be read

Outputs None

Returns value read from the addressed register location

Format `#define sysSpectraRead(addr)`

Writing Values: sysSpectraWrite

This function serves as the most basic hardware connection by writing the supplied value to the specific register location. This macro should be UINT1 oriented and should be defined by the user to reflect the target system's addressing logic. There is no need for error recovery in this function.

Prototype `void sysSpectraWrite(UINT1 *addr, UINT value)`

Inputs `addr` : register location to be read

Outputs None

Returns value read from the addressed register location

Format `#define sysSpectraWrite(addr, value)`

6.2 Interrupt Servicing

The porting of an ISR routine between platforms is a rather difficult task. There are many different implementations of these hardware level routines. In this driver, the USER is responsible for installing an interrupt handler (`sysSpectraISRHandler`) in the interrupt vector table of the system processor. This handler shall call `spectraISR` for each device that has interrupt servicing enabled, to perform the ISR related housekeeping required by each device.

During execution of the API function `spectraModuleStart` / `spectraModuleStop` the driver informs the application that it is time to install / uninstall this shell via `sysSpectraISRHandlerInstall` / `sysSpectraISRHandlerRemove`, that needs to be supplied by the USER.

Note: A device can be initialized with ISR disabled. In that mode, the USER should periodically invoke a provided ‘polling’ routine (`spectraPoll`) that in turn calls `spectraISR`.

Installing the ISR Handler: sysSpectraISRHandlerInstall

This function installs the USER-supplied Interrupt Service Routine (ISR), `sysSpectraISRHandler`, into the processor’s interrupt vector table.

Prototype `void sysSpectraISRHandlerInstall(void *func)`

Inputs `func` : (pointer to) the function `spectraISR`

Outputs None

Returns None

Valid States None

Format `#define sysSpectraISRHandlerInstall(func)`

ISR Handler: sysSpectraISRHandler

This routine is invoked when one or more SPECTRA-622 devices raise the interrupt line to the microprocessor. This routine invokes the driver-provided routine (`spectraISR`) for each device registered with the driver.

Prototype `void sysSpectraISRHandler(void)`

Inputs None

Outputs None

Returns None

Format `#define sysSpectraISRHandler()`

Removing Handlers: sysSpectraISRHandlerRemove

This function disables Interrupt processing for this device. It removes the USER-supplied Interrupt Service routine (ISR), `sysSpectraISRHandler`, from the processor's interrupt vector table.

Prototype `void sysSpectraISRHandlerRemove(void)`

Inputs None

Outputs None

Returns None

Format `#define sysSpectraISRHandlerRemove()`

DPR Task: sysSpectraDPRTask

This routine is installed as a separate task within the RTOS. It runs periodically and retrieves the interrupt status information sent to it by the `spectraISRHandler` routine, thereafter invoking the `spectraDPR` routine for the appropriate device.

Prototype `void sysSpectraDPRTask(void)`

Inputs None

Outputs None

Returns None

Format `#define sysSpectraDPRTask()`

7 RTOS INTERFACE

The SPECTRA-622 driver requires the use of some RTOS resources. In this section, a listing of each required resource is shown, along with a declaration and any specific porting instructions. It is the responsibility of the USER to connect these requirements into the RTOS, either by defining a macro or writing a function for each item listed. Care should be taken when matching parameters and return values.

7.1 Memory Allocation / De-Allocation

Allocating Memory: sysSpectraMemAlloc

This function allocates specified number of bytes of memory.

Prototype `UINT1 *sysSpectraMemAlloc(UINT4 numBytes)`

Inputs `numBytes` : number of bytes to be allocated

Outputs None

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

Format `#define sysSpectraMemAlloc(numBytes)`

Freeing Memory: sysSpectraMemFree

This function frees the memory allocated when using the `sysSpectraMemAlloc`.

Prototype `void sysSpectraMemFree(UINT1 *pfirstByte)`

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

Outputs None

Returns None

Format `#define sysSpectraMemFree(pfirstByte)`

7.2 Buffer Management

All operating systems provide some sort of buffer system, particularly for use in sending and receiving messages. The following calls, provided by the USER, allow the Driver to Get and Return buffers from the RTOS. It is the USER’s responsibility to create any special resources or pools to handle buffers of these sizes during the `sysSpectraBufferStart` call.

Starting Buffer Management: sysSpectraBufferStart

This function alerts the RTOS that the ISV buffers and DPV buffers are available and should be sized correctly. This may or may not involve the creation of new buffer pools, depending on the RTOS.

Prototype `INT4 sysSpectraBufferStart(void)`

Inputs None

Outputs None

Returns Success = `SPE_SUCCESS`
Failure = `<SPECTRA-622 ERROR CODE>`

Format `#define sysSpectraBufferStart()`

Getting DPV Buffers: sysSpectraDPVBufferGet

This function retrieves a buffer from the RTOS. The buffer is used by the DPR code to create a Deferred Processing Vector (DPV). The DPV contains information about the state of the device. This information is passed on to the USER via a callback function.

Prototype `sSPE_DPV *sysSpectraDPVBufferGet(void)`

Inputs None

Outputs None

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

Format `#define sysSpectraDPVBufferGet()`

Getting ISV Buffers: sysSpectraISVBufferGet

This function retrieves a buffer from the RTOS. The buffer is used by the ISR code to create a Interrupt Service Vector (ISV). The ISV contains data transferred from the devices interrupt status registers.

Prototype `sSPE_ISV *sysSpectraISVBufferGet(void)`

Inputs None

Outputs None

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

Format `#define sysSpectraISVBufferGet()`

Returning DPV Buffers: sysSpectraDPVBufferRtn

This device returns a DPV buffer to the RTOS when the information in the block is no longer needed by the DPR.

Prototype `void sysSpectraDPVBufferRtn(sSPE_DPV *pdpv)`

Inputs `pdpv` : (pointer to) a DPV buffer

Outputs None

Returns None

Format `#define sysSpectraDPVBufferRtn(pdpv)`

Returning ISV Buffers: sysSpectraISVBufferRtn

This device returns a ISV buffer to the RTOS when the information in the block is no longer needed by the DPR.

Prototype `void sysSpectraISVBufferRtn(sSPE_ISV *pisv)`

Inputs `pisv` : (pointer to) a ISV buffer

Outputs None

Returns None

Format `#define sysSpectraISVBufferRtn(pisv)`

Stopping Buffer Management: sysSpectraBufferStop

This function alerts the RTOS that the Driver no longer needs the ISV buffers or DPV buffers. If any special resources were created to handle these buffers, they can be deleted at this time.

Prototype `void sysSpectraBufferStop(void)`

Inputs None

Outputs None

Returns None

Format `#define sysSpectraBufferStop()`

7.3 Preemption

Disabling Preemption: sysSpectraPreemptDisable

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

Prototype `INT4 sysSpectraPreemptDisable(void)`

Inputs None

Outputs None

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

Format `#define sysSpectraPreemptDisable()`

Re-Enabling Preemption: sysSpectraPreemptEnable

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

Prototype `void sysSpectraPreemptEnable(INT4 key)`

Inputs `key` : preemption key (returned by `sysSpectraPreemptDisable`)

Outputs None

Returns None

Format #define `sysSpectraPreemptEnable(key)`

7.4 Timers

Suspending a Task Execution: sysSpectraTimerSleep

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

Prototype `void sysSpectraTimerSleep(UINT4 msec)`

Inputs `msec` : sleep time in milliseconds

Outputs None

Returns None

Format `#define sysSpectraTimerSleep(msec)`

8 PORTING DRIVERS

This section outlines how to port the SPECTRA-622 device driver to your hardware and OS platform. However, this manual can offer only guidelines for porting the SPECTRA-622 driver because each platform and application is unique.

8.1 Driver Source Files

The C files listed in the following table contain the code for the SPECTRA-622 driver. You may need to modify the code or develop additional code. The code is in the form of constants, macros, and functions. For the ease of porting, the code is grouped into source files (`src`) and include files (`inc`). The source files contain the functions and the include files contain the structures, constants and macros.

Directory	File	Description
`src`	`spe_api1.c`	All the API functions that take care of module, device and profile management
	`spe_api2.c`	All the SPECTRA-622 specific API functions.
	`spe_hw.c`	Hardware interface functions
	`spe_isr.c`	Internal functions that deal with interrupt servicing
	`spe_prof.c`	Internal functions that deal with profiles
	`spe_rtos.c`	RTOS interface functions
	`spe_stat.c`	Internal functions that deal with statistics
	`spe_util.c`	All the remaining internal functions
`inc`	`spe_api.h`	All API headers
	`spe_defs.h`	Driver macros, constants and definitions (such as register mapping and bit masks)
	`spe_err.h`	SPECTRA-622 error codes
	`spe_fns.h`	Prototype of non-API functions
	`spe_hw.h`	HW interface macros and prototype
	`spe_rtos.h`	RTOS interface macros and prototypes
	`spe_strs.h`	driver structures
	`spe_typs.h`	types definitions
`example`	`spe_app.c`	Sample driver callback functions and example code
	`spe_app.h`	Prototypes, macros and structures used inside the example code

8.2 Driver Porting Procedures

The following procedures summarize how to port the SPECTRA-622 driver to your platform. The subsequent sections describe these procedures in more detail.

To port the SPECTRA-622 driver to your platform:

Step 1: Port the driver's RTOS interface (page 127):

Step 2: Port the driver's hardware interface (page 128):

Step 3: Port the driver's application-specific elements (page 130):

Step 4: Build the driver (page 130).

Porting Assumptions

The following porting assumptions have been made:

- It is assumed that ram assigned to the Driver's static variables is initialized to ZERO before any Driver function is called.
- It is assumed that a ram stack of 4K is available to all of the Driver's non-ISR functions and that a ram stack of 1K is available to the Driver's ISR functions.
- It is assumed that there is no memory management or MMU in the system or that all accesses by the driver, to memory or hardware can be direct.

Step 1: 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 OS extensions:

1. Redefine the following macros and functions in the `spe_rtos.h` file to the corresponding system calls that your target system supports:

Service Type	Macro Name	Description
Memory	`sysSpectraMemAlloc`	Allocates a memory block
	`sysSpectraMemFree`	Frees a memory block
	`sysSpectraMemCpy`	Copies the contents of one memory block to another
	`sysSpectraMemSet`	Fills a memory block with a specified value

Timer	`sysSpectraTimerSleep`	Delays the task execution for a given number of milliseconds
Pre-emption Lock/Unlock	`sysSpectraPreemptDisable`	Disables pre-emption of the currently executing task by any other task or interrupt
	`sysSpectraPreemptEnable`	Re-enables pre-emption of a task by other tasks and/or interrupts

2. Modify the example implementation of the buffer management routines provided in the `spe_rtos.h` file with the corresponding system calls that your target system supports:

Service Type	Macro Name	Description
Buffer	`sysSpectraBufferStart`	Starts buffer management
	`sysSpectraBufferStop`	Stops buffer management
	`sysSpectraISVBufferGet`	Gets an ISV buffer from the ISV buffer queue
	`sysSpectraISVBufferRtn`	Returns an ISV buffer to the ISV buffer queue
	`sysSpectraDPVBufferGet`	Gets a DPV buffer from the DPV buffer queue
	`sysSpectraDPVBufferRtn`	Returns a DPV buffer to the DPV buffer queue

3. Define the following constants for your OS-specific services in `spe_rtos.h`:

Task Constant	Description	Default
`SPE_DPR_TASK_PRIORITY`	Deferred Task (DPR) task priority	85
`SPE_DPR_TASK_STACK_SZ`	DPR task stack size, in bytes	8192
`SPE_MAX_ISV_BUF`	The queue message depth of the queue used for pass interrupt context between the ISR task and DPR task	50
`SPE_MAX_DPV_BUF`	The queue message depth of the queue used for pass interrupt context between the ISR task and DPR task	950

Step 2: Porting the Hardware Interface

This section describes how to modify the SPECTRA-622 driver for your hardware platform.

To port the driver to your hardware platform:

1. Modify the variable type definitions in `spe_typs.h.`
2. Modify the low-level hardware-dependent functions and macros in the `spe_hw.h` file. You may need to modify the raw read/write access macros (`sysSpectraRead` and `sysSpectraWrite`) to reflect your system's addressing logic.

Service Type	Function Name	Description
Register Access	sysSpectraRead	Reads a device register given its real address in memory
	sysSpectraWrite	Writes to a device register given its real address in memory
Interrupt	sysSpectraISRHandlerInstall	Installs the interrupt handler for the OS
	sysSpectraISRHandlerRemove	Removes the interrupt handler from the OS
	sysSpectraISRHandler	Interrupt handler for the SPECTRA-622 device
	sysSpectraDPRTask	Task that calls the SPECTRA-622 DPR

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

Device Constant	Description	Default
SPE_MAX_DEVS	The maximum number of SPECTRA-622 devices that can be supported by the driver	5
SPE_MAX_DELAY	Delay between two consecutive polls of a busy bit	100us
SPE_MAX_POLL	Maximum number of times a busy bit will be polled before the operation times out	100

Step 3: Porting the Application-Specific Elements

Porting the application-specific elements includes coding the application callback and defining all the constants used by the API.

To port the driver's application-specific elements:

1. Modify the base value of `SPE_ERR_BASE` (default = -300) in `spe_err.h`.

2. Define the following constants as required by your application in `spe_rtos.h`:

Task Constant	Description	Default
SPE_MAX_INIT_PROFS	The maximum number of initialization profiles that can be added to the driver	5
SPE_MAX_DIAG_PROFS	The maximum number of diagnostic profiles that can be added to the driver	5

3. Code the callback functions according to your application. Example implementations of these callbacks are provided in `app.c`. The driver will call these callback functions when an event occurs on the device. These functions must conform to the following prototype:
```
void cbackXX(sSPE_USR_CTXT usrCtxt, void *pdpv)
```

Step 4: Building the Driver

This section describes how to build the SPECTRA-622 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 SPECTRA-622 API driver library using your make utility.

4. Link the SPECTRA-622 API driver library to your application code.

APPENDIX A: DRIVER RETURN CODES

Table 25 describes the driver's return codes.

Table 25: Return Codes

Return Type	Description
SPE_ERR_MEM_ALLOC	Memory allocation failure
SPE_ERR_INVALID_ARG	Invalid argument
SPE_ERR_INVALID_MODULE_STATE	Invalid Module state
SPE_ERR_INVALID_MIV	Invalid Module Initialization Vector
SPE_ERR_PROFILES_FULL	Maximum number of profiles already added
SPE_ERR_INVALID_PROFILE	Invalid profile
SPE_ERR_INVALID_PROFILE_MODE	Invalid profile mode selected
SPE_ERR_INVALID_PROFILE_NUM	Invalid profile number
SPE_ERR_INVALID_DEVICE_STATE	Invalid Device state
SPE_ERR_DEVS_FULL	Maximum number of devices already added
SPE_ERR_DEV_ALREADY_ADDED	Device already added
SPE_ERR_INVALID_DEV	Invalid device handle
SPE_ERR_INVALID_DIV	Invalid Device Initialization Vector
SPE_ERR_INT_INSTALL	Error while installing interrupts
SPE_ERR_INVALID_MODE	Invalid ISR/polling mode
SPE_ERR_INVALID_REG	Invalid register number
SPE_ERR_POLL_TIMEOUT	Time-out while polling

APPENDIX B: CODING CONVENTIONS

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

Variable Type Definitions

Table 26: 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

Naming Conventions

Table 27 presents a summary of the naming conventions followed by all PMC driver software. A detailed description is then given in the following sub-sections.

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

Table 27: Naming Conventions

Type	Case	Naming convention	Examples
Macros	Uppercase	prefix with "m" and device abbreviation	mSPE_SLICE_OFFSET
Constants	Uppercase	prefix with device abbreviation	SPE_MAX_REGS
Structures	Hungarian Notation	prefix with "s" and device abbreviation	sSPE_DDB

Type	Case	Naming convention	Examples
API Functions	Hungarian Notation	prefix with device name	`spectraAdd()`
Porting Functions	Hungarian Notation	prefix with “sys” and device name	`sysSpectraRead()`
Other Functions	Hungarian Notation		`myOwnFunction()`
Variables	Hungarian Notation		`maxDevs`
Pointers to variables	Hungarian Notation	prefix variable name with “p”	`pmaxDevs`
Global variables	Hungarian Notation	prefix with device name	`spectraMdb`

Macros

The following list identifies the marcro conventions used in the driver code:

- Macro names can be uppercase.
- Words can be separated by an underscore.
- The letter ‘m’ in lowercase is used as a prefix to specify that it is a macro, then the device abbreviation appears.
- Example: `mSPE_SLICE_OFFSET` is a valid name for a macro.

Constants

The following list identifies the constants conventions used in the driver code:

- Constant names can be uppercase.
- Words can be separated by an underscore.
- The device abbreviation can appear as a prefix.
- Example: `SPE_MAX_REGS` is a valid name for a constant.

Structures

The following list identifies the structures conventions used in the driver code:

- Structure names can be uppercase.
- Words can be separated by an underscore.

- The letter ‘s’ in lowercase can be used as a prefix to specify that it is a structure, then the device abbreviation appears.
- Example: `sSPE_DDB` is a valid name for a structure.

Functions

API Functions

- Naming of the API functions follows the hungarian notation.
- The device’s full name in all lowercase can be used as a prefix.
- Example: `spectraAdd()` is a valid name for an API function.

Porting Functions

Porting functions correspond to all function that are HW and/or RTOS dependant.

- Naming of the porting functions follows the hungarian notation.
- The ‘sys’ prefix can be used to indicate a porting function.
- The device’s name starting with an uppercase can follow the prefix.
- Example: `sysSpectraRead()` is a hardware / RTOS specific.

Other Functions

- Other Functions are all the remaining functions that are part of the driver and have no special naming convention. However, they can follow the hungarian notation.
- Example: `myOwnFunction()` is a valid name for such a function.

Variables

- Naming of variables follows the hungarian notation.
- A pointer to a variable shall use ‘p’ as a prefix followed by the variable name unchanged. If the variable name already starts with a ‘p’, the first letter of the variable name may be capitalized, but this is not a requirement. Double pointers might be prefixed with ‘pp’, but this is not required.
- Global variables are identified with the device’s name in all lowercase as a prefix.
- Examples: `maxDevs` is a valid name for a variable, `pmaxDevs` is a valid name for a pointer to `maxDevs`, and `spectraMdb` is a valid name for a global variable.
- Note: Both `pprevBuf` and `pPrevBuf` are accepted names for a pointer to the `prevBuf` variable, and that both pmatrix and ppmatrix are accepted names for a double pointer to the variable matrix.

File Organization

Table 28 presents a summary of the file naming conventions. All file names must start with the device abbreviation, followed by an underscore and the actual file name. File names should convey their purpose with a minimum amount of characters. If a file size is getting too big one might separate it into two or more files, providing that a number is added at the end of the file name (e.g. `spe_api1.c` or `spe_api2.c`).

There are 4 different types of files:

- The API file containing all the API functions
- The hardware file containing the hardware dependant functions
- The RTOS file containing the RTOS dependant functions
- The other files containing all the remaining functions of the driver

Table 28: File Naming Conventions

File Type	File Name
API	spe_api1.c, spe_api.h
Hardware Dependant	spe_hw.c, spe_hw.h
RTOS Dependant	spe_rtos.c, spe_rtos.h
Other	spe_isr.c, spe_defs.h

API Files

- The name of the API files must start with the device abbreviation followed by an underscore and ‘`api`’. Eventually a number might be added at the end of the name.
- Examples: `spe_api1.c` is the only valid name for the file that contains the first part of the API functions, `spe_api.h` is the only valid name for the file that contains all of the API functions headers.

Hardware Dependent Files

- The name of the hardware dependent files must start with the device abbreviation followed by an underscore and ‘`hw`’. Eventually a number might be added at the end of the file name.
- Examples: `spe_hw.c` is the only valid name for the file that contains all of the hardware dependent functions, `spe_hw.h` is the only valid name for the file that contains all of the hardware dependent functions headers.
- RTOS Dependant Files

- The name of the RTOS dependant files must start with the device abbreviation followed by an underscore and `‘rtos’`. Eventually a number might be added at the end of the file name.
- Examples: `spe_rtos.c` is the only valid name for the file that contains all of the RTOS dependent functions, `spe_rtos.h` is the only valid name for the file that contains all of the RTOS dependent functions headers.

Other Driver Files

- The name of the remaining driver files must start with the device abbreviation followed by an underscore and the file name itself, which should convey the purpose of the functions within that file with a minimum amount of characters.
- Examples: `spe_isr.c` is a valid name for a file that would deal with interrupt servicing, `spe_defs.h` is a valid name for the header file that conatins all the driver’s definitions.

LIST OF TERMS

APPLICATION: Refers to protocol software used in a real system as well as validation software written to validate the SPECTRA-622 driver on a validation platform.

API (Application Programming Interface): Describes the connection between this MODULE and the USER's Application code.

ISR (Interrupt Service Routine): A common function for intercepting and servicing DEVICE events. This function is kept as short as possible because an Interrupt preempts every other function starting the moment it occurs and gives the service function the highest priority while running. Data is collected, Interrupt indicators are cleared, and the function ended.

DPR (Deferred Processing Routine): This function is installed as a task, at a USER configurable priority, that serves as the next logical step in Interrupt processing. Data that was collected by the ISR is analyzed and then calls are made into the Application to inform it of the events that caused the ISR in the first place. Because this function is operating at the task level, the USER can decide on its importance in the system, relative to other functions.

DEVICE: A single SPECTRA-622 Integrated Circuit. There can be many Devices; all served by this ONE Driver MODULE

- DIV (DEVICE Initialization Vector): A structure passed from the API to the DEVICE during initialization; it contains parameters that identify the specific modes and arrangements of the physical DEVICE being initialized.
- DDB (DEVICE Data Block): A structure that holds the Configuration Data for each DEVICE.

MODULE: All of the code that is part of this driver; there is only ONE instance of this MODULE connected to one or more SPECTRA-622 chips.

- MIV (MODULE Initialization Vector): Structure passed from the API to the MODULE during initialization; it contains parameters that identify the specific characteristics of the Driver MODULE being initialized.
- MDB (MODULE Data Block): A structure that holds the Configuration Data for this MODULE.

RTOS (Real Time Operating System): The host for this driver

ACRONYMS

API: Application programming interface

APGM: Add bus PRBS Generator and Monitor

DDB: Device data block

DIV: Device initialization vector

DPGM: Drop bus PRBS Generator and Monitor

DPR: Deferred processing routine

DPV: Deferred processing (routine) vector

FIFO: First in, first out

IO: Input/Output

ISR: Interrupt service routine

ISV: Initialization service (routine) vector

LOP: Line overhead processor

MDB: Module data block

MIV: Module initialization vector

PRBS: Pseudo random byte sequence

RING: RING control ports

RPPS: Receive path processing slice

RTOS: Real-time operating system

SOP: Section overhead processor

SSTB: Sonet/SDH section trace buffer

TOC: Transport overhead controller

TPPS: Receive path processing slice

WANS: WAN synchronization controller

INDEX

T

U