PM5316

SPECTRA-4x155 DEVICE DRIVER

DRIVER MANUAL

PROPRIETARY AND CONFIDENTIAL

PRELIMINARY

ISSUE 1: JANUARY, 2001

ABOUT THIS MANUAL AND SPECTRA-4X155

This manual describes the SPECTRA-4x155 (PM5316) device driver. It describes the driver's functions, data structures, and architecture. This manual focuses on the driver's interfaces and their relationship to your application, real-time operating system, and to the device. It also describes in general terms how to modify and port the driver to your software and hardware platform.

Audience

This manual was written for people who need to:

- Evaluate and test the SPECTRA-4x155 devices
- Modify and add to the SPECTRA-4x155 driver's functions
- Port the SPECTRA-4x155 driver to a particular platform.

References

For more information about the SPECTRA-4x155 driver, see the driver's Release Notes.(PMC-2001967) For more information about the SPECTRA-4x155 device, see the documents listed in Table 1 and any related errata documents.

Table 1: Related Documents

Document Number	Document Name
PMC-1990822	PM5316 SPECTRA-4X155 Telecom Standard Product Data Sheet

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	January 2001	Document created

Legal Issues

PMC-2001693 (P1), ref PMC-2000216 (P2)

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 INTRODUCTION

The following sections of the SPECTRA-4x155 Driver Manual describe the SPECTRA-4x155 device driver. The code provided throughout this document is written in the C language. This has been done to promote greater driver portability to other embedded hardware (Section 6) and Real-Time Operating System environments (Section 7).

Section 3 of this document, Software Architecture, defines the software architecture of the SPECTRA-4x155 device driver by including a discussion of the driver's external interfaces and its main components. The Data Structure information in Section 4 describes the elements of the driver that either configure or control its behavior. Included here are the constants, variables, and structures that the SPECTRA-4x155 device driver uses to store initialization, configuration, and status information. Section 5 provides a detailed description of each function that is a member of the SPECTRA-4x155 driver Application Programming Interface (API). This section outlines: (1) function calls that hide device-specific details and (2) application callbacks that notify the user of significant device events.

For your convenience, Section 8 of this manual provides a brief guide to porting the SPECTRA-4x155 device driver to your hardware and RTOS platform. In addition, an extensive Appendix (page 129) and Index (page 144) provides you with useful reference information.

2 DRIVER FUNCTIONS AND FEATURES

This section describes the main functions and features supported by the SPECTRA-4x155 driver.

2.1 General Driver Functions

Open / Close Driver Module

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

Starting the driver module involves allocating all Real-Time Operating System (RTOS) resources needed by the driver, such as timers, as well as installing the Interrupt-Service Routine (ISR) and spawning the Deferred-Processing Routine (DPR) task (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

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

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-4x155 device.

Activate / De-Activate Device

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 Application Programming Interface (API) invocations.

De-activating a device removes it from its operating state, disables interrupts and other global registers.

Read / Write Device Registers

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

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

Functions are provided to retrieve a snapshot of the various counts that are accumulated by the SPECTRA-4x155 device.

2.2 SPECTRA-4x155 Specific Driver Functions

These functions provide control and monitoring of the various sections of the SPECTRA-4x155 device. These device sections are generally enabled or disabled and configured by the mode specified during device initialization. Changes to device registers that would violate the characteristics of the initialized mode should be disallowed in application software.

Input / Output Interfaces

Functions will be provided to configure the following Input / Output interfaces:

- Configuration of the RCLK, TCLK, PGMRCLK, and PGMTCLK clocks.
- System side telecom ADD/DROP bus and timeslot interchange (TSI)
- Ring control port activation and alarm insertion.

- STS-3 or STS-3c mode on a per-155Mbps channel basis.

Section Overhead

The following functions will be provided to control the Section Overhead (SOH) processing:

- Enable/Disable section overhead termination
- Read/write the section trace message (J0).
- Read/reset the SOH BIP-8 error counter (B1).
- Detect and report the following alarm conditions: loss of frame (LOF), out of frame (OOF), loss of signal (LOS), trace identifier unstable (TIU), and trace identifier mismatch (TIM).

Line Overhead

The following functions will be provided to configure and monitor the Line Overhead processing:

- Enable/Disable line overhead termination
- Read/write the automatic protection switch (APS) bytes (K1,K2)
- Read/reset the LOH BIP-8 error counter (B2).
- Control automatic line alarm indication signal (AIS) insertion.
- Control the automatic insertion of remote error indications (REI) into the Z2 byte.
- Read/reset the incoming LOH REI counter (Z2).
- Control remote defect indication (RDI) transmission.
- Read the received synchronization byte (Z1).
- Configure signal fail (SF) and signal degrade (SD) accumulation period and threshold parameters.
- Detect and report LRDI, LAIS, APS, SF, and SD errors.

Path Overhead

The following functions will be provided to configure and monitor the Path Overhead processing:

- Enable/Disable path overhead termination
- Read/write the path trace message (J1).
- Read/write the path signal label (C2).
- Read/reset the BIP-8 error counter (B3).
- Detects and reports Loss of Pointer (LOP), loss of tributary multiframe (LOM), path alarm indication signal (PAIS), and path remote defect indication (RDI).
- Control automatic insertion of PAIS and RDI into the transmit stream.

- Control automatic insertion of the remote error indication (REI) count into the path status byte (G1).
- Read the tandem connection maintenance byte (Z5).
- PRBS generation and monitoring.

Device Update

A function is provided to update the device's configuration without forcing a hardware reset.

Device Diagnostics

Functions to perform device diagnostics.

- Device register read/write test.
- Line side diagnostic loopback between the receive and transmit clock and data interfaces.
- Parallel diagnostic loopback (between RSOP and TSOP blocks).
- Serial diagnostic loopback (between RXD+/- and TXD +/-).
- System side telecom bus interface loopback on an STS-1 basis.

Statistics and Alarm Monitoring

Functions to gather statistics and do alarm monitoring.

- Retrieve a snapshot of the various overhead bytes.
- Retrieve a snapshot of the current alarm status for the various SONET/SDH alarms.
- Read/clear the accumulated error counts.

Specific Callback Functions

Callback functions are available to the application for event notification from the device driver. Application will be notified via the callback functions for selected events of interest such as:

- Detection of I/O alarm conditions.
- Detection of Section Overhead alarm conditions.
- Detection of Line Overhead alarm conditions.
- Detection of Path Overhead alarm conditions.

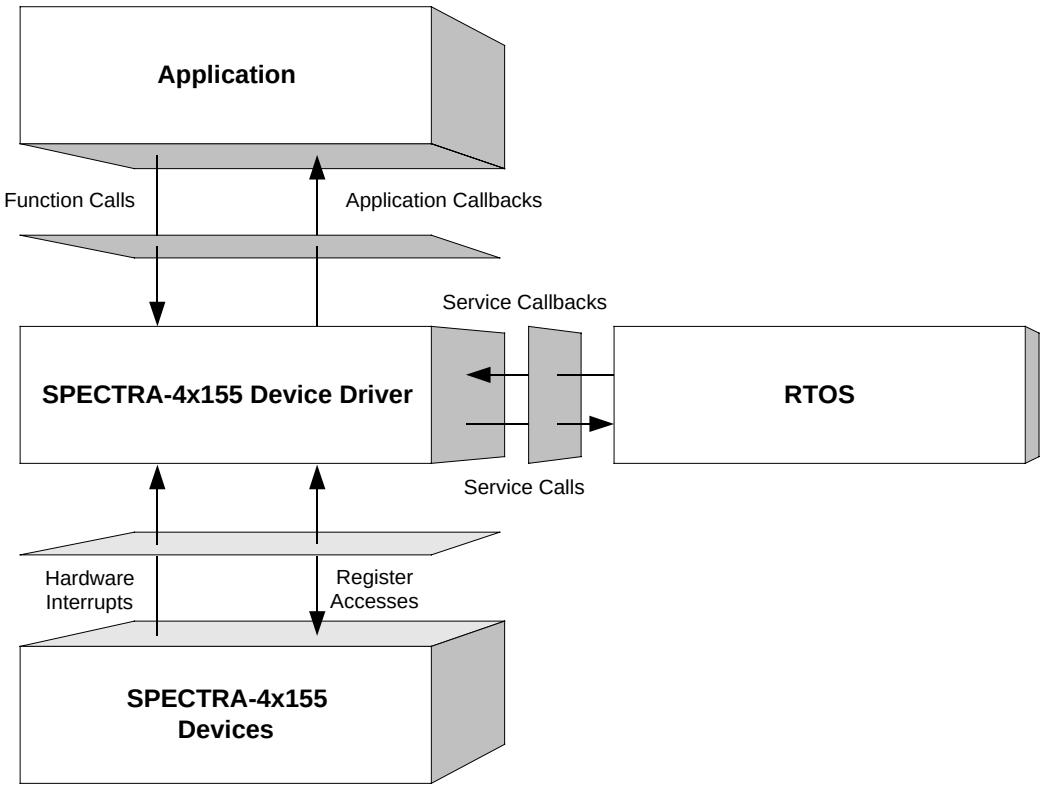
3 SOFTWARE ARCHITECTURE

This section describes the software architecture of the SPECTRA-4x155 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-4x155 device driver.

Figure 1: Driver External Interfaces



Application Programming Interface

The driver Application Programming Interface (API) is a list of high-level functions that can be invoked by application programmers to configure, control and monitor SPECTRA-4x155 devices. The API functions perform operations that are more meaningful from a system's perspective. The API includes functions that:

- Initialize the device(s)

- Perform diagnostic tests
- Validate configuration information
- Retrieve status and statistics information.

The driver API functions use the services of the other driver components to provide this system-level functionality to the application programmer.

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

Real-Time OS Interface

The driver's Real-Time Operating System (RTOS) interface provides functions that let the driver use RTOS services. The driver requires the memory, interrupt, and preemption services from the RTOS. The RTOS interface functions perform the following tasks for the driver:

- Allocate and de-allocate memory
- Manage buffers for the Interrupt-Service Routine (ISR) and the Deferred-Processing Routine (DPR)
- Enable and disable preemption

The RTOS interface also includes service callbacks. These are functions installed by the driver using RTOS service calls such as installing interrupts. These service callbacks are invoked when an interrupt occurs.

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

Hardware Interface

The hardware interface provides functions that read from and write to the 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

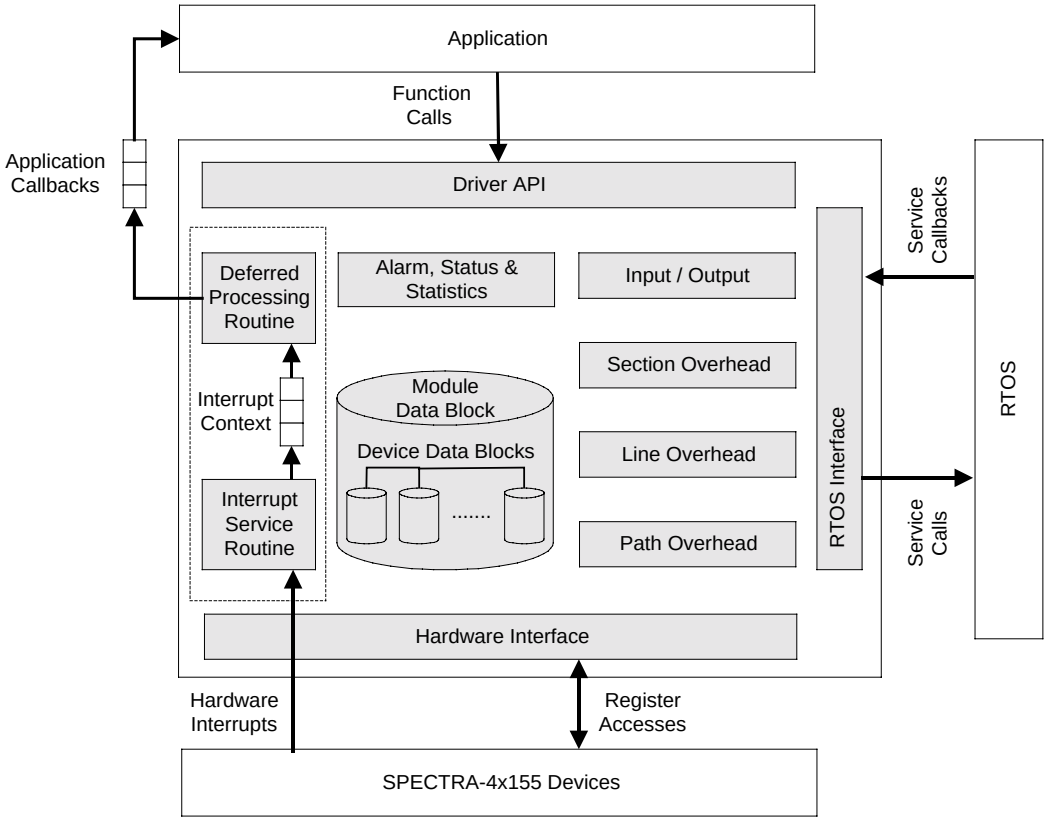
Figure 2 illustrates the top-level architectural components of the SPECTRA-4x155 device driver. This applies in both polled and interrupt driven operation. In polled operation the ISR is called periodically. In interrupt operation the interrupt directly triggers the ISR.

The driver includes eight main components:

- Module and device(s) data-blocks
- Interrupt-Service Routine

- Deferred-Processing Routine
- Alarm, status and statistics
- Input / Output
- Section Overhead
- Line Overhead
- Path Overhead

Figure 2: Driver Architecture



Module Data-Block and Device(s) Data-Blocks

The Module Data-Block (MDB) is the top layer data structure, created by the SPECTRA-4x155 driver to store context information about the driver module, such as:

- Module state
- Maximum number of devices
- The DDB(s)

The Device Data-Block (DDB) is contained in the MDB, and initialized by the driver module for each SPECTRA-4x155 device that is registered. There is one DDB per device and there is a limit on the number of DDBs, and that limit is set by the user when the module is initialized. The DDB is used to store context information about one device, such as:

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

Interrupt-Service Routine

The SPECTRA-4x155 driver provides an ISR called `spe4x155ISR` that checks if there is any valid interrupt condition 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 `spe4x155ISR` 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 in section 6.2. You can customize these example implementations to suit your specific needs.

See section 3.5 for a detailed explanation of the ISR and interrupt-servicing model.

Deferred-Processing Routine

The SPECTRA-4x155 driver provides a DPR called `spe4x155DPR` that processes any interrupt condition gathered by the ISR for that device. Typically, a system specific function, which runs as a separate task within the RTOS, will call `spe4x155DPR`.

Example implementations of a DPR task and functions that install and remove it are provided as a reference in section 7.5. You can customize these example implementations to suit your specific needs.

See section 3.5 for a detailed explanation of the DPR and interrupt-servicing model.

Alarms, Status and Statistics

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

Input / Output

The Input / Output component is responsible for configuring the line side and system side device interfaces. On the line-side, functions are provided to control the four 155Mbps clock and data interfaces and ring control port. On the system-side, functions are provided to control the Add/Drop Telecom Bus data interfaces and the Time-Slot Interchange (TSI). As well, this component configures STS-3 or STS-3c mode for each of the four channels and reports mismatches between incoming STS-3/STS-3c streams and the channel configuration.

Section Overhead

The Section Overhead component provides functions to control and monitor the Section Overhead processing for each STS-3 stream. Read/Write access is given to the section trace message (J0). This message is compared with a configurable reference and mismatches are reported. As well, section BIP-8 (B1) errors are accumulated in a counter that can be read and cleared. Section Overhead alarms are detected and reported. For diagnostic purposes, errors can be introduced in the Section Overhead bytes.

Line Overhead

The Line Overhead component is responsible for configuring and monitoring the Line Overhead on both receive and transmit sides for each STS-3. Read/write access is given to the APS bytes (K1 and K2) and most of the other overhead bytes. Line BIP-8 (B2) errors and REI indications are accumulated in counters that can be read and cleared. Line Overhead alarms are detected and reported. For diagnostic purposes, errors can be introduced in the Line Overhead bytes. Additional functions are provided to configure the device to automatically insert line RDI and AIS.

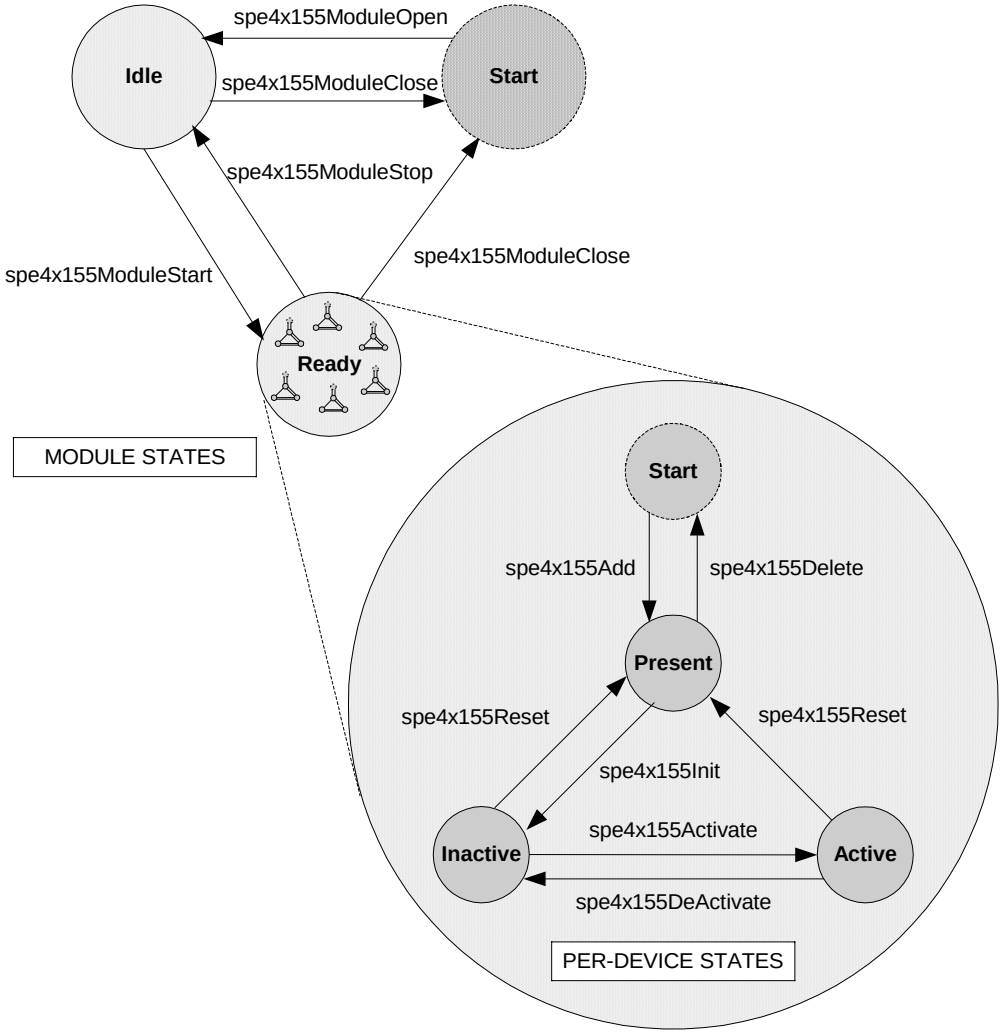
Path Overhead

The Path Overhead component configures and monitors the Path Overhead on both receive and transmit sides for each STS-3. Read/write access is given to the path trace message (J1) and the path signal label (C2). Both are compared with a configurable reference and mismatches are reported. Path BIP-8 (B3) errors and REI are accumulated in a counter that can be read and cleared. Path Overhead alarms are detected and reported. For diagnostic purposes, errors can be introduced in the Path Overhead bytes. Additional functions are provided to configure the device to automatically insert path RDI, path enhanced RSI, and path AIS.

3.3 Software States

Figure 3 shows the software state diagram for the SPECTRA-4x155 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-4x155 module states. See section 5.1 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); it 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-4x155 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 section 5.3 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

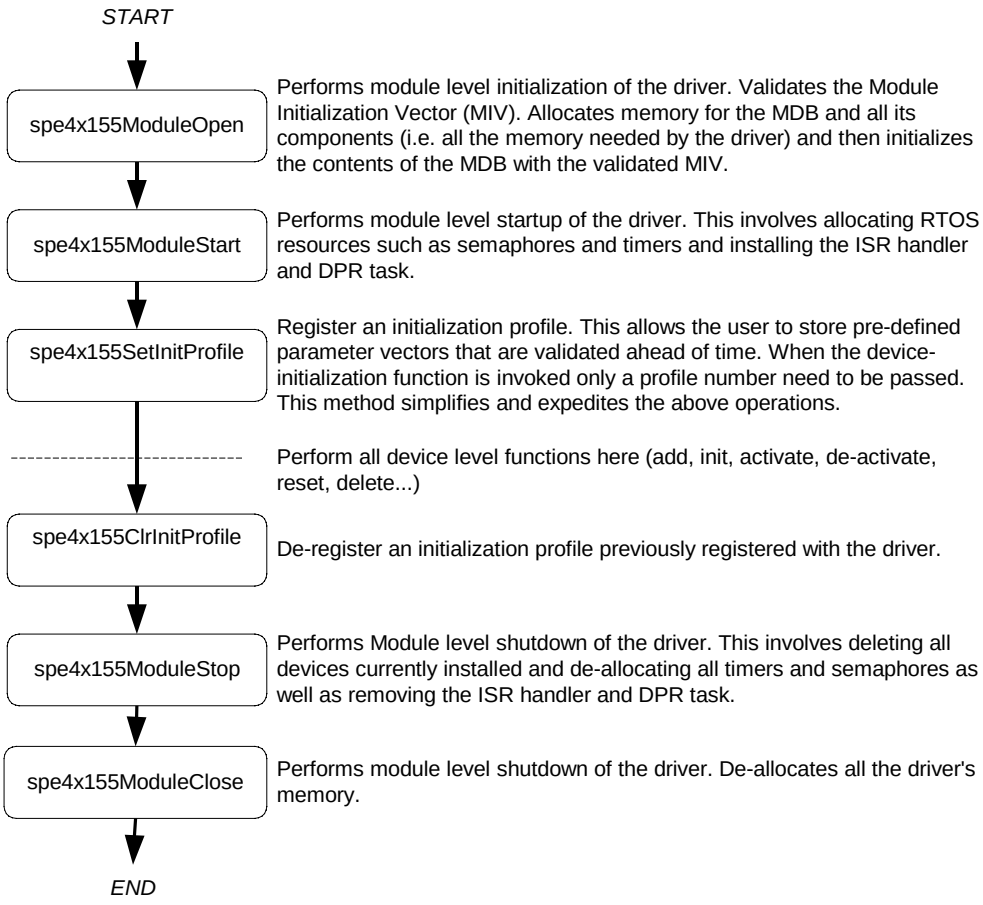
This section describes the main processing flows of the SPECTRA-4x155 driver components.

The flow diagrams below 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-4x155 driver module.

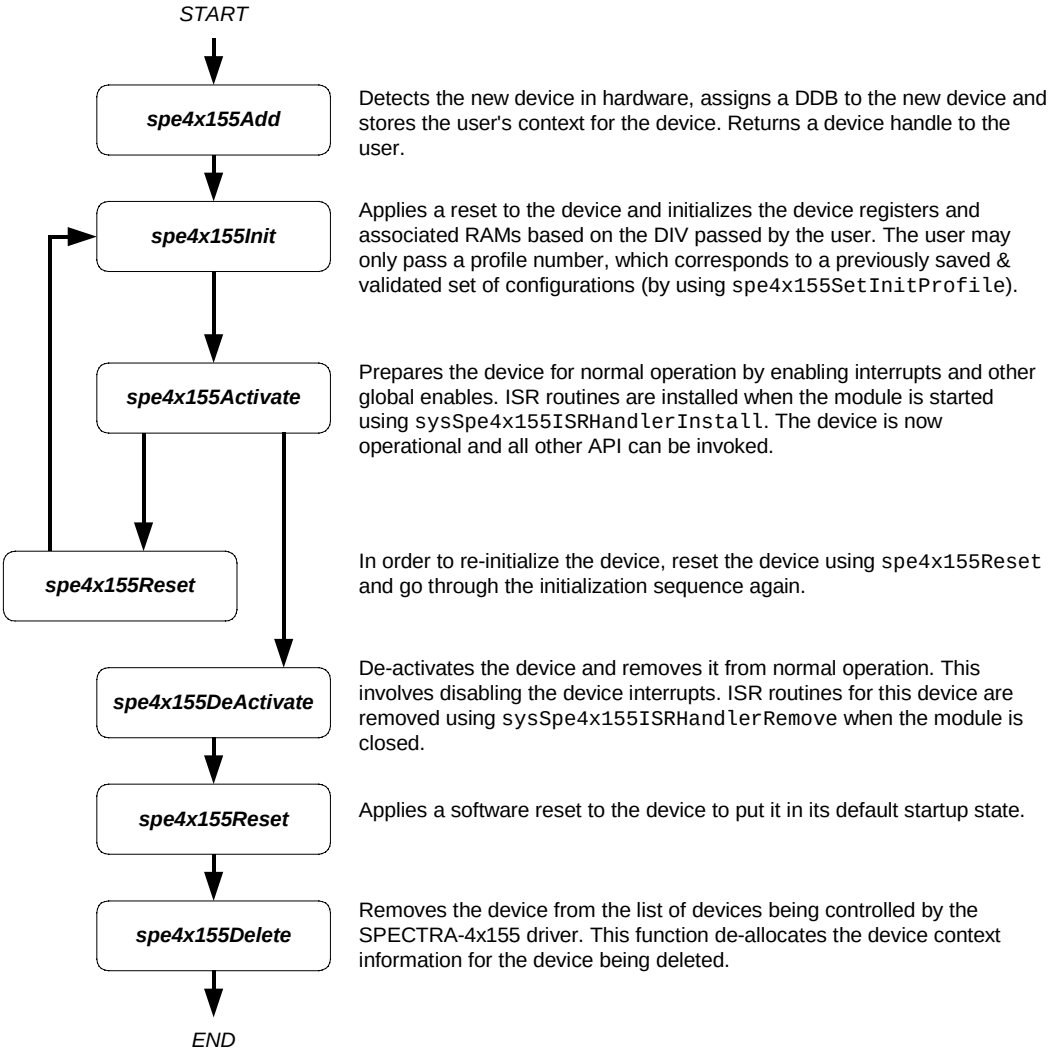
Figure 4: Module Management Flow Diagram



Device Management

The following figure shows the typical function call sequences that the driver uses to add, initialize, re-initialize, and delete the SPECTRA-4x155 device.

Figure 5: Device Management Flow Diagram



3.5 Interrupt Servicing

The SPECTRA-4x155 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-4x155 driver.

The driver provides system-independent functions, `spe4x155ISR` and `spe4x155DPR`. You must fill in the corresponding system-specific functions, `sysSpe4x155ISRHandler` and `sysSpe4x155DPRTask`. The system-specific functions isolate the system-specific communication mechanism (between the ISR and DPR) from the system-independent functions, `spe4x155ISR` and `spe4x155DPR`.

Figure 6 illustrates the interrupt service model used in the SPECTRA-4x155 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-4x155 driver to process the device's event-indication registers (see Figure 7).

Calling spe4x155ISR

An interrupt handler function, which is system dependent, must call `spe4x155ISR`. 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 (`sysSpe4x155ISRHandler`) appears on page 114. You can customize this example implementation to suit your needs.

The interrupt handler that you implement (`sysSpe4x155ISRHandler`) is installed in the interrupt vector table of the system processor. It is called when one or more SPECTRA-4x155 devices interrupt the processor. The interrupt handler then calls `spe4x155ISR` for each device in the active state that has interrupt processing enabled.

The spe4x155ISR function reads from the master interrupt-status registers and the miscellaneous interrupt-status registers of the SPECTRA-4x155. If at least one valid interrupt condition is found then spe4x155ISR fills an Interrupt-Service Vector (ISV) with this status information as well as the current device handle. The spe4x155ISR function also clears and disables all the device's interrupts detected. The sysSpe4x155ISRHandler function is then responsible to send this ISV buffer to the DPR task.

Note: Normally you should save the status information for deferred-processing by implementing a message queue. The interrupt handler sends the status information to the queue inside sysSpe4x155ISRHandler.

Calling spe4x155DPR

The sysSpe4x155DPRTask 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-4x155 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, sysSpe4x155DPRTask calls the DPR (spe4x155DPR) with the received ISV.

Then spe4x155DPR 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, spe4x155DPR 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 this callback function does not call any API functions that would change the driver's state, such as spe4x155Delete. Also, ensure that the callback function is non-blocking because the DPR task executes while SPECTRA-4x155 interrupts are disabled. You can customize these callbacks to suit your system. See page 107 for example implementations of the callback functions.

Note: Since the spe4x155ISR and spe4x155DPR 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, sysSpe4x155ISRHandler and sysSpe4x155DPRTask. When the driver calls sysSpe4x155ISRHandlerInstall, the application installs sysSpe4x155ISRHandler in the interrupt vector table of the processor, and the sysSpe4x155DPRTask function is spawned as a task by the application. The sysSpe4x155ISRHandlerInstall function also creates the communication channel between sysSpe4x155ISRHandler and sysSpe4x155DPRTask. This communication channel is most commonly a message queue associated with the sysSpe4x155DPRTask.

Similarly, during removal of interrupts, the driver removes `sysSpe4x155ISRHandler` from the microprocessor's interrupt vector table and deletes the task associated with `sysSpe4x155DPRTask`.

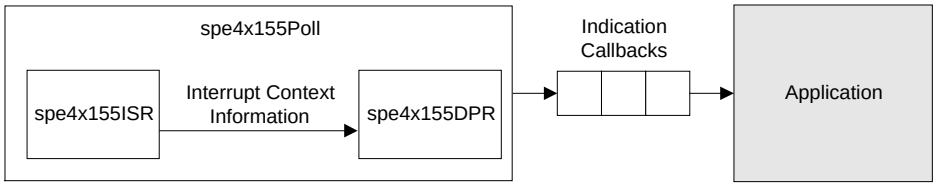
As a reference, this manual provides example implementations of the interrupt installation and removal functions on page 114. You can customize these prototypes to suit your specific needs.

Calling spe4x155Poll

Instead of using an interrupt service model, you can use a polling service model in the SPECTRA-4x155 driver to process the device's event-indication registers.

Figure 7 illustrates the polling service model used in the SPECTRA-4x155 driver design.

Figure 7: Polling Service Model



In polling mode, the application is responsible for calling `spe4x155Poll` often enough to service any pending error or alarm conditions. When `spe4x155Poll` is called, the `spe4x155ISR` function is called internally.

The `spe4x155ISR` function reads from the master interrupt-status registers and the miscellaneous interrupt-status registers of the SPECTRA-4x155. If at least one valid interrupt condition is found then `spe4x155ISR` fills an Interrupt-Service Vector (ISV) with this status information as well as the current device handle. The `spe4x155ISR` 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 `spe4x155DPR` internally.

4 DATA STRUCTURES

This section describes the elements of the driver that configure or control its behavior and therefore should be of interest to the application programmer. Included here are the constants, variables and structures that the SPECTRA-4x155 device driver uses to store initialization, configuration and statistics information. For more information on our naming convention, the reader is referred to Appendix A (page 129).

Note: All (STM-1, AU-3) arrays that are used to pass 4 STM-1 and 3 AU-3 information are shown as arrays of size [4][3] in this document. However, in the driver source code, these arrays are declared to be of size [4+1][3+1] with the zeroeth element left unused. For example, an sts3c[4] array shown in this document is declared in the source code as sts3c[4+1]. The zeroeth element, sts3c[0], is left unused.

4.1 Constants

The following Constants are used throughout the driver code:

- `<SPECTRA-4x155 ERROR CODES>`: error codes used throughout the driver code, returned by the API functions and used in the global error number field of the Module Data Block (MDB) and Device Data Block (DDB). See Appendix B (page 134) for a summary of the driver error codes.
- `SPE4X155_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.
- `SPE4X155_MOD_START, SPE4X155_MOD_IDLE, SPE4X155_MOD_READY`: are the three possible module states (stored in `stateModule`).
- `SPE4X155_START, SPE4X155_PRESENT, SPE4X155_ACTIVE, SPE4X155_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 Interrupt-Service Routine (ISR) mask.

Module Initialization Vector: MIV

Passed via the `spe4x155ModuleOpen` 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 SPE4X155_MAX_DEVS (see section 4.1).
- maxInitProfs is used to inform the driver how many Initialization profiles can be saved during this session. The number is used to calculate the amount of memory that will be allocated by the driver. The maximum value that can be passed is SPE4X155_MAX_INIT_PROFS (see section 4.1).

Table 1: SPECTRA-4x155 Module Initialization Vector: sSPE4X155_MIV

Field Name	Field Type	Field Description
perrModule	INT4 *	(pointer to) errModule (see description in the MDB)
maxDevs	UINT2	Maximum number of devices supported during this session
maxInitProfs	UINT2	Maximum number of initialization profiles

Device Initialization Vector: DIV

Passed via the spe4x155Init call, this structure contains all the information needed by the driver to initialize a SPECTRA-4x155 device. This structure is also passed via the spe4x155SetInitProfile 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' (SPE4X155_POLL_MODE), and 'interrupt driven' (SPE4X155_ISR_MODE). When configured in polling the interrupt capability of the device is NOT used, and the user is responsible for calling spe4x155Poll periodically. The actual processing of the event information is the same for both modes.
- cbackIO, cbackSOH, cbackLOH, cbackRPOH, cbackTPOH, cbackDPGM, and cbackAPGM are used to pass the address of application functions that will be used by the Deferred-Processing Routine (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 2: SPECTRA-4x155 Device Initialization Vector: sSPE4X155_DIV

Field Name	Field Type	Field Description
valid	UINT1	Indicates that this structure is valid

Field Name	Field Type	Field Description
pollISR	eSPE4X155_POLL	Indicates the type of ISR / polling to do
cbackIO	void *	Address for the callback function for Input / Output (IO) Events
cbackSOH	void *	Address for the callback function for Section Overhead (SOH) Events
cbackLOH	void *	Address for the callback function for Line Overhead (LOH) Events
cbackRPOH	void *	Address for the callback function for Receive Path Overhead (RPOH) Events
cbackTPOH	void *	Address for the callback function for Transmit Path Overhead (TPOH) Events
cbackDPGM	void *	Address for the callback function for DROP Bus PRBS Generator/Monitor (DPGM) Events
cbackAPGM	void *	Address for the callback function for ADD Bus PRBS Generator/Monitor (APGM) Events
sts3c[4]	UINT1	When set to 1, the given STS-3 (STM-1) payload is concatenated (STS-3c).
ringEna[4]	UINT1	Enables the ring control ports
cfgClock	sSPE4X155_CFG_CLK	Parameters for controlling the master clock configuration
cfgSF[4]	sSPE4X155_CFG_SFSD	Signal fail accumulation period, saturation threshold, declaring threshold, and clearing threshold parameters.
cfgSD[4]	sSPE4X155_CFG_SFSD	Signal degrade accumulation period, saturation threshold, declaring threshold, and clearing threshold parameters
cfgCnt	sSPE4X155_CFG_CNT	Counter configuration

Clock Configuration (CFG_CLK)

The user populates this structure to configure the device clocks.

Table 3: Clock Config: sSPE4X155_CFG_CLK

Field Name	Field Type	Field Description
pgmrchsel	UINT1	Selects the timing source of the PGMRCLK output
rclken	UINT1	Controls gating of the RCLK output
tclken	UINT1	Controls gating of the TCLK output
pgmrclken	UINT1	Controls gating of the PGMRCLK output
pgmrclksel	UINT1	Selects the clock frequency of the PGMRCLK output
pgmrtclken	UINT1	Controls gating of the PGMTCLK output
pgmrtclksel	UINT1	Selects the clock frequency of the PGMTCLK output
dropBusCfg	UINT1	DROP Bus clocking, 1=4x19.44 MHz or 0=77.76 MHz
addBusCfg	UINT1	ADD Bus clocking, 1=4x19.44 MHz or 0=77.76 MHz

Signal Fail / Signal Degrade Configuration (CFG_SFSD)

This structure contains all the fields needed to configure the signal fail (SF) and signal degrade (SD) accumulation period and thresholds. This structure contains all the fields needed to configure the device counters. It is part of the DIV and initialization profile.

Table 4: Counters Config: sSPE4X155_CFG_SFSD

Field Name	Field Type	Field Description
enable	UINT1	1=Enable SF/SD
accumPeriod	UINT4	The number of 8 KHz frames used to accumulate the B2 error subtotal
satuThresh	UINT2	Allowable number of B2 errors that can be accumulated during an evaluation window before an SF threshold event is declared

Field Name	Field Type	Field Description
declaThresh	UINT2	The threshold for the declaration of the SF alarm
clearThresh	UINT2	The threshold of maximum B2 allowed errors for clearing the SF alarm
clearMode	UINT1	1=clearing window size == 8x declaring window size 0=clearing==declaring
satuMode	UINT1	1=limit number accumulated B2 errors to saturation threshold

Config Counters (CFG_CNT)

The user passes this structure when invoking `spe4x155CfgStats`. It allows the user to configure the behavior of the device counts.

Table 5: SPECTRA-4x155 Statistic Counters: sSPE4X155_CFG_CNT

Field Name	Field Type	Field Description
sohBipBlk[4]	UINT1	Controls the indication of Section BIP saturated errors. When non-zero, Section BIP errors are counted by block
lohReiBlk[4]	UINT1	Controls the extraction of Line REI errors from the M1 byte. When non-zero, Line REI are counted by block
lohBipBlk[4]	UINT1	Controls the indication of Line BIP errors. When non-zero, Line BIP errors are counted by block
rpohBipBlk[4][3]	UINT1	Controls the indication of receive path BIP errors. When non-zero, Path BIP errors are counted by block
tpohBipBlk[4][3]	UINT1	Controls the indication of transmit path BIP errors. When non-zero, Path BIP errors are counted by block
rpohReiBlk[4][3]	UINT1	Controls the indication of Path REI errors. When non-zero, Path REI errors are counted by block

Field Name	Field Type	Field Description
rpohMonrs[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

ISR Enable/Disable Mask

Passed via the spe4x155SetMask, spe4x155GetMask and spe4x155ClearMask calls, this structure contains all the information needed by the driver to enable and disable any of the interrupts in the SPECTRA-4x155. The field value is non-zero if the interrupt is enabled, zero if disabled.

Table 6: SPECTRA-4x155 ISR Mask: sSPE4X155_MASK

Field Name	Field Type	Field Description
minte	UINT1	Master Interrupt Enable
trool	UINT1	Transmit reference clock out of lock
rrool[4]	UINT1	Receive reference clock out of lock
rdool[4]	UINT1	Receive data out of lock
aparrerr[4]	UINT1	ADD bus parity error
rdoolAux[4]	UINT1	Receive data out of lock state change
sbipe[4]	UINT1	Section BIP error
los[4]	UINT1	Loss of Signal
lof[4]	UINT1	Loss of Frame
oof[4]	UINT1	Out of frame
stiu[4]	UINT1	Section trace identifier unstable
stim[4]	UINT1	Section trace identifier mismatch
oofAux[4]	UINT1	Out of frame state change

Field Name	Field Type	Field Description
losAux[4]	UINT1	Loss of signal state change
lofAux[4]	UINT1	Loss of frame state change
lbipe[4]	UINT1	Line BIP error
lrei[4]	UINT1	Line remote error indication error
sfber[4]	UINT1	Signal failure
sdber[4]	UINT1	Signal degrade
cossm[4]	UINT1	Change of SSM message (Z1/S1)
coaps[4]	UINT1	Change of APS bytes (K1, K2)
apsbf[4]	UINT1	APS byte failure
lais[4]	UINT1	Line alarm indication signal
lrdi[4]	UINT1	Line remote defect indication
lrdiAux[4]	UINT1	Line RDI state change
laisAux[4]	UINT1	Line AIS state change
sfAux[4]	UINT1	Signal fail
sdAux[4]	UINT1	Signal degrade
prdiAux[4][3]	UINT1	Path RDI state change
rpaisAux[4][3]	UINT1	Path AIS state change
psluAux[4][3]	UINT1	Path signal label unstable state change
pslmAux[4][3]	UINT1	Path signal label mismatch state change
rplopAux[4][3]	UINT1	Loss of pointer state change
rplomAux[4][3]	UINT1	Loss of multiframe state change
tiuAux[4][3]	UINT1	Trace identifier unstable mode 1 state change
timAux[4][3]	UINT1	Trace identifier mismatch state change

Field Name	Field Type	Field Description
rplopcAux[4][3]	UINT1	Loss of pointer concatenation state change
rpaiscAux[4][3]	UINT1	Path AIS concatenation state change
tiu2Aux[4][3]	UINT1	Trace identifier unstable mode 2 state change
perdiAux[4][3]	UINT1	Path enhanced remote defect indication state change
rnewptr[4][3]	UINT1	Reception of new pointer indication
rcopsl[4][3]	UINT1	Path signal label change
rplopc[4][3]	UINT1	Path loss of pointer concatenated
rplop[4][3]	UINT1	Path loss of pointer
rpaisc[4][3]	UINT1	Path AIS concatenated
rpais[4][3]	UINT1	Path AIS
rprdi[4][3]	UINT1	Path remote defect indication
rpbipe[4][3]	UINT1	Path BIP error
rpreie[4][3]	UINT1	Path remote error indication
rperdi[4][3]	UINT1	Change of path enhanced remote defect indication
rilljreq[4][3]	UINT1	Illegal pointer justification request
rconcat[4][3]	UINT1	Concatenation indicator error in STS-1/STM-0/AU-3.
rdiscopa[4][3]	UINT1	Change of pointer alignment event
rinvndf[4][3]	UINT1	Invalid NDF code
rillptr[4][3]	UINT1	Illegal Pointer
rpse[4][3]	UINT1	Positive pointer justification indication received
rnse[4][3]	UINT1	Negative pointer justification indication received
rndf[4][3]	UINT1	Detection of NDF_enable indication
rlom[4][3]	UINT1	Loss of multiframe

Field Name	Field Type	Field Description
rcoma[4][3]	UINT1	Change of multiframe alignment
rese[4][3]	UINT1	Elastic store error
rpjee[4][3]	UINT1	DROP bus pointer justification event enable. Used with set/clear mask to enable/disable this interrupt
rppji[4][3]	UINT1	DROP bus positive pointer justification inserted. Event reported when rpjee is enabled
rnpji[4][3]	UINT1	DROP bus negative pointer justification inserted. Event reported when rpjee is enabled
rpuneq[4][3]	UINT1	Path connection unequipped
rptiu[4][3]	UINT1	Path trace identifier unstable
rptim[4][3]	UINT1	Path trace identifier mismatch
rpslm[4][3]	UINT1	Path signal label mismatch
rpslu[4][3]	UINT1	Path signal label unstable
dpgmGenSig[4][3]	UINT1	DROP generator signal
dpgmMonErr[4][3]	UINT1	DROP monitor error detected in received PRBS byte
dpgmMonSync[4][3]	UINT1	DROP monitor synchronization state change
dpgmMonSig[4][3]	UINT1	DROP monitor signal verification state by a slave monitor
tplopcAux[4][3]	UINT1	Change in transmit loss of pointer concatenation state
tpaiscAux[4][3]	UINT1	Change in transmit path AIS concatenation state
tpaisAux[4][3]	UINT1	Change in transmit path AIS state
tplopAux[4][3]	UINT1	Change in transmit loss of pointer state
tplomAux[4][3]	UINT1	Change in transmit loss of multiframe state
tese[4][3]	UINT1	Elastic store FIFO overflow/underflow error

Field Name	Field Type	Field Description
tpjee[4][3]	UINT1	Enable interrupts for pointer justification events inserted in the transmit stream. Used with set/clear mask to enable/disable this interrupt
tppji[4][3]	UINT1	Positive pointer justification event inserted into transmit stream reported when tpjee is enabled
tnpji[4][3]	UINT1	Negative pointer justification event inserted into transmit stream. Event reported when tpjee is enabled
tnewptr[4][3]	UINT1	New point indication received
tplopc[4][3]	UINT1	Change in transmit AU-3 loss of pointer concatenation status
tplop[4][3]	UINT1	Change in transmit loss of pointer status
tpaisc[4][3]	UINT1	Change in transmit path AIS concatenation status
tpais[4][3]	UINT1	Change in transmit path AIS status
tprdi[4][3]	UINT1	Change in transmit path remote defect indication
tpbipe[4][3]	UINT1	Change in transmit path BIP error status
tprei[4][3]	UINT1	Change in transmit path extended remote error indication status
tillreq[4][3]	UINT1	Illegal justification request event
tdiscopa[4][3]	UINT1	Discontinuous pointer change
tinvndf[4][3]	UINT1	Invalid NDF observed in receive stream
tconcat[4][3]	UINT1	Concatenation error
tillptr[4][3]	UINT1	Illegal Pointer
tnse[4][3]	UINT1	Negative pointer justification event detected in transmit stream
tpse[4][3]	UINT1	Positive pointer justification event detected in transmit stream
tndf[4][3]	UINT1	NDF enable pattern observed in receive stream

Field Name	Field Type	Field Description
tlom[4][3]	UINT1	Change in loss of multiframe status in received stream
tcoma[4][3]	UINT1	Change in multiframe alignment
apgmGenSig[4][3]	UINT1	ADD generator signal
apgmMonErr[4][3]	UINT1	ADD monitor error detected in received PRBS byte
apgmMonSync[4][3]	UINT1	ADD monitor synchronization state change
apgmMonSig[4][3]	UINT1	ADD monitor signal verification state by a slave monitor

Statistic Counters (CNT)

This structure, as well as its component structures, are being used by the statistics collection APIs to retrieve the device counts. The user can either collect all statistics at once by using spe4x155GetCnt, collect statistics from individual blocks using spe4x155GetCntSOH, spe4x155GetCntLOH, and/or spe4x155GetCntPOH.

Counters Top-level Structure

Table 7: SPECTRA-4x155 Statistic Counters: sSPE4X155_STAT_CNT

Field Name	Field Type	Field Description
cntSOH[4]	sSPE4X155_STAT_CNT_SOH	Statistics counters of the Section Overhead (SOH)
cntLOH[4]	sSPE4X155_STAT_CNT_LOH	Statistics counters of the Line Overhead (LOH)
cntRPOH[4][3]	sSPE4X155_STAT_CNT_RPOH	Statistics counters of the Receive Path Overhead (RPOH)
cntTPOH[4][3]	sSPE4X155_STAT_CNT_TPOH	Statistics counters of the Transmit Path Overhead (TPOH)
cntPJ[4][3]	sSPE4X155_STAT_CNT_PJ	Statistics counters for pointer justifications (PJ)

Section Overhead (SOH) Statistics Counters

Table 8: SPECTRA-4x155 Section Overhead Statistics Counters: sSPE4X155_STAT_CNT_SOH

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

Line Overhead (LOH) Statistics Counters

Table 9: SPECTRA-4x155 Line Overhead Status: sSPE4X155_STAT_CNT_LOH

Field Name	Field Type	Field Description
lohBip	UINT4	Line BIP errors counter
lohRei	UINT4	Line REI error counter

Receive Path Overhead (RPOH) Statistics Counters

Table 10: SPECTRA-4x155 Path Processing Statistics Counters: sSPE4X155_STAT_CNT_RPOH

Field Name	Field Type	Field Description
rpohBip	UINT4	Path BIP error counter
rpohRei	UINT4	Path REI error counter
pohDPGMPrse	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 11: SPECTRA-4x155 Transmit Path Processing Statistics Counters: sSPE4X155_STAT_CNT_TPOH

Field Name	Field Type	Field Description
tpohBip	UINT4	Path BIP error counter
pohAPGMPrse	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 (PJ) Statistics Counters

Table 12: SPECTRA-4x155 Pointer Justification Statistics Counters: sSPE4X155_STAT_CNT_PJ

Field Name	Field Type	Field Description
rPosJust	UINT4	Positive pointer justification event counter – receive side
rNegJust	UINT4	Negative pointer justification event counter – receive side
tPosJust	UINT4	Positive pointer justification event counter – transmit side
tNegJust	UINT4	Negative pointer justification event counter – transmit side

Time-Slot structure: TSLOT

Passed inside the Time-Slot Interchange (TSI) API function calls, this structure fully describes a single STM-1, AU-3 timeslot as configured in the TSI.

Table 13: SPECTRA-4x155 Time-Slot: sSPE4X155_TSLOT

Field Name	Field Type	Field Description
stm1	UINT1	STM-1 index
au3	UINT1	AU3 index

PRBS Configuration Structure

Passed inside the PRBS (DPGM and APGM) API function calls, this structure fully describes PRBS generator and monitor configuration.

Table 14: SPECTRA-4x155 PRBS Configuration: sSPE4X155_CFG_PRBS

Field Name	Field Type	Field Description
enable	UINT1	Enable PRBS generator/monitor 0=disable 1=enable
fsEnable	UINT1	Fixed stuff enable 0=insert in fixed stuff bytes 1=do not
invPrbs	UINT1	Invert PRBS value 0=do not invert 1=invert
autoMode	UINT1	Generator, autonomous input mode Monitor, allows automatic resynchronization 0=disable, 1=enable

Device and Alarm Status

Table 15: SPECTRA-4x155 Alarm Status: sSPE4X155_STATUS

Field Name	Field Type	Field Description
statIO	sSPE4X155_STATUS_IO	Alarm status of the Input / Output (IO)
statSOH[4]	sSPE4X155_STATUS_SOH	Alarm status of the Section Overhead (SOH)
statLOH[4]	sSPE4X155_STATUS_LOH	Alarm status of the Line Overhead (LOH)
statRPOH[4][3]	sSPE4X155_STATUS_RPOH	Alarm status of the Receive Path Overhead (RPOH)
statTPOH[4][3]	sSPE4X155_STATUS_TPOH	Alarm status for the Transmit Path Overhead (TPOH)

Input / Output (IO) Alarm Status

Table 16: SPECTRA-4x155 Input / Output Alarm Status: sSPE4X155_STATUS_IO

Field Name	Field Type	Field Description
refclkActive	UINT1	Monitors for low to high transitions on the REFCLK reference clock input
trool	UINT1	Transmit reference clock out of lock
dckAct	UINT1	Monitors for low to high transitions on the DCK input
ackAct	UINT1	Monitors for low to high transitions on the ACK input
insLRDI[4]	UINT1	Value of SENDLRDI bit position in the transmit ring control port
insLAIS[4]	UINT1	Value of SENDLAIS bit position in the transmit ring control port
rrool[4]	UINT1	Receive reference clock out of lock
rdool[4]	UINT1	Receive data clock out of lock
addControlAct[4]	UINT1	Monitors for low to high transitions in the corresponding APL[m], AC1J1V1[m] and ADP[m] inputs. addControlAct[n] is non-zero when rising edges have been observed on all these signals
addDataAct[4]	UINT1	Monitors for low to high transitions in the corresponding AD[7:0], AD[15:8}, AD[23:16], or AD[31:24] bus when configured for byte Telecom ADD bus mode. addDataAct[n] is non-zero when rising edges have been observed on all the required signals in the corresponding Telecom ADD bus

Section Overhead (SOH) Alarm Status

Table 17: SPECTRA-4x155 Section Overhead Alarm Status: sSPE4X155_STATUS_SOH

Field Name	Field Type	Field Description
los	UINT1	LOS

Field Name	Field Type	Field Description
lof	UINT1	LOF
oof	UINT1	OOF
tiu	UINT1	Section trace identifier unstable
tim	UINT1	Section trace identifier mismatch

Line Overhead (LOH) Alarm Status

Table 18: SPECTRA-4x155 Line Overhead Status: sSPE4X155_STATUS_LOH

Field Name	Field Type	Field Description
sfber	UINT1	SF
sdber	UINT1	SD
psbf	UINT1	APS byte failure
lrdi	UINT1	Line RDI
lais	UINT1	Line AIS

Receive Path Overhead (RPOH) Alarm Status

Table 19: SPECTRA-4x155 Receive Path Processing Alarm Status: sSPE4X155_STATUS_RPOH

Field Name	Field Type	Field Description
ptiu	UINT1	Path trace identifier unstable
ptim	UINT1	Path trace identifier mismatch
au3paisc	UINT1	STS-1 Path AIS concatenated
au3plopc	UINT1	STS-1 Path LOP concatenated
pais	UINT1	Path AIS

Field Name	Field Type	Field Description
lop	UINT1	Path LOP
prdi	UINT1	Path RDI
perdi	UINT1	Path Enhanced RDI
lom	UINT1	Loss of multiframe declared
uneq	UINT1	Path unequipped
pslm	UINT1	Path signal label mismatch
pslu	UINT1	Path signal label unstable
dropGenSig	UINT1	Generator signature status
dropMonSig	UINT1	Monitor signature status
dropMonSync	UINT1	DROP monitor change of synchronization state

Transmit Path Overhead (TPOH) Alarm Status

Table 20: SPECTRA-4x155 Transmit Path Processing Alarm Status: sSPE4X155_STATUS_TPOH

Field Name	Field Type	Field Description
au3lopc	UINT1	AU-3 concatenation path loss of pointer in transmit STS-1 stream
au3paisc	UINT1	AU-3 concatenation path AIS in transmit STS-1 stream
lop	UINT1	Loss of pointer detected
pais	UINT1	Path alarm indication signal detected
lom	UINT1	Loss of multiframe declared
addGenSig	UINT1	Generator signature status
addMonSig	UINT1	Monitor signature status
addMonSync	UINT1	ADD monitor change of synchronization state

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. These structures are the Module Data Block (MDB) and the Device Data Block (DDB).

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.

- `errModule` indicates that most of the module API functions return a specific error code directly. When the returned code is `SPE4X155_FAILURE`, this indicates that the top-level function was not able to carry the specified error code back to the application. Under those circumstances, the proper error code is recorded in this element. The element is the first in the structure so that the user can cast the MDB pointer into a INT4 pointer and retrieve the local error (this eliminates the need to include the MDB template into the application code).
- `valid` indicates that this structure has been properly initialized and may be read by the user.
- `stateModule` contains the current state of the module and could be set to: `SPE4X155_MOD_START, SPE4X155_MOD_IDLE` or `SPE4X155_MOD_READY`.

Table 21: SPECTRA-4x155 Module Data Block: sSPE4X155_MDB

Field Name	Field Type	Field Description
`errModule`	`INT4`	Global error Indicator for module calls
`valid`	`UINT1`	Indicates that this structure has been initialized
`stateModule`	`eSPE4X155_MOD_STATE`	Module state - can be one of the following IDLE or READY
`maxDevs`	`UINT2`	Maximum number of devices supported
`numDevs`	`UINT2`	Number of devices currently registered
`maxInitProfs`	`UINT2`	Maximum number of initialization profiles
`pddb`	`sSPE4X155_DDB *`	(array of) Device Data Blocks (DDB) in context memory
`pinitProfs`	`sSPE4X155_DIV *`	(array of) Initialization profiles in context memory

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.

- `errDevice` indicates that most of the device API functions return a specific error code directly. When the returned code is `SPE4X155_FAILURE`, this indicates that the top-level function was not able to carry the specific error code back to the application. In addition, some device functions do not return an error code. Under those circumstances, the proper error code is recorded in this element. The element is the first in the structure so that the user can cast the DDB pointer to a INT4 pointer and retrieve the local error (this eliminates the need to include the DDB template in the application code).
- `valid` indicates that this structure has been properly initialized and may be read by the user.
- `stateDevice` contains the current state of the device and could be set to: `SPE4X155_START, SPE4X155_PRESENT, SPE4X155_ACTIVE` or `SPE4X155_INACTIVE`.
- `usrCtxt` is a value that can be used by the user to identify the device during the execution of the callback functions. It is passed to the driver when `spe4x155Add` is called and returned to the user in the DPV when a callback function is invoked. The element is unused by the driver itself and may contain any value.

Table 22: SPECTRA-4x155 Device Data Block: sSPE4X155_DDB

Field Name	Field Type	Field Description
`errDevice`	`INT4`	Global error indicator for device calls
`valid`	`UINT1`	Indicates that this structure has been initialized
`stateDevice`	`eSPE4X155_DEV_STATE`	Device State - can be one of the following PRESENT, ACTIVE or INACTIVE
`baseAddr`	`UINT1 *`	Base address of the device
`usrCtxt`	`void *`	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
`pollISR`	`eSPE4X155_POLL`	Indicates the current type of ISR / polling
`cbackIO`	`void *`	Address for the callback function for Input / Output (IO) Events

Field Name	Field Type	Field Description
cbackSOH	void *	Address for the callback function for Section Overhead (SOH) Events
cbackLOH	void *	Address for the callback function for Line Overhead (LOH) Events
cbackRPOH	void *	Address for the callback function for Receive Path Overhead (RPOH) Events
cbackTPOH	void *	Address for the callback function for Transmit Path Overhead (TPOH) Events
cbackDPGM	void *	Address for the callback function for DROP Bus PRBS Generator/Monitor (DPGM) Events
cbackAPGM	void *	Address for the callback function for ADD Bus PRBS Generator/Monitor (APGM) Events
sts3c[4]	UINT1	When set to 1, the given STS-3 (STM-1) payload is concatenated (STS-3c).
ringEna[4]	UINT1	Enables the ring control ports
cfgClock	sSPE4X155_CFG_CLK	Parameters for controlling the master clock configuration
cfgDropGen[4][3]	sSPE4X155_CFG_PRBS	DROP Bus PRBS Generator configuration block
cfgDropMon[4][3]	sSPE4X155_CFG_PRBS	DROP Bus PRBS Monitor configuration block
cfgAddGen[4][3]	sSPE4X155_CFG_PRBS	ADD Bus PRBS Generator configuration block
cfgAddMon[4][3]	sSPE4X155_CFG_PRBS	ADD Bus PRBS Monitor configuration block
cfgSF[4]	sSPE4X155_CFG_SFSD	Signal fail accumulation period, saturation threshold, declaring threshold, and clearing threshold parameters

Field Name	Field Type	Field Description
cfgSD[4]	sSPE4X155_CFG_SFSD	Signal degrade accumulation period, saturation threshold, declaring threshold, and clearing threshold parameters
cfgCnt	sSPE4X155_CFG_CNT	Counter configuration
ver	UINT1	Device version number
threshold	sSPE4X155_MASK	application callback event thresholds
threshCntr	sSPE4X155_MASK	number of events that have occurred since the last time the thresholds have been reached
mask	sSPE4X155_MASK	Interrupt Enable Mask

4.4 Structures Passed through RTOS Buffers

Interrupt-Service Vector: ISV

This buffer structure is used to capture the status of the device (during a poll or ISR processing) for use by the Deferred-Processing Routine (DPR). It is the template for all device registers that are involved in exception processing. It is the application's responsibility to create a pool of ISV buffers (using this template to determine the buffer's size) when the driver calls the user-supplied sysSpe4x155BufferStart function. An individual ISV buffer is then obtained by the driver via sysSpe4x155ISVBufferGet and returned to the 'pool' via sysSpe4x155ISVBufferRtn.

Table 23: SPECTRA-4x155 Interrupt-Service Vector: sSPE4X155_ISV

Field Name	Field Type	Field Description
deviceHandle	sSPE4X155_HNDL	Handle to the device in cause
mask	sSPE4X155_MASK	ISR mask filled with interrupt status

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.

The DPR divides the SPECTRA-4x155 into 7 sections: IO, SOH, LOH, RPOH, TPOH, DPGM, and APGM. Seven user-supplied callback routines (one per section) are used to inform the application which section of the device has caused the event being reported.

Table 24: SPECTRA-4x155 Deferred-Processing Vector: sSPE4X155_DPV

Field Name	Field Type	Field Description
`event`	`SPE4X155_DPR_EVENT`	Event being reported
`cause`	`UINT4`	Reason for the Event

4.5 Global Variable

Although most of the variables within the driver are not meant to be used by the application code, there is one global variable that can be of great use to the application code.

`spe4x155Mdb`: A global pointer to the Module Data Block (MDB). The content of this global variable should 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 for functions that do not return an error code or when a value of `SPE4X155_FAILURE` is returned.
- `stateModule`: This structure element is used to store the module state (as shown in Figure 3).
- `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 array of DDBs is 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 for functions that do not return an error code or when a value of `SPE4X155_FAILURE` is returned.
    - `stateDevice`: This structure element is used to store the device state (as shown in Figure 3).

5 APPLICATION PROGRAMMING INTERFACE

This section provides a detailed description of each function that is a member of the SPECTRA-4x155 driver Application Programming Interface (API).

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

Note: These functions are not re-entrant. This means that two application tasks can not invoke the same API at the same time. However the driver protects its data structures from concurrent accesses by the application and the Deferred-Processing Routine (DPR) task.

5.1 Module Management

The module management is a set of API functions that are used by the application to open, start, stop and close the driver module. These functions take care of initializing the driver, allocating memory and all RTOS resources needed by the driver. They are also used to change the module state. For more information on the module states see the state diagram on page 23. For a typical module management flow diagram see page 25.

Opening the Driver Module: spe4x155ModuleOpen

This function performs module level initialization of the device driver. This involves allocating all of the memory needed by the driver and initializing the internal structures.

Prototype `INT4 spe4x155ModuleOpen(sSPE4X155_MIV *pmiv)`

Inputs `pmiv` : (pointer to) Module Initialization Vector (MIV)

Outputs Places the address of the MDB into the MIV passed by the application

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_MODULE_STATE`
`SPE4X155_ERR_INVALID_MIV`
`SPE4X155_ERR_MEM_ALLOC`

Valid States `SPE4X155_MOD_START`

Side Effects Changes the MODULE state to `SPE4X155_MOD_IDLE`

Closing the Driver Module: spe4x155ModuleClose

This function performs module level shutdown of the driver. This involves deleting all devices being controlled by the driver (by calling `spe4x155Delete` for each device) and de-allocating all the memory allocated by the driver.

Prototype `INT4 spe4x155ModuleClose(void)`

Inputs None

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_MODULE_STATE`

Valid States `SPE4X155_MOD_IDLE, SPE4X155_MOD_READY`

Side Effects Changes the MODULE state to `SPE4X155_MOD_START`

Starting the Driver Module: spe4x155ModuleStart

This function connects the RTOS resources to the driver. This involves initializing buffers and installing the Interrupt-Service Routine (ISR) handler and DPR task. Upon successful return from this function the driver is ready to add devices.

Prototype `INT4 spe4x155ModuleStart(void)`

Inputs None

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_MODULE_STATE`

Valid States `SPE4X155_MOD_IDLE`

Side Effects Changes the MODULE state to `SPE4X155_MOD_READY`

Stopping the Driver Module: spe4x155ModuleStop

This function disconnects the RTOS resources from the driver. This involves freeing-up buffers and uninstalling the ISR handler and the DPR task. If there are any registered devices, `spe4x155Delete` is called for each.

Prototype `INT4 spe4x155ModuleStop(void)`

Inputs None

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_MODULE_STATE`

Valid States `SPE4X155_MOD_READY`

Side Effects Changes the MODULE state to `SPE4X155_MOD_IDLE`

5.2 Profile Management

This section describes the functions that add, get and clear an initialization profile. Initialization profiles allow the user to store pre-defined Device Initialization Vectors (DIV) that are validated ahead of time. When the device initialization function is invoked only a profile number need to be passed. This method simplifies and expedites the initialization process.

Setting an Initialization Profile: spe4x155SetInitProfile

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

Prototype
```
INT4 spe4x155SetInitProfile(sSPE4X155_DIV
*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 = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_MODULE_STATE`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_PROFILE`
`SPE4X155_ERR_PROFILES_FULL`

Valid States `SPE4X155_MOD_IDLE, SPE4X155_MOD_READY`

Side Effects None

Getting an Initialization Profile: spe4x155GetInitProfile

This function gets the content of an initialization profile given its profile number.

Prototype `INT4 spe4x155GetInitProfile(UINT2 profileNum, sSPE4X155_DIV *pProfile)`

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

Outputs
`pProfile` : contents of the corresponding profile

Returns
Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_MODULE_STATE`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_PROFILE_NUM`

Valid States `SPE4X155_MOD_IDLE, SPE4X155_MOD_READY`

Side Effects None

Deleting an Initialization Profile: spe4x155ClrInitProfile

This function deletes an initialization profile given its profile number.

Prototype `INT4 spe4x155ClrInitProfile(UINT2 profileNum)`

Inputs
`profileNum` : initialization profile number

Outputs None

Returns
Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_MODULE_STATE`
`SPE4X155_ERR_INVALID_PROFILE_NUM`

Valid States `SPE4X155_MOD_IDLE, SPE4X155_MOD_READY`

Side Effects None

5.3 Device Management

The device management is a set of API functions that are used by the application to control the device. These functions take care of initializing a device in a specific configuration, enabling the device general activity as well as enabling interrupt processing for that device. They are also used to change the software state for that device. For more information on the device states see the state diagram on page 23. For a typical device management flow diagram see page 26.

Adding a Device: spe4x155Add

This function 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's used by the driver to identify the device on which the operation is to be performed.

Prototype `sSPE4X155_HNDL spe4x155Add(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 ERROR code written to the MDB on failure

`SPE4X155_ERR_INVALID_MODULE_STATE`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_DEVS_FULL`
`SPE4X155_ERR_DEV_ALREADY_ADDED`
`SPE4X155_ERR_INVALID_DEV`

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

Returns Success = device handle (to be used bas an argument to most of the SPECTRA-4x155 APIs)
Failure = NULL (pointer)

Valid States `SPE4X155_MOD_READY`

Side Effects Changes the DEVICE state to `SPE4X155_PRESENT`

Deleting a Device: spe4x155Delete

This function is used to remove the specified device from the list of devices being controlled by the SPECTRA-4x155 driver. Deleting a device involves invalidating the DDB for that device and releasing its associated device handle.

Prototype `INT4 spe4x155Delete(sSPE4X155_HNDL deviceHandle)`

Inputs `deviceHandle` : device handle (from `spe4x155Add`)

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`

Valid States `SPE4X155_PRESENT, SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects Changes the DEVICE state to `SPE4X155_START`

Initializing a Device: spe4x155Init

This function initializes the Device Data Block (DDB) associated with that device during `spe4x155Add`, applies a soft 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 spe4x155Init(sSPE4X155_HNDL deviceHandle,
sSPE4X155_DIV *pdiv, UINT2 profileNum)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`pdiv`	: (pointer to) Device Initialization Vector
`profileNum`	: profile number (ignored if `pdiv` is NULL)

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_DEVICE_STATE`
`SPE4X155_ERR_INVALID_DIV`
`SPE4X155_ERR_INVALID_PROFILE_NUM`

Valid States `SPE4X155_PRESENT`

Side Effects Changes the DEVICE state to `SPE4X155_INACTIVE`

Updating the Configuration of a Device: spe4x155Update

This function 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 `spe4x155Update` and `spe4x155Init` is that no soft reset will be applied to the device.

Prototype `INT4 spe4x155Update(sSPE4X155_HNDL deviceHandle, sSPE4X155_DIV *pdiv, UINT2 profileNum)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`pdiv`	: (pointer to) Device Initialization Vector (DIV)
`profileNum`	: profile number (ignored if `pdiv` is NULL)

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_DEVICE_STATE`
`SPE4X155_ERR_INVALID_DIV`
`SPE4X155_ERR_INVALID_PROFILE_NUM`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Resetting a Device: spe4x155Reset

This function applies a software reset to the SPECTRA-4x155 device. It also resets all the DDB contents (except for the user context). This function is typically called before re-initializing the device (via `spe4x155Init`).

Prototype `INT4 spe4x155Reset(sSPE4X155_HNDL deviceHandle)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`

Valid States `SPE4X155_PRESENT, SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects Changes the DEVICE state to `SPE4X155_PRESENT`

Activating a Device: spe4x155Activate

This function restores the state of a device after a de-activate. Interrupts may be re-enabled.

Prototype `INT4 spe4x155Activate(sSPE4X155_HNDL deviceHandle`

Inputs `deviceHandle` : device handle (from `spe4x155Add`)

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_INACTIVE`

Side Effects Changes the DEVICE state to `SPE4X155_ACTIVE`

De-Activating a Device: spe4x155DeActivate

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

Prototype `INT4 spe4x155DeActivate(sSPE4X155_HNDL deviceHandle`

Inputs `deviceHandle` : device handle (from `spe4x155Add`)

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE`

Side Effects Changes the DEVICE state to `SPE4X155_INACTIVE`

5.4 Device Read and Write

Reading from Device Registers: spe4x155Read

This function can be used to read a register of a specific SPECTRA-4x155 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, `sysSpe4x155Read`. Note that a failure to read returns a zero and any error indication is written to the associated DDB.

Prototype `UINT1 spe4x155Read(sSPE4X155_HNDL deviceHandle, UINT2 regNum)`

Inputs
`deviceHandle` : device handle (from `spe4x155Add`)
`regNum` : register number

Outputs
ERROR code written to the MDB
`SPE4X155_ERR_INVALID_DEV`
ERROR code written to the DDB
`SPE4X155_ERR_INVALID_REG`

Returns
Success = value read
Failure = 0

Valid States `SPE4X155_PRESENT, SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects May affect registers that change after a read operation

Writing to Device Registers: spe4x155Write

This function can be used to write to a register of a specific SPECTRA-4x155 device by providing the register number. This 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, `sysSpe4x155Write`. Note that a failure to write returns a zero and any error indication is written to the DDB.

Prototype `UINT1 spe4x155Write(sSPE4X155_HNDL deviceHandle, UINT2 regNum, UINT1 value)`

Inputs
`deviceHandle` : device handle (from `spe4x155Add`)
`regNum` : register number
`value` : value to be written

Outputs
ERROR code written to the MDB
`SPE4X155_ERR_INVALID_DEV`
ERROR code written to the DDB

SPE4X155_ERR_INVALID_REG

Returns Success = value written
Failure = 0

Valid States SPE4X155_PRESENT, SPE4X155_ACTIVE, SPE4X155_INACTIVE

Side Effects May change the configuration of the device

Reading from a block of Device Registers: spe4x155ReadBlock

This function can be used to read a register block of a specific SPECTRA-4x155 device by providing the starting register number, and the size to read. This 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, sysSpe4x155Read. Note that a failure to read returns a zero and any error indication is written to the DDB. It is the user's responsibility to allocate enough memory for the block read.

Prototype
```
void spe4x155ReadBlock(sSPE4X155_HNDL
deviceHandle, UINT2 startRegNum, UINT2 size,
UINT1 *pblock)
```

Inputs

deviceHandle	: device handle (from spe4x155Add)
startRegNum	: starting register number
size	: size of the block to read
pblock	: (pointer to) the block to read

Outputs ERROR code written to the MDB
SPE4X155_ERR_INVALID_DEV
ERROR code written to the DDB
SPE4X155_ERR_INVALID_ARG
SPE4X155_ERR_INVALID_REG
pblock : (pointer to) the block read

Returns Success = Last register value read
Failure = 0

Valid States SPE4X155_PRESENT, SPE4X155_ACTIVE, SPE4X155_INACTIVE

Side Effects May affect registers that change after a read operation

Writing to a Block of Device Registers: spe4x155WriteBlock

This function can be used to write to a register block of a specific SPECTRA-4x155 device by providing the starting register number and the block size. This 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, `sysSpe4x155Write`. 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 that any error indication is written to the DDB

Prototype
```
void spe4x155WriteBlock(sSPE4X155_HNDL
deviceHandle, UINT2 startRegNum, UINT2 size,
UINT1 *pblock, UINT1 *pmask)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`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 MDB
`SPE4X155_ERR_INVALID_DEV`
ERROR code written to the DDB
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_REG`

Returns
Success = Last register value written
Failure = 0

Valid States
`SPE4X155_PRESENT, SPE4X155_ACTIVE,`
`SPE4X155_INACTIVE`

Side Effects
May change the configuration of the device

5.5 Input Output

The I/O section provides functions to control the ring control ports on the device, giving the application an interface for applying line AIS and RDI, as well as configuration of the Time Slot Interchange (TSI).

Sending Line AIS Maintenance Signal: spe4x155RINGLineAISControl

This function forces a mate SPECTRA-4x155 to send the line AIS maintenance signal.

Prototype
```
INT4 spe4x155RINGLineAISControl(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 enable)
```

Inputs
`deviceHandle` : device handle (from `spe4x155Add`)
`stm1` : STM-1 index
`enable` : flag to start/stop Line-AIS insertion

Outputs None

Returns
Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Sending Line RDI Maintenance Signal: spe4x155RINGLineRDIControl

This function forces a mate SPECTRA-4x155 to send the line RDI maintenance signal.

Prototype
```
INT4 spe4x155RINGLineRDIControl(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 enable)
```

Inputs
`deviceHandle` : device handle (from `spe4x155Add`)
`stm1` : STM-1 index
`enable` : flag to start/stop Line-AIS insertion

Outputs None

Returns
Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Creating a Timeslot Mapping: spe4x155IOMapSlot

This function maps a source timeslot to one or more destination timeslots. This function supports a unicast or a multicast mapping.

Prototype
```
INT4 spe4x155IOMapSlot(sSPE4X155_HNDL
deviceHandle, UINT1 direction, sSPE4X155_TSLOT
*psrcSlot, sSPE4X155_TSLOT *pdestSlot, UINT1
numDestSlots)
```

Inputs
`deviceHandle` : device handle (from `spe4x155Add`)

	`direction`	: direction (symmetry) 0 = Drop 1 = Add
	`psrcSlot`	: (pointer to) source timeslot
	`pdestSlot`	: (pointer to) destination timeslot(s)
	`numDestSlots`	: number of destination timeslots

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEV`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Determining if a Timeslot is being Multicasted: spe4x155IOIsMulticast

This function informs the user of whether or not a source timeslot is being multicasted.

Prototype
```
INT4 spe4x155IOIsMulticast(sSPE4X155_HNDL
deviceHandle, UINT1 direction, sSPE4X155_TSLOT
*psrcSlot, UINT1 *pnumDestSlots)
```

Inputs	`deviceHandle`	: device handle (from `spe4x155Add`)
	`direction`	: direction (symmetry) 0 = Drop 1 = Add
	`psrcSlot`	: (pointer to) source timeslot
	`pnumDestSlots`	: (pointer to) allocated memory
Outputs	`pnumDestSlots`	: number of destination timeslot(s)

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEV`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Getting Destination Timeslot(s): spe4x155IOGetDestSlot

This function retrieves a list of destination timeslots mapped to a specified source timeslot.

Prototype
```
INT4 spe4x155IOGetDestSlot(sSPE4X155_HNDL
deviceHandle, UINT1 direction, sSPE4X155_TSLOT
```

```
*psrcSlot, sSPE4X155_TSLOT *pdestSlot, UINT1
*pnumDestSlots)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`direction`	: direction (symmetry) 0 = Drop 1 = Add
`psrcSlot`	: (pointer to) source timeslot
`pdestSlot`	: (pointer to) allocated memory
`pnumDestSlots`	: (pointer to) allocated memory

Outputs

`pdestSlot`	: destination timeslot
`pnumDestSlots`	: number of destination timeslot(s)

Returns

Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Getting Source Timeslot: spe4x155IOGetSrcSlot

This function retrieves the source timeslot mapped to a specified destination timeslot.

Prototype

```
INT4 spe4x155IOGetSrcSlot(sSPE4X155_HNDL
deviceHandle, UINT1 direction, sSPE4X155_TSLOT
*pdestSlot, sSPE4X155_TSLOT *psrcSlot)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`direction`	: direction (symmetry) 0 = Drop 1 = Add
`pdestSlot`	: (pointer to) destination timeslot
`psrcSlot`	: (pointer to) allocated memory

Outputs

`psrcSlot`	: source timeslot

Returns

Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

5.6 Section Overhead

The Section Overhead section provides functions to control and monitor the Section Overhead processing. Read / Write access is given to the section trace message (J0). This message is compared with a configurable reference and mismatches are reported. Section BIP-8 (B1) errors are accumulated in a counter that can be read and cleared. Section Overhead alarms are detected and reported. For diagnostic purposes, errors can be introduced in the Section Overhead bytes.

Terminating Section Overhead: spe4x155SOHTermination

This function enables or disables the section termination of a particular STM-1. There are two possible modes: no section termination (SPE4X155_NO_TERM) and section termination (SPE4X155_TERM). Enabling section termination allows insertion of ZO and a section trace message in the transmit stream.

Prototype `INT4 spe4x155SOHTermination (sSPE4X155_HNDL deviceHandle, UINT2 stm1, eSPE4X155_TERM mode)`

Inputs

`deviceHandle`	: device Handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`mode`	: section termination mode, one of the following: `SPE4X155_NO_TERM` or `SPE4X155_TERM`

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_DEVICE_STATE`
`SPE4X155_ERR_INVALID_ARG`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Reading and Setting the Section Trace Message (J0): spe4x155SectionTraceMsg

This function retrieves and sets the section trace message (J0) in the Sonet/SDH Section Trace Buffer. If this STM-1 is not configured for section termination, this function will have no effect on the section overhead.

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 spe4x155SectionTraceMsg(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 rw, UINT2 type,`

```
UINT2 length, UINT1* pJ0)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`rw`	: read/write flag, write if zero
`type`	: type of access 0 = tx section trace 1 = rx accepted section trace 2 = rx captured section trace 3 = rx expected section trace
`length`	: length of the message 0 = 16 bytes 1 = 64 bytes
`pJ0`	: (pointer to) the section trace message

Outputs `pJ0` : section trace message

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_POLL_TIMEOUT`
`SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Writing the Z0 Byte: spe4x155SOHWriteZ0

This function writes the Z0 byte into the transmit Section Overhead. If this STM-1 is not configured for section termination, this function will have no effect on the section overhead.

Prototype

```
INT4 spe4x155SOHWriteZ0(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT1 Z0)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`Z0`	: Z0 byte to write

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Forcing Out-of-Frame: spe4x155SOHDiagOOF

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

Prototype
```
INT4 spe4x155SOHDiagOOF(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 enable)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`enable`	: flag to start / stop forcing OOF

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Forcing Errors in the Framing Bytes: spe4x155SOHDiagFB

This function enables the insertion of a single bit error continuously in the most significant bit (bit1) of the A1 Section Overhead framing byte. A1 bytes are set to 76H instead of F6H.

Prototype
```
INT4 spe4x155SOHDiagFB(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 enable)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`enable`	: flag to start / stop error insertion

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Forcing Section BIP-8 Errors: spe4x155SOHDiagB1

This function enables insertion of bit errors continuously in the B1 Section Overhead byte.

Prototype `INT4 spe4x155SOHDiagB1(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 enable)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`enable`	: flag to start / stop error insertion

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Forcing Loss-of-Signal: spe4x155SOHDiagLOS

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

Prototype `INT4 spe4x155SOHDiagLOS(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 enable)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`enable`	: flag to start / stop error insertion

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

5.7 Line Overhead

The Line Overhead section is responsible for configuring and monitoring the Line Overhead on both receive and transmit sides. Read / Write access is given to the APS bytes (K1 and K2) and most of the other overhead bytes. Line BIP-8 (B2) errors and REI are accumulated in counters that can be read and cleared. Line Overhead alarms are detected and reported. For diagnostic purposes, errors can be introduced in the Line Overhead bytes. Additional functions are provided to configure the device to automatically insert line RDI and AIS.

Terminating Line Overhead: spe4x155LOHTermination

This function enables or disables the line termination of a particular STM-1. There are three possible modes: no line termination (SPE4X155_NO_TERM), line termination (SPE4X155_TERM) and line termination with automatic alarm insertion (SPE4X155_TERM_WITH_AUTO_INSERT). In termination mode, the K1 K2 values in device registers are inserted by the device, and in no termination mode the K1 K2 values are inserted by the overhead pins. In auto-insertion mode, line AIS is asserted on the DROP bus stream for signal degrade, signal failure, loss of frame, loss of signal, trace identifier mismatch, and trace identifier unstable alarms. Line RDI is asserted on the transmit stream for line AIS, signal degrade, signal failure, loss of frame, loss of signal, trace identifier mismatch, and trace identifier unstable alarms. As well, Line REI B2 error counts are inserted in the Z2/M1 byte on the transmit stream.

Prototype
```
INT4 spe4x155LOHTermination (sSPE4X155_HNDL
deviceHandle, UINT2 stm1, eSPE4X155_TERM mode)
```

Inputs

deviceHandle	: device Handle (from spe4x155Add)
stm1	: STM-1 index
mode	: line termination mode, one of the following: SPE4X155_NO_TERM, SPE4X155_TERM or SPE4X155_TERM_WITH_AUTO_INSERT

Outputs None

Returns Success = SPE4X155_SUCCESS
Failure = SPE4X155_ERR_INVALID_DEV
SPE4X155_ERR_INVALID_DEVICE_STATE
SPE4X155_ERR_INVALID_ARG

Valid States SPE4X155_ACTIVE, SPE4X155_INACTIVE

Side Effects None

Reading the Received K1 and K2 Bytes: spe4x155LOHReadK1K2

This function reads the K1 and K2 bytes from the received Line Overhead.

Prototype `INT4 spe4x155LOHReadK1K2(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT1 *pK1, UINT1 *pK2)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`pK1`	: (pointer to) K1 byte
`pK2`	: (pointer to) K2 byte

Outputs

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

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Writing the Transmitted K1 and K2 Bytes: spe4x155LOHWriteK1K2

This function writes the K1 and K2 bytes into the transmit line overhead. If this STM-1 is not configured for line termination, this function will have no effect on the line overhead.

Prototype `INT4 spe4x155LOHWriteK1K2(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT1 K1, UINT1 K2)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`K1`	: K1 byte to write
`K2`	: K2 byte to write

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Reading the S1 Byte: spe4x155LOHReadS1

This function read the S1 byte from the receive Line Overhead.

Prototype `INT4 spe4x155LOHReadS1(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT1 *pS1))`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`pS1`	: (pointer to) allocated memory

Outputs

`pS1`	: received S1 byte

Returns

Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Writing the S1 Byte: spe4x155LOHWriteS1

This function writes the S1 byte into the transmit Line Overhead.

Prototype `INT4 spe4x155LOHWriteS1(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT1 S1)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`S1`	: S1 byte to write

Outputs None

Returns

Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Inserting Line Remote Defect Indication: spe4x155LOHInsertLineRDI

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 spe4x155LOHInsertLineRDI(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 enable)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`enable`	: flag to start / stop Line RDI insertion

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Forcing Line BIP-8 Errors: spe4x155LOHDiagB2

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

Prototype `INT4 spe4x155LOHDiagB2(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 enable)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`enable`	: flag to start / stop B2 error insertion

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Inserting Line AIS: spe4x155LOHInsertLineAIS

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

Prototype `INT4 spe4x155LOHInsertLineAIS(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 enable)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`enable`	: flag to start / stop Line-AIS insertion

Outputs `None`

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Configuring Signal Fail and Signal Degrade: spe4x155LOHCfgSFSD

This function configures the Signal Fail and Signal Degrade bit error rate monitoring capabilities.

Prototype `INT4 spe4x155LOHCfgSFSD(sSPE4X155_HNDL deviceHandle, UINT2 stm1, sSPE4X155_CFG_SFSD *pcfgSF, sSPE4X155_CFG_SFSD *pcfgSD)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`pcfgSF`	: SF configuration
`pcfgSD`	: SD configuration

Outputs `None`

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

5.8 Path Overhead

The Path Overhead section configures and monitors the Path Overhead on both receive and transmit sides. Read / Write access is given to the path trace message (J1) and the path signal label (C2). Both are compared with a configurable reference and mismatches are reported. Path BIP-8 (B3) errors and REI are accumulated in a counter that can be read and cleared. Path Overhead alarms are detected and reported. For diagnostic purposes, errors can be introduced in the Path Overhead bytes. Additional functions are provided to configure the device to automatically insert path RDI, path enhanced RSI and path AIS.

Path Trace Message

Retrieving and Setting the Path Trace Messages: spe4x155PathTraceMsg

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. If this STM-1, AU-3 is not configured for TPOH termination, this function will have no effect on the path overhead.

Prototype

```
INT4 spe4x155PathTraceMsg(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 au3, UINT2 rw,
UINT2 type, UINT2 length, UINT1* pJ1)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`rw`	: read/write flag, write if zero
`type`	: type of access
`length`	: length of the message 0 = 16 bytes 1 = 64 bytes
`pJ1`	: (pointer to) the path trace message

Outputs

`pJ1`	: updated path trace message

Returns

Success = `SPE4X155_SUCCESS`

Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`
`SPE4X155_ERR_POLL_TIMEOUT`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Receive Path Overhead (RPOH)

Terminating Receive Path Overhead: spe4x155RPOHTermination

This function enables or disables the path termination of a particular (STM-1, AU3) slice. There are two possible modes: no path termination (SPE4X155_NO_TERM), and path termination with automatic alarm insertion (SPE4X155_TERM_WITH_AUTO_INSERT). In automatic alarm insertion mode, on detection of loss of pointer, path alarm indication signal, loss of signal, loss of frame, line alarm indication signal, unequipped, path signal label unstable, path signal label mismatch, path trace identifier unstable, and path trace identifier mismatch alarms, path AIS is inserted on the DROP bus, plus RDI and Enhanced RDI are inserted in the transmit stream. On detection of received B3 errors, the path REI count is inserted in the transmit G1 byte.

Prototype
```
INT4 spe4x155RPOHTermination(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 au3,
eSPE4X155_TERM mode)
```

Inputs

deviceHandle	: device Handle (from spe4x155Add)
stm1	: STM-1 index
au3	: AU-3 index
mode	: path termination mode, one of the following: SPE4X155_NO_TERM or SPE4X155_TERM_WITH_AUTO_INSERT

Outputs None

Returns Success = SPE4X155_SUCCESS
Failure = SPE4X155_ERR_INVALID_DEV
SPE4X155_ERR_INVALID_DEVICE_STATE
SPE4X155_ERR_INVALID_ARG

Valid States SPE4X155_ACTIVE, SPE4X155_INACTIVE

Side Effects None

Path Signal Label: spe4x155RPOHPathSignalLabel

This function retrieves and sets the expected, as well, as retrieves the captured path signal label (C2).

Prototype
```
spe4x155RPOHPathSignalLabel(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 au3, UINT2 rw,
UINT2 type, UINT1* pPSL)
```

Inputs

deviceHandle	: device Handle (from spe4x155Add)
stm1	: STM-1 index
au3	: AU-3 index
rw	: read/write flag, write if zero

	`type`	: type of access 1 = received PSL 2 = expected PSL
	`pPSL`	: (pointer to) path signal label
Outputs	`pPSL`	: updated path signal label

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Forcing Loss-of-Pointer: spe4x155RPOHDiagLOP

This function forces the downstream pointer processing to enter the Loss of Pointer (LOP) state. It does so by using the Diagnose LOP functionality activiated by setting the DLOP bit in the RTAL control register.

Prototype
```
INT4 spe4x155RPOHDiagLOP(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 au3, UINT2
enable)
```

Inputs	`deviceHandle`	: device handle (from `spe4x155Add`)
	`stm1`	: STM-1 index
	`au3`	: AU-3 index
	`enable`	: flag to start/stop NDF inversion
Outputs	None	

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Forcing Pointer Justification: spe4x155RPOHDiagPJ

This function diagnoses the downstream pointer processing elements so there are correct reactions to pointer justification events. The function forces positive or negative justification events, or disables the generation of any pointer justification element.

Prototype `INT4 spe4x155RPOHDiagPJ(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 au3, UINT2 enable, INT1 type)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`enable`	: flag to start/stop diagnostic
`type`	: type of pointer justification event: 0 = no pointer justification –1 = negative pointer justification +1 = positive pointer justification

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Forcing Errors in the H4 Byte: spe4x155RPOHDiagH4

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 spe4x155RPOHDiagH4(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 au3)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`enable`	: flag to start/stop H4 inversion

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Forcing Tributary Path AIS: spe4x155RPOHInsertTUAIS

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 spe4x155RPOHInsertTUAIS(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 au3, UINT2 enable)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`enable`	: flag to start/stop TUAIS

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Forcing Path AIS: spe4x155RPOHInsertPAIS

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

Prototype `INT4 spe4x155RPOHInsertPAIS(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 au3, UINT2 enable)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`enable`	: flag to start/stop PAIS insertion

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States SPE4X155_ACTIVE, SPE4X155_INACTIVE

Side Effects None

Transmit Path Overhead (TPOH)

Terminating Transmit Path Overhead: spe4x155TPOHTermination

This function enables or disables the path termination of a particular (STM-1, AU3) slice. There are three possible modes: no path termination (SPE4X155_NO_TERM), path termination (SPE4X155_TERM) and path termination with automatic alarm insertion (SPE4X155_TERM_WITH_AUTO_INSERT). If selected, automatic error insertion in G1 is performed. In path termination mode, device insertion of the path trace message, REI, C2, F2, Z3, Z4, and N1/Z5 is enabled. In alarm auto-insertion mode, path AIS is inserted in the transmit stream on detection of LOP and path AIS on the ADD bus, as well as for detection of concatenated LOP and PAIS detection on AU-3 #2 and #3 in a concatenated stream.

Prototype
```
INT4 spe4x155TPOHTermination(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 au3,
eSPE4X155_TERM mode)
```

Inputs

deviceHandle	: device Handle (from spe4x155Add)
stm1	: STM-1 index
au3	: AU-3 index
mode	: path termination mode, one of the following: SPE4X155_NO_TERM, SPE4X155_TERM or

SPE4X155_TERM_WITH_AUTO_INSERT

Outputs None

Returns
Success = SPE4X155_SUCCESS
Failure = SPE4X155_ERR_INVALID_DEV
SPE4X155_ERR_INVALID_DEVICE_STATE
SPE4X155_ERR_INVALID_ARG

Valid States SPE4X155_ACTIVE, SPE4X155_INACTIVE

Side Effects None

Forcing Path BIP-8 Errors: spe4x155TPOHDiagB3

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 spe4x155TPOHDiagB3(sSPE4X155_HNDL
```

```
deviceHandle, UINT2 stm1, UINT2 au3, UINT2
enable)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`enable`	: flag to start/stop B3 inversion

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Forcing a Pointer Value: spe4x155TPOHForceTxPtr

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 spe4x155TPOHForceTxPtr(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 au3, UINT2
enable, UINT2 aptr)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`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 = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Writing the New Data Flag Bits: spe4x155TPOHInsertNDF

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

Prototype `INT4 spe4x155TPOHInsertNDF(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 au3, UINT2 enable, UINT1 ndf)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`enable`	: flag to start/stop NDF insertion
`ndf`	: NDF value

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Writing the J1 Byte: spe4x155TPOHWriteJ1

This function writes the J1 byte into the Transmit Path Overhead.

Prototype `INT4 spe4x155TPOHWriteJ1(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 au3, UINT1 J1)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`J1`	: J1 byte to write

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Writing the C2 Byte: spe4x155TPOHWriteC2

This function writes the C2 byte into the Transmit Path Overhead. If this (STM-1, AU3) is not configured for path termination, this function will have no effect on the path overhead.

Prototype `INT4 spe4x155TPOHWriteC2(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 au3, UINT1 C2)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`C2`	: C2 byte to write

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Writing the Path Remote Error Indication Count: spe4x155TPOHInsertPREI

This function inserts the path remote error indication count passed in argument inside the path status byte. If this (STM-1, AU3) is not configured for path termination, this function will have no effect on the path overhead.

Prototype `INT4 spe4x155TPOHInsertPREI(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 au3, UINT1 PREI)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`prei`	: PREI value

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Writing the F2 Byte: spe4x155TPOHWriteF2

This function writes the F2 byte into the Transmit Path Overhead. If this (STM-1, AU3) is not configured for path termination, this function will have no effect on the path overhead.

Prototype
```
INT4 spe4x155TPOHWriteF2(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 au3, UINT1 F2)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`F2`	: F2 byte to write

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Writing the Z3 Byte: spe4x155TPOHWriteZ3

This function writes the Z3 byte into the Transmit Path Overhead. If this (STM-1, AU3) is not configured for path termination, this function will have no effect on the path overhead.

Prototype
```
INT4 spe4x155TPOHWriteZ3(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 au3, UINT1 Z3)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`Z3`	: Z3 byte to write

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Writing the Z4 Byte: spe4x155TPOHWriteZ4

This function writes the Z4 byte into the Transmit Path Overhead. If this (STM-1, AU3) is not configured for path termination, this function will have no effect on the path overhead.

Prototype
```
INT4 spe4x155TPOHWriteZ4(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 au3, UINT1 Z4)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`Z4`	: Z4 byte to write

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Writing the N1/Z5 Byte: spe4x155TPOHWriteZ5

This function writes the N1/Z5 byte into the Transmit Path Overhead. If this (STM-1, AU3) is not configured for path termination, this function will have no effect on the path overhead.

Prototype
```
INT4 spe4x155TPOHWriteZ5(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 au3, UINT1 Z5)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`Z5`	: Z5 byte to write

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Controlling Pointer Justification: spe4x155TPOHDiagPJ

This function diagnoses the downstream pointer processing elements for correct reaction to pointer justification events. It can force positive or negative stuff justification events, or disable the generation of any pointer justification element.

Prototype
```
INT4 spe4x155TPOHDiagPJ(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 au3, UINT2
enable, INT1 type)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`enable`	: flag to start/stop diagnostic
`type`	: type of pointer justification event: 0 for no pointer justification -1 for negative +1 for positive

Outputs None

Returns
Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Forcing Multiframe Error: spe4x155TPOHDiagH4

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 spe4x155TPOHDiagH4(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 au3, UINT2
enable)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`enable`	: flag to start/stop H4 inversion

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Forcing Tributary Path AIS: spe4x155TPOHInsertTUAIS

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 spe4x155TPOHInsertTUAIS(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 au3, UINT2
enable)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`enable`	: flag to start/stop TUAIS insertion

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Forcing Path AIS: spe4x155TPOHInsertPAIS

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 spe4x155TPOHInsertPAIS(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 au3, UINT2
enable)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)

	`stm1`	: STM-1 index
	`au3`	: AU-3 index
	`enable`	: flag to start/stop PAIS insertion

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

5.9 DROP / ADD Bus PRBS Generator and Monitor (DPGM / APGM)

DROP Bus PRBS Generator and Monitor (DPGM)

Configuring the PRBS Generator: spe4x155DPGMGenCfg

This function fully configures the Drop bus PRBS generator.

Prototype
```
INT4 spe4x155DPGMGenCfg(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 au3,
sSPE4X155_CFG_PRBS *pcfgPrbs)
```

Inputs	`deviceHandle`	: device handle (from `spe4x155Add`)
	`stm1`	: STM-1 index
	`au3`	: AU-3 index
	`pcfgPrbs`	: PRBS configuration structure

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Configuring the PRBS Monitor: spe4x155DPGMMonCfg

This function fully configures the Drop bus PRBS monitor.

Prototype `INT4 spe4x155DPGMMonCfg(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 au3, sSPE4X155_CFG_PRBS *pcfgPrbs)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`pcfgPrbs`	: PRBS configuration structure

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Forcing Generation of a New PRBS: spe4x155DPGMGenRegen

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 spe4x155DPGMGenRegen(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 au3)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Forcing Bit Errors: spe4x155DPGMGenForceErr

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 spe4x155DPGMGenForceErr(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 au3)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Forcing a Resynchronization: spe4x155DPGMMonResync

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 spe4x155DPGMMonResync(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 au3)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

ADD Bus PRBS Generator and Monitor (APGM)

Configuring the PRBS Generator: spe4x155APGMGenCfg

This function fully configures the Add bus PRBS generator.

Prototype `INT4 spe4x155APGMGenCfg(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 au3, sSPE4X155_CFG_PRBS *pcfgPrbs)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`pcfgPrbs`	: PRBS configuration structure

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Configuring the PRBS Monitor: spe4x155APGMMonCfg

This function fully configures the Add bus PRBS monitor.

Prototype `INT4 spe4x155APGMMonCfg(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 au3, sSPE4X155_CFG_PRBS *pcfgPrbs)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`pcfgPrbs`	: PRBS configuration structure

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Forcing Generation of a New PRBS: spe4x155APGMGenRegen

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 spe4x155APGMGenRegen(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 au3)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Forcing Bit Errors: spe4x155APGMGenForceErr

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 spe4x155APGMGenForceErr(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 au3)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Forcing a Resynchronization: spe4x155APGMMonResync

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 spe4x155APGMMonResync(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 au3)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

5.10 Interrupt Service Functions

This Section describes interrupt-service functions that perform the following tasks:

- Set, get and clear the interrupt enable mask
- Read and process the interrupt-status registers
- Poll and process the interrupt-status registers

See page 27 for an explanation of our interrupt servicing architecture.

Configuring ISR Processing: spe4x155ISRConfig

This function allows the user to configure how ISR processing is to be handled: polling (`SPE4X155_POLL_MODE`) or interrupt driven (`SPE4X155_ISR_MODE`). If polling is selected, the user is responsible for calling periodically `spe4x155Poll` to collect exception data form the device.

Prototype `INT4 spe4x155ISRConfig(sSPE4X155_HNDL deviceHandle, UINT2 mode)`

Inputs `deviceHandle` : device handle (from `spe4x155Add`)
`mode` : mode of operation

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_MODE`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Getting the Interrupt Status Mask: spe4x155GetMask

This function returns the contents of the interrupt mask registers of the SPECTRA-4x155 device.

Prototype
```
INT4 spe4x155GetMask(sSPE4X155_HNDL
deviceHandle, sSPE4X155_MASK *pmask)
```

Inputs `deviceHandle` : device handle (from `spe4x155Add`)
`pmask` : (pointer to) mask structure

Outputs `pmask` : (pointer to) updated mask structure

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Setting the Interrupt Enable Mask: spe4x155SetMask

This function sets the contents of the interrupt mask registers of the SPECTRA-4x155 device.

Prototype
```
INT4 spe4x155SetMask(sSPE4X155_HNDL
deviceHandle, sSPE4X155_MASK *pmask)
```

Inputs `deviceHandle` : device handle (from `spe4x155Add`)
`pmask` : (pointer to) mask structure

Outputs None

Returns Success = `SPE4X155_SUCCESS`

Failure = SPE4X155_ERR_INVALID_DEV
SPE4X155_ERR_INVALID_ARG
SPE4X155_ERR_INVALID_DEVICE_STATE

Valid States SPE4X155_ACTIVE, SPE4X155_INACTIVE

Side Effects May change the operation of the ISR / DPR

Clearing the Interrupt Enable Mask: spe4x155ClearMask

This function clears individual interrupt bits and registers in the SPECTRA-4x155 device. Any bits that are set in the passed structure are cleared in the associated SPECTRA-4x155 registers.

Prototype
```
INT4 spe4x155ClearMask(sSPE4X155_HNDL
deviceHandle, sSPE4X155_MASK *pmask)
```

Inputs
`deviceHandle` : device handle (from `spe4x155Add`)
`pmask` : (pointer to) mask structure

Outputs None

Returns Success = SPE4X155_SUCCESS
Failure = SPE4X155_ERR_INVALID_DEV
SPE4X155_ERR_INVALID_ARG
SPE4X155_ERR_INVALID_DEVICE_STATE

Valid States SPE4X155_ACTIVE, SPE4X155_INACTIVE

Side Effects May change the operation of the ISR / DPR

Polling the Interrupt Status Registers: spe4x155Poll

This function commands the driver to poll the interrupt registers in the device. The call will fail unless the device was initialized (via `spe4x155Init`) or configured (via `spe4x155ISRConfig`) into polling mode.

Prototype
```
INT4 spe4x155Poll(sSPE4X155_HNDL deviceHandle)
```

Inputs
`deviceHandle` : device handle (from spe4x155Add)

Outputs None

Returns Success = SPE4X155_SUCCESS
Failure = SPE4X155_ERR_INVALID_DEV
SPE4X155_ERR_INVALID_MODE
SPE4X155_ERR_INVALID_DEVICE_STATE

Valid States SPE4X155_ACTIVE

Side Effects None

Interrupt-Service Routine: spe4x155ISR

This function reads the state of the interrupt registers in the SPECTRA-4x155 and stores them in an ISV. Performs whatever functions are needed to clear the interrupt, from simply clearing bits to complex functions. This routine is called by the application code, from within `sysSpe4x155ISRHandler`. If ISR mode is configured all interrupts that were detected are disabled and the ISV is returned to the application. Note that the application is then responsible for sending this buffer to the DPR task. If polling mode is selected, no ISV is returned to the application and the DPR is called directly with the ISV.

Prototype `void * spe4x155ISR(sSPE4X155_HNDL deviceHandle)`

Inputs `deviceHandle` : device handle (from `spe4x155Add`)

Outputs None

Returns (pointer to) ISV buffer (to send to the DPR) or NULL (pointer)

Valid States `SPE4X155_ACTIVE`

Side Effects None

Deferred-Processing Routine: spe4x155DPR

This function acts on data contained in the passed ISV, allocates one or more DPV buffers (via `sysSpe4x155DPVBufferGet`) and invokes one or more callbacks (if defined and enabled). This routine is called by the application code, within `sysSpe4x155DPRTask`. Note that the callbacks are responsible for releasing the passed DPV. It is recommended that it be done as soon as possible to avoid running out of DPV buffers.

Prototype `void spe4x155DPR(void *pisv)`

Inputs `pisv` : (pointer to) ISV buffer

Outputs None

Returns None

Valid States `SPE4X155_ACTIVE`

Side Effects None

Setting the Thresholds for Callbacks: spe4x155SetThresh

This function sets the number of events that must be received (thresholds) before those events are reported via the callbacks.

Prototype `INT4 spe4x155SetThresh(sSPE4X155_HNDL deviceHandle, sSPE4X155_MASK *pthreshold)`

Inputs
`deviceHandle` : device handle (from `spe4x155Add`)
`pthreshold` : (pointer to) threshold structure

Outputs None

Returns
Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Getting the Thresholds for Callbacks: spe4x155GetThresh

This function sets the number of events that must be received (thresholds) before those events are reported via the callbacks.

Prototype `INT4 spe4x155GetThresh(sSPE4X155_HNDL deviceHandle, sSPE4X155_MASK *pthreshold)`

Inputs
`deviceHandle` : device handle (from `spe4x155Add`)
`pthreshold` : (pointer to) allocated memory

Outputs `pthreshold` : threshold structure

Returns
Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Getting the Event Counters: spe4x155GetThreshCntr

This function returns the number of events that have occurred while the thresholds have not been reached.

Prototype `INT4 spe4x155SetThreshCntr(sSPE4X155_HNDL deviceHandle, sSPE4X155_MASK *pthreshCntr)`

Inputs
`deviceHandle` : device handle (from `spe4x155Add`)
`pthreshold` : (pointer to) allocated memory

Outputs `pthreshold` : event counters

Returns
Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

5.11 Alarm, Status and Statistics Functions

Configuring the Device Statistics: spe4x155CfgStats

This function configures the device counts.

Prototype `INT4 spe4x155CfgStats(sSPE4X155_HNDL deviceHandle, sSPE4X155_CFG_CNT cfgCnt)`

Inputs
`deviceHandle` : device handle (from `spe4x155Add`)
`cfgCnt` : counters configuration block

Outputs None

Returns
Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Statistics Collection Routine: spe4x155GetCnt

This function retrieves all the device counts. This routine should be called by the application code, in the context of a task. It is the user's responsibility to ensure that this function is called often enough to prevent the device counts from saturating.

Prototype
```
INT4 spe4x155GetCnt(sSPE4X155_HNDL
deviceHandle, sSPE4X155_STAT_CNT *pcnt)
```

Inputs
`deviceHandle` : device handle (from `spe4x155Add`)
`pcnt` : (pointer to) allocated memory

Outputs
`pcnt` : current device counts

Returns
Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States
`SPE4X155_ACTIVE`

Side Effects
None

Getting Counter for SOH Block: spe4x155GetCntSOH

This function retrieves the specified device counts block. It is the user's responsibility to poll this function often enough to prevent the device counts from saturating.

Prototype
```
INT4 spe4x155GetCntSOH(sSPE4X155_HNDL
deviceHandle, UINT2 stm1,
sSPE4X155_STAT_CNT_SOH *pcntSOH)
```

Inputs
`deviceHandle` : device handle (from `spe4x155Add`)
`stm1` : STM-1 index
`pcntSOH` : (pointer to) allocated memory

Outputs
`pcntSOH` : current device counts

Returns
Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States
`SPE4X155_ACTIVE`

Side Effects
None

Getting Counters for LOH Block: spe4x155GetCntLOH

This function retrieves the specified device counts block. It is the user's responsibility to poll this function often enough to prevent the device counts from saturating.

Prototype
```
INT4 spe4x155GetCntLOH(sSPE4X155_HNDL
deviceHandle, UINT2 stm1,
sSPE4X155_STAT_CNT_LOH *pcntLOH)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`pcntLOH`	: (pointer to) allocated memory

Outputs

`pcntLOH`	: current device counts

Returns

Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE`

Side Effects None

Getting Counters for RPOH Block: spe4x155GetCntRPOH

This function retrieves the specified device counts block. It is the user's responsibility to poll this function often enough to prevent the device counts from saturating.

Prototype
```
INT4 spe4x155GetCntRPOH(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 au3,
sSPE4X155_STAT_CNT_RPOH *pcntRPOH)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`pcntRPOH`	: (pointer to) allocated memory

Outputs

`pcntRPOH`	: current device counts

Returns

Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE,`

Side Effects None

Getting Counters for TPOH Block: spe4x155GetCntTPOH

This function retrieves the specified device counts block. It is the user's responsibility to poll this function often enough to prevent the device counts from saturating.

Prototype
```
INT4 spe4x155GetCntTPOH(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 au3,
sSPE4X155_STAT_CNT_TPOH *pcntTPOH)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`pcntTPOH`	: (pointer to) allocated memory

Outputs

`pcntTPOH`	: current device counts

Returns

Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE, SPE4X155_INACTIVE`

Side Effects None

Getting Counters for Pointer Justifications: spe4x155GetCntPJ

This function retrieves the specified device counts block. It is the user's responsibility to poll this function often enough to prevent the device counts from saturating.

Prototype
```
UINT4 statSpe4x155GetCntPJ(sSPE4X155_DDB *pddb,
UINT2 stm1, UINT2 au3, sSPE4X155_STAT_CNT_PJ
*pcntPJ)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`pcntPJ`	: (pointer to) allocated memory

Outputs

`pcntPJ`	: current device counts

Returns

Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE,`

Side Effects None

Getting Current Status: spe4x155GetStatus

This function retrieves the current alarm status by reading all the alarm status registers. It is the user's responsibility to ensure that the passed structure points to an area of memory large enough to hold a copy of the status structure.

Prototype
```
INT4 spe4x155GetStatus(sSPE4X155_HNDL
deviceHandle, sSPE4X155_STATUS *palm)
```

Inputs
`deviceHandle` : device handle (from `spe4x155Add`)
`palm` : (pointer to) allocated memory

Outputs
`palm` : current alarm status

Returns
Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_DEVICE_STATE`
`SPE4X155_ERR_INVALID_ARG`

Valid States `SPE4X155_ACTIVE`

Side Effects None

Getting Current Status for IO block: spe4x155GetStatusIO

This function retrieves the clock activity from the status registers.

Prototype
```
INT4 spe4x155GetStatusIO(sSPE4X155_HNDL
deviceHandle, sSPE4X155_STATUS_IO *palmIO)
```

Inputs
`deviceHandle` : device handle (from `spe4x155Add`)
`palmIO` : (pointer to) allocated memory

Outputs
`palmIO` : current alarm status

Returns
Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_DEVICE_STATE`
`SPE4X155_ERR_INVALID_ARG`

Valid States `SPE4X155_ACTIVE`

Side Effects None

Getting Current Alarm Status for SOH block: spe4x155GetStatusSOH

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

Prototype
```
INT4 spe4x155GetStatusSOH(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, sSPE4X155_STATUS_SOH
*palmSOH)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`palmSOH`	: (pointer to) allocated memory

Outputs

`palmSOH`	: current alarm status

Returns

Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE`

Side Effects None

Getting Current Alarm Status for LOH block: spe4x155GetStatusLOH

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

Prototype
```
INT4 spe4x155GetStatusLOH(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, sSPE4X155_STATUS_LOH
*palmLOH)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`palmLOH`	: (pointer to) allocated memory

Outputs

`palmLOH`	: current alarm status

Returns

Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE`

Side Effects None

Getting Current Alarm Status for RPOH block: spe4x155GetStatusRPOH

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

Prototype `INT4 spe4x155GetStatusRPOH(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 au3, sSPE4X155_STATUS_RPOH *palmRPOH)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`palmRPOH`	: (pointer to) allocated memory

Outputs

`palmRPOH`	: current alarm status

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE`

Side Effects None

Getting Current Alarm Status for TPOH block: spe4x155GetStatusTPOH

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

Prototype `INT4 spe4x155GetStatusTPOH(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 au3, sSPE4X155_STATUS_TPOH *palmTPOH)`

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`au3`	: AU-3 index
`palmTPOH`	: (pointer to) allocated memory

Outputs

`palmTPOH`	: current alarm status

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE`

Side Effects None

5.12 Device Diagnostics

Testing Register Accesses: spe4x155DiagTestReg

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

Prototype `INT4 spe4x155DiagTestReg(sSPE4X155_HNDL deviceHandle)`

Inputs `deviceHandle` : device handle (from `spe4x155Add`)

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_PRESENT`

Side Effects None

Clearing and Setting a Line Loopback: spe4x155DiagLineLoop

This function clears and sets a Line Loopback (SLLE=1). The `spe4x155DiagLineLoop` 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 spe4x155DiagLineLoop(sSPE4X155_HNDL deviceHandle, UINT2 stm1, UINT2 enable)`

Inputs `deviceHandle` : device handle (from `spe4x155Add`)
`stm1` : STM-1 index
`enable` : sets loop if non-zero, else clears loop

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE`

Side Effects Will inhibit the flow of active data

Clearing and Setting a Serial Loopback: spe4x155DiagSerialLoop

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 spe4x155DiagSerialLoop(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 enable)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`enable`	: sets loop if non-zero, else clears loop

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`
`SPE4X155_ERR_INVALID_DEVICE_STATE`

Valid States `SPE4X155_ACTIVE`

Side Effects Will inhibit the flow of active data

Clearing and Setting a Parallel Loopback: spe4x155DiagParaLoop

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 spe4x155DiagParaLoop(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 enable)
```

Inputs

`deviceHandle`	: device handle (from `spe4x155Add`)
`stm1`	: STM-1 index
`enable`	: sets loop if non-zero, else clears loop

Outputs None

Returns Success = `SPE4X155_SUCCESS`
Failure = `SPE4X155_ERR_INVALID_DEV`
`SPE4X155_ERR_INVALID_ARG`

SPE4X155_ERR_INVALID_DEVICE_STATE

Valid States SPE4X155_ACTIVE

Side Effects Will inhibit the flow of active data

Clearing and Setting a System-Side Loopback: spe4x155DiagSysSideLineLoop

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. SPECTRA-4x155 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 spe4x155DiagSysSideLineLoop(sSPE4X155_HNDL
deviceHandle, UINT2 stm1, UINT2 au3, UINT2
enable)
```

Inputs

deviceHandle	: device handle (from spe4x155Add)
stm1	: STM-1 index
au3	: AU-3 index
enable	: sets loop if non-zero, else clears loop

Outputs None

Returns Success = SPE4X155_SUCCESS
Failure = SPE4X155_ERR_INVALID_DEV
SPE4X155_ERR_INVALID_ARG
SPE4X155_ERR_INVALID_DEVICE_STATE

Valid States SPE4X155_ACTIVE

Side Effects Will inhibit the flow of active data

5.13 Callback Functions

The SPECTRA-4x155 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 spe4x155DPR function are passed during the spe4x155Init call (inside a DIV). However the user shall use the exact same prototype. The application is left responsible for releasing the passed DPV as soon as possible (to avoid running out of DPV buffers) by calling sysSpe4x155DPVBufferRtn either within the callback function or later inside the application code.

Notifying the Application of IO Events: cbackSpe4x155IO

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 callback function's addresses are passed to the driver doing the spe4x155Init call. If the address of the callback function was passed as a NULL at initialization no callback will be made.

Prototype `void cbackSpe4x155IO(sSPE4X155_USR_CTXT usrCtxt, void *pdpv)`

Inputs

`usrCtxt`	: user context (from `spe4x155Add`)
`pdpv`	: (pointer to) DPV that describes this event

Outputs None

Returns None

Valid States `SPE4X155_ACTIVE`

Side Effects None

Notifying the Application of SOH Events: cbackSpe4x155SOH

This callback function is provided by the user and is used by the DPR to report significant SOH 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 callback function's addresses are passed to the driver doing the spe4x155Init call. If the address of the callback function was passed as a NULL at initialization no callback will be made.

Prototype `void cbackSpe4x155SOH(sSPE4X155_USR_CTXT usrCtxt, void *pdpv)`

Inputs

`usrCtxt`	: user context (from `spe4x155Add`)
`pdpv`	: (pointer to) DPV that describes this

event

Outputs None

Returns None

Valid States SPE4X155_ACTIVE

Side Effects None

Notifying the Application of LOH Events: cbackSpe4x155LOH

This callback function is provided by the user and is used by the DPR to report significant LOH 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 callback function's addresses are passed to the driver doing the `spe4x155Init` call. If the address of the callback function was passed as a NULL at initialization no callback will be made.

Prototype
```
void cbackSpe4x155LOH(sSPE4X155_USR_CTXT
usrCtxt, void *pdpv)
```

Inputs

`usrCtxt`	: user context (from `spe4x155Add`)
`pdpv`	: (pointer to) DPV that describes this event

Outputs None

Returns None

Valid States SPE4X155_ACTIVE

Side Effects None

Notifying the Application of RPOH Events: cbackSpe4x155RPOH

This callback function is provided by the user and is used by the DPR to report significant RPOH 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 callback function's addresses are passed to the driver doing the `spe4x155Init` call. If the address of the callback function was passed as a NULL at initialization no callback will be made.

Prototype
```
void cbackSpe4x155RPOH(sSPE4X155_USR_CTXT
usrCtxt, void *pdpv)
```

Inputs	`usrCtxt`	: user context (from `spe4x155Add`)
	`pdpv`	: (pointer to) DPV that describes this event
Outputs	None	
Returns	None	
Valid States	`SPE4X155_ACTIVE`	
Side Effects	None	

Notifying the Application of TPOH Events: cbackSpe4x155TPOH

This callback function is provided by the user and is used by the DPR to report significant TPOH 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 callback function's addresses are passed to the driver doing the `spe4x155Init` call. If the address of the callback function was passed as a NULL at initialization no callback will be made.

Prototype	`void cbackSpe4x155TPOH(sSPE4X155_USR_CTXT usrCtxt, void *pdpv)`	
Inputs	`usrCtxt`	: user context (from `spe4x155Add`)
	`pdpv`	: (pointer to) DPV that describes this event
Outputs	None	
Returns	None	
Valid States	`SPE4X155_ACTIVE`	
Side Effects	None	

Notifying the Application of DPGM Events: cbackSpe4x155DPGM

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 callback function's addresses are passed to the driver doing the `spe4x155Init` call. If the address of the callback function was passed as a NULL at initialization no callback will be made.

Prototype `void cbackSpe4x155DPGM(sSPE4X155_USR_CTXT usrCtxt, void *pdpv)`

Inputs

`usrCtxt`	: user context (from `spe4x155Add`)
`pdpv`	: (pointer to) DPV that describes this event

Outputs None

Returns None

Valid States `SPE4X155_ACTIVE`

Side Effects None

Pseudocode

```
IF GEN_SIG interrupt occurred
        call spe4x155DPGMGenRegen

…Application specific behavior…
```

Notifying the Application of APGM Events: cbackSpe4x155APGM

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 callback function's addresses are passed to the driver doing the `spe4x155Init` call. If the address of the callback function was passed as a NULL at initialization no callback will be made.

Prototype `void cbackSpe4x155APGM(sSPE4X155_USR_CTXT usrCtxt, void *pdpv)`

Inputs

`usrCtxt`	: user context (from `spe4x155Add`)
`pdpv`	: (pointer to) DPV that describes this event

Outputs None

Returns None

Valid States `SPE4X155_ACTIVE`

Side Effects None

6 HARDWARE INTERFACE

The SPECTRA-4x155 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 from a Device Register: sysSpe4x155Read

`sysSpe4x155Read` is the most basic hardware connection and reads 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.

Format `#define sysSpe4x155Read(base, offset)`

Prototype `UINT1 sysSpe4x155Read(void *base, UINT2 offset)`

Inputs

`base`	: base address of device being accessed
`offset`	: offset to the memory location as it appears in the hardware data-sheet

Outputs None

Returns value read from the addressed register location

Writing to a Device Register: sysSpe4x155Write

`sysSpe4x155Write` is the most basic hardware connection and writes 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.

Format `#define sysSpe4x155Write(base, offset, data)`

Prototype
```
UINT1 sysSpe4x155Write(void *base, UINT2
offset, UINT2 data)
```

Inputs

`base`	: base address of device being accessed
`offset`	: offset to the memory location as it appears in the hardware data-sheet
`data`	: data to be written

Outputs None

Returns Value written to the addressed register location

Polling a Bit: sysSpe4x155PollBit

This function simply polls a register masked data until it is zero or times out.

Format
```
#define sysSpe4x155PollBit(base, offset, mask)
```

Prototype
```
INT4 sysSpe4x155PollBit(void *base, UINT2
offset, UINT1 mask)
```

Inputs

`base`	: base address of device being accessed
`offset`	: offset to the memory location as it appears in the hardware data-sheet
`mask`	: mask to apply to byte read

Outputs None

Returns Success = 0
Failure = <any other value>

6.2 System-Specific Interrupt Servicing

The porting of an Interrupt-Service Routine (ISR) 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 (`sysSpe4x155ISRHandler`) in the interrupt vector table of the system processor. This handler shall call `spe4x155ISR` for each device that has interrupt servicing enabled, to perform the ISR related housekeeping required by each device.

During execution of the API function `spe4x155ModuleStart` / `spe4x155ModuleStop` the driver informs the application that it is time to install / uninstall this shell via `sysSpe4x155ISRHandlerInstall` / `sysSpe4x155ISRHandlerRemove`, 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 (`spe4x155Poll`) that in turn calls `spe4x155ISR`.

Installing the ISR Handler: sysSpe4x155ISRHandlerInstall

This routine installs the user-supplied Interrupt-Service Routine (ISR), `sysSpe4x155ISRHandler`, into the processor's interrupt vector table.

Format `#define sysSpe4x155ISRHandlerInstall()`

Prototype `INT4 sysSpe4x155ISRHandlerInstall(void)`

Inputs None

Outputs None

Returns Success = 0
Failure = <any other value>

Pseudocode

```
Begin
    Establish communication channel (message queue)
        for DPR.
    Spawn DPR task
    install sysSpe4x155ISRHandler in processor's
        interrupt vector table
End
```

ISR Handler: sysSpe4x155ISRHandler

This routine is invoked when one or more SPECTRA-4x155 devices raise the interrupt line to the microprocessor. This routine invokes the driver-provided routine, `spe4x155ISR`, for each device registered with the driver.

Format `#define sysSpe4x155ISRHandler()`

Prototype `void sysSpe4x155ISRHandler(void)`

Inputs None

Outputs None

Returns None

Pseudocode

```
Begin
    for each device registered with the driver
        call spe4x155ISR
        if returned ISV buffer is not NULL
            send ISV buffer to the DPR
End
```

Removing the ISR Handler: sysSpe4x155ISRHandlerRemove

This routine disables interrupt processing for this device. Removes the user-supplied Interrupt-Service Routine (ISR), `sysSpe4x155ISRHandler`, from the processor's interrupt vector table.

Format `#define sysSpe4x155ISRHandlerRemove()`

Prototype `void sysSpe4x155ISRHandlerRemove(void)`

Inputs None

Outputs None

Returns None

Pseudocode

```
Begin
        remove sysSpe4x155ISRHandler from the
                processor's interrupt vector table
        destroy DPR task
        destroy communication channel (message queue)
End
```

7 RTOS INTERFACE

The SPECTRA-4x155 driver requires the use of some Real-Time Operating System (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: sysSpe4x155MemAlloc

This function allocates specified number of bytes of memory.

Format `#define sysSpe4x155MemAlloc(numBytes)`

Prototype `UINT1 *sysSpe4x155MemAlloc(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)

Freeing Memory: sysSpe4x155MemFree

This function frees memory allocated using `sysSpe4x155MemAlloc`.

Format `#define sysSpe4x155MemFree(pfirstByte)`

Prototype `void sysSpe4x155MemFree(UINT1 *pfirstByte)`

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

Outputs None

Returns None

Copying Memory: sysSpe4x155MemCpy

This function copies a block of memory.

Format `#define sysSpe4x155MemCpy(pdst, psrc, sz)`

Prototype `void sysSpe4x155MemCpy (UINT1 *pdst, UINT1 *psrc, UINT4 sz)`

Inputs
`pdst` : (pointer to) first byte of destination block
`psrc` : (pointer to) first byte of source block
`sz` : block size

Outputs `pdst` : updated memory block

Returns None

Setting Memory: sysSpe4x155MemSet

This function initializes a block of memory with a given value.

Format `#define sysSpe4x155MemSet(pmem, val, sz)`

Prototype `void sysSpe4x155MemSet(UINT1 *pmem, UINT1 val, UINT4 sz)`

Inputs
`pmem` : (pointer to) first byte of the memory region to initialize
`val` : value to use
`sz` : size of the block to initialize

Outputs `pmem` : updated memory block

Returns None

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 `sysSpe4x155BufferStart` call.

Starting Buffer Management: sysSpe4x155BufferStart

This function alerts the RTOS that the time has come to make sure Interrupt-Service Vector (ISV) buffers and Deferred-Processing Vector (DPV) buffers are available and sized correctly. This may involve the creation of new buffer pools and it may involve nothing, depending on the RTOS.

Format `#define sysSpe4x155BufferStart()`

Prototype `INT4 sysSpe4x155BufferStart(void)`

Inputs None

Outputs None

Returns Success = 0
Failure = <any other value>

Getting an ISV Buffer: sysSpe4x155ISVBufferGet

This function gets a buffer from the RTOS that will be used by the Interrupt-Service Routine (ISR) code to create an Interrupt-Service Vector (ISV). The ISV consists of data transferred from the devices interrupt status registers.

Format `#define sysSpe4x155ISVBufferGet()`

Prototype `sSPE4X155_ISV *sysSpe4x155ISVBufferGet(void)`

Inputs None

Outputs None

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

Returning an ISV Buffer: sysSpe4x155ISVBufferRtn

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

Format `#define sysSpe4x155ISVBufferRtn(pISV)`

Prototype
```
void sysSpe4x155ISVBufferRtn(sSPE4X155_ISV
*pisv)
```

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

Outputs None

Returns None

Getting a DPV Buffer: sysSpe4x155DPVBufferGet

This function gets a buffer from the RTOS that will be used by the Deferred-Processing Routine (DPR) code to create a Deferred-Processing Vector (DPV). The DPV consists of information about the state of the device that is to be passed to the user via a callback function.

Format `#define sysSpe4x155DPVBufferGet()`

Prototype `sSPE4X155_DPV *sysSpe4x155DPVBufferGet(void)`

Inputs None

Outputs None

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

Returning a DPV Buffer: sysSpe4x155DPVBufferRtn

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

Format `#define sysSpe4x155DPVBufferRtn(pDPV)`

Prototype
```
void sysSpe4x155DPVBufferRtn(sSPE4X155_DPV
*pdpv)
```

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

Outputs None

Returns None

Stopping Buffer Management: sysSpe4x155BufferStop

This function alerts the RTOS that the driver no longer needs any of the ISV buffers or DPV buffers and that if any special resources were created to handle these buffers, they can be deleted now.

Format `#define sysSpe4x155BufferStop()`

Prototype `void sysSpe4x155BufferStop(void)`

Inputs None

Outputs None

Returns None

7.3 Timers

Sleeping a Task: sysSpe4x155TimerSleep

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

Format `#define sysSpe4x155TimerSleep(time)`

Prototype `void sysSpe4x155TimerSleep(UINT4 time)`

Inputs `time` : sleep time in milliseconds

Outputs None

Returns Success = 0
Failure = <any other value>

7.4 Preemption

Disabling Preemption: sysSpe4x155PreemptDisable

This function 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.

Format `#define sysSpe4x155PreemptDisable()`

Prototype `INT4 sysSpe4x155PreemptDisable(void)`

Inputs None

Outputs None

Returns Preemption key (passed back as an argument in `sysSpe4x155PreemptEnable`)

Re-Enabling Preemption: sysSpe4x155PreemptEnable

This function 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.

Format `#define sysSpe4x155PreemptEnable(key)`

Prototype `void sysSpe4x155PreemptEnable(INT4 key)`

Inputs `key` : preemption key (returned by `sysSpe4x155PreemptDisable`)

Outputs None

Returns None

7.5 System-Specific DPR Routine

The porting of a task between platforms is not always simple. There are many different implementations of the RTOS level parameters. In this driver, the user is responsible for creating a 'shell' (`sysSpe4x155DPRTask`) that in turn calls `spe4x155DPR` with an ISV to perform the ISR related processing that is required by each interrupting device.

During execution of the API function `spe4x155ModuleStart` / `spe4x155ModuleStop` the driver informs the application that it is time to install / uninstall this shell that needs to be supplied by the user.

DPR Task: sysSpe4x155DPRTask

This routine is installed as a separate task within the RTOS. It runs periodically and retrieves the interrupt status information sent to it by `spe4x155ISR` and then invokes `spe4x155DPR` for the appropriate device.

Format `#define sysSpe4x155DPRTask()`

Prototype `void sysSpe4x155DPRTask(void)`

Inputs None

Outputs None

Returns None

Pseudocode

```
Begin
do
wait for an ISV buffer (sent by spe4x155ISR)
call spe4x155DPR with that ISV
        loop forever
End
```

8 PORTING THE SPECTRA-4X155 DRIVER

This section outlines how to port the SPECTRA-4x155 device driver to your hardware and OS platform. However, this manual can offer only guidelines for porting the SPECTRA-4x155 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-4x155 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 header files (`inc`). The source files contain the functions and the header files contain the structures, constants and macros.

Directory	File	Description
src	spe4x155_api1.c	All the API functions that take care of module, device and profile management
	spe4x155_api2.c	All the SPECTRA-4x155 specific API functions
	spe4x155_hw.c	Hardware interface functions
	spe4x155_isr.c	Internal functions for interrupt servicing
	spe4x155_prof.c	Internal functions for profiles
	spe4x155_rtos.c	RTOS interface functions
	spe4x155_stat.c	Internal functions for statistics
	spe4x155_util.c	All the remaining internal functions
inc	spe4x155_api.h	All API headers
	spe4x155_defs.h	Driver macros, constants and definitions (such as register mapping and bit masks)
	spe4x155_err.h	SPECTRA-4x155 error codes
	spe4x155_fns.h	Prototype of non-API functions
	spe4x155_hw.h	HW interface macros and prototype
	spe4x155_rtos.h	RTOS interface macros and prototypes

Directory	File	Description
inc	spe4x155_strs.h	Driver structures
	spe4x155_typs.h	Types definitions
example	spe4x155_app.c	Sample driver callback functions and example code
	spe4x155_app.h	Prototypes, macros and structures used inside the example code
	spe4x155_debug.c	Example debug task
	spe4x155_debug.h	Example debug task defines and prototypes

8.2 Driver Porting Procedures

The following procedures summarize how to port the SPECTRA-4x155 driver to your platform. The subsequent sections describe these procedures in more detail.

To port the SPECTRA-4x155 driver to your platform:

Step 1: Port the driver's RTOS interface (page 124).

Step 2: Port the driver's hardware interface (page 126).

Step 3: Port the driver's application-specific elements (page 127).

Step 4: Build the driver.

Step 1: Porting the RTOS interface

The Real-Time Operating System (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 spe4x155_rtos.h file to the corresponding system calls that your target system supports:

Service Type	Macro Name	Description
Memory	sysSpe4x155MemAlloc	Allocates a memory block
	sysSpe4x155MemFree	Frees a memory block
	sysSpe4x155MemCpy	Copies the contents of one memory block to another
	sysSpe4x155MemSet	Fills a memory block with a specified value
Timer	sysSpe4x155TimerSleep	Delays the task execution for a given number of milliseconds
Pre-emption Lock/Unlock	sysSpe4x155PreemptDisable	Disables pre-emption of the currently executing task by any other task or interrupt
	sysSpe4x155PreemptEnable	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 spe4x155_rtos.h file with the corresponding system calls that your target system supports:

Service Type	Macro Name	Description
Buffer	sysSpe4x155BufferStart	Starts buffer management
	sysSpe4x155BufferStop	Stops buffer management
	sysSpe4x155ISVBufferGet	Gets an ISV buffer from the ISV buffer queue
	sysSpe4x155ISVBufferRtn	Returns an ISV buffer to the ISV buffer queue
	sysSpe4x155DPVBufferGet	Gets a DPV buffer from the DPV buffer queue
	sysSpe4x155DPVBufferRtn	Returns a DPV buffer to the DPV buffer queue

3. Define the following constants for your OS-specific services in `spe4x155_rtos.h`:

Task Constant	Description	Default
`SPE4X155_DPR_TASK_PRIORITY`	Deferred Task (DPR) task priority	85
`SPE4X155_DPR_TASK_STACK_SZ`	DPR task stack size, in bytes	8192

Step 2: Porting the Hardware Interface

This section describes how to modify the SPECTRA-4x155 driver for your hardware platform.

To port the driver to your hardware platform:

1. Modify the variable type definitions in `spe4x155_typs.h`.

2. Modify the low-level hardware-dependent functions and macros in the `spe4x155_hw.h` file. You may need to modify the raw read/write access macros (`sysSpe4x155Read` and `sysSpe4x155Write`) to reflect your system's addressing logic.

Service Type	Function Name	Description
Register Access	`sysSpe4x155Read`	Reads a device register given its real address in memory
	`sysSpe4x155Write`	Writes to a device register given its real address in memory
Interrupt	`sysSpe4x155ISRHandlerInstall`	Installs the interrupt handler for the OS, spawns the DPR task, and creates the DPR message queue
	`sysSpe4x155ISRHandlerRemove`	Removes the interrupt handler from the OS, destroys the DPR task and frees the DPR message queue
	`sysSpe4x155ISRHandler`	Interrupt handler for the SPECTRA-4x155 device
	`sysSpe4x155DPRTask`	Task that calls the SPECTRA-4x155 DPR

3. Define the hardware system-configuration constants in the `spe4x155_hw.h` file. Modify the following constants to reflect your system's hardware configuration:

Device Constant	Description	Default
SPE4X155_MAX_DELAY	Delay between two consecutive polls of a busy bit	100us
SPE4X155_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 SPE4X155_ERR_BASE (default = -600) in `spe4x155_err.h`.

2. Define the following application specific constants in `spe4x155_defs.h`:

Task Constant	Description	Default
SPE4X155_MAX_DEVS	The maximum number of SPECTRA-4x155 devices that can be supported by the driver	16
SPE4X155_MAX_INIT_PROFS	The maximum number of initialization profiles that can be added to the driver	20

3. Define the following constants for your OS-specific services in `spe4x155_rtos.h`

SPE4X155_MAX_ISV_BUF	The queue message depth of the queue used for passing interrupt context between the ISR handler and the DPR task	50
SPE4X155_MAX_DPV_BUF	The queue message depth of the queue used for pass interrupt context between the DPR task and the callback functions	950

4. 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 (sSPE4X155_USR_CTXT usrCtxt, void *pdpv)`

Step 4: Building the Driver

This section describes how to build the SPECTRA-4x155 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-4x155 API driver library using your make utility.

4. Link the SPECTRA-4x155 API driver library to your application code.

APPENDIX A: CODING CONVENTIONS

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

Variable Type Definitions

Table 25: 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 26 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 26: Naming Conventions

Type	Case	Naming convention	Examples
Macros	Uppercase	prefix with "m" and device abbreviation	mSPE4X155_SLICE_OFFSET
Constants	Uppercase	prefix with device abbreviation	SPE4X155_MAX_DEVS

Type	Case	Naming convention	Examples
Structures	Hungarian Notation	prefix with "s" and device abbreviation	sSPE4X155_DDB
API Functions	Hungarian Notation	prefix with device name	spe4x155Add()
Porting Functions	Hungarian Notation	prefix with "sys" and device name	sysSpe4x155Read()
Other Functions	Hungarian Notation	—	utilSpe4x155ApplyDIV()
Variables	Hungarian Notation	—	maxDevs
Pointers to variables	Hungarian Notation	prefix variable name with "p"	pmaxDevs
Global variables	Hungarian Notation	prefix with device name	spe4x155Mdb

Macros

- Macro names must be all uppercase.
- Words shall be separated by an underscore.
- The letter 'm' in lowercase is used as a prefix to specify that it is a macro, then the device abbreviation must appear.
- Example: mSPE4X155_SLICE_OFFSET is a valid name for a macro.

Constants

- Constant names must be all uppercase.
- Words shall be separated by an underscore.
- The device abbreviation must appear as a prefix.
- Example: SPE4X155_MAX_DEVS is a valid name for a constant.

Structures

- Structure names must be all uppercase.
- Words shall be separated by an underscore.

- The letter 'sʼ in lowercase must be used as a prefix to specify that it is a structure, then the device abbreviation must appear.
- Example: sSPE4X155_DDB is a valid name for a structure.

Functions

API Functions

- Naming of the API functions must follow the hungarian notation.
- The device's full name in all lowercase shall be used as a prefix.
- Example: spe4x155Add() is a valid name for an API function.

Porting Functions

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

- Naming of the porting functions must follow the hungarian notation.
- The 'sysʼ prefix shall be used to indicate a porting function.
- The device's name starting with an uppercase must follow the prefix.
- Example: sysSpe4x155Read() is a hardware / RTOS specific.

Other Functions

- Other Functions are all the remaining functions that are part of the driver and have no special naming convention. However, they must follow the hungarian notation.
- Example: utilSpe4x155ApplyDIV is a valid name for such a function.

Variables

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

File Organization

Table 27 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. `spe4x155_api1.c` or `spe4x155_api2.c`).

There are 4 different types of files:

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

Table 27: File Naming Conventions

File Type	File Name
API	`spe4x155_api1.c, spe4x155_api.h`
Hardware Dependent	`spe4x155_hw.c, spe4x155_hw.h`
RTOS Dependent	`spe4x155_rtos.c, spe4x155_rtos.h`
Other	`spe4x155_stat.c, spe4x155_util.c`

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: `spe4x155_api1.c` is the only valid name for the file that contains the first part of the API functions, `spe4x155_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: `spe4x155_hw.c` is the only valid name for the file that contains all of the hardware dependent functions, `spe4x155_hw.h` is the only valid name for the file that contains all of the hardware dependent functions headers.

- RTOS Dependent Files
- The name of the RTOS dependent files must start with the device abbreviation followed by an underscore and `'rtos'`. Eventually a number might be added at the end of the file name.
- Examples: `spe4x155_rtos.c` is the only valid name for the file that contains all of the RTOS dependent functions, `spe4x155_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: `spe4x155_isr.c` is a valid name for a file that would deal with interrupt servicing for the device.

APPENDIX B: DRIVER RETURN CODES

This appendix describes the error codes used in the SPECTRA-4X155 device driver.

Table 28: Return Codes

Return Type	Description
SPE4X155_ERR_MEM_ALLOC	Memory allocation failure
SPE4X155_ERR_INVALID_ARG	Invalid argument
SPE4X155_ERR_INVALID_MODULE_STATE	Invalid module state
SPE4X155_ERR_INVALID_MIV	Invalid Module Initialization Vector
SPE4X155_ERR_PROFILES_FULL	Maximum number of profiles already added
SPE4X155_ERR_INVALID_PROFILE	Invalid profile
SPE4X155_ERR_INVALID_PROFILE_NUM	Invalid profile number
SPE4X155_ERR_INVALID_DEVICE_STATE	Invalid device state
SPE4X155_ERR_DEVS_FULL	Maximum number of devices already added
SPE4X155_ERR_DEV_ALREADY_ADDED	Device already added
SPE4X155_ERR_INVALID_DEV	Invalid device handle
SPE4X155_ERR_INVALID_DIV	Invalid Device Initialization Vector
SPE4X155_ERR_INT_INSTALL	Error while installing interrupts
SPE4X155_ERR_INVALID_MODE	Invalid ISR/polling mode
SPE4X155_ERR_INVALID_REG	Invalid register number
SPE4X155_ERR_POLL_TIMEOUT	Time-out while polling

APPENDIX C: EVENTS

This appendix describes the events used in the SPECTRA-4x155 device driver.

Table 29: SPECTRA-4x155 Events for IO callbacks

Event	Description
SPE4X155_EVENT_IO_TROOL	Transmit reference clock out of lock
SPE4X155_EVENT_IO_RROOL	Receive reference clock out of lock
SPE4X155_EVENT_IO_RDOOL	Receive data out of lock
SPE4X155_EVENT_IO_APARRERR	ADD bus parity error
SPE4X155_EVENT_IO_RDOOL_AUX	Receive data out of lock state change

Table 30: SPECTRA-4x155 Events for SOH callbacks

Event	Description
SPE4X155_EVENT_SOH_BIPE	Section BIP error
SPE4X155_EVENT_SOH_LOS	Loss of signal
SPE4X155_EVENT_SOH_LOF	Loss of frame
SPE4X155_EVENT_SOH_OOF	Out of frame
SPE4X155_EVENT_SOH_TIU	Section trace identifier unstable
SPE4X155_EVENT_SOH_TIM	Section trace identifier mismatch
SPE4X155_EVENT_SOH_OOF_AUX	Out of frame state change
SPE4X155_EVENT_SOH_LOS_AUX	Loss of signal state change
SPE4X155_EVENT_SOH_LOF_AUX	Loss of frame state change

Table 31: SPECTRA-4x155 Events for LOH callbacks

Event	Description
SPE4X155_EVENT_LOH_BIPE	Line BIP error
SPE4X155_EVENT_LOH_REI	Line remote error indication error
SPE4X155_EVENT_LOH_SF	Signal failure
SPE4X155_EVENT_LOH_SD	Signal degrade
SPE4X155_EVENT_LOH_COSSM	Change of SSM message (Z1/S1)
SPE4X155_EVENT_LOH_COAPS	Change of APS bytes (K1, K2)
SPE4X155_EVENT_LOH_APSBF	APS byte failure
SPE4X155_EVENT_LOH_RDI	Line alarm indication signal
SPE4X155_EVENT_LOH_AIS	Line remote defect indication
SPE4X155_EVENT_LOH_RDI_AUX	Line RDI state change
SPE4X155_EVENT_LOH_AIS_AUX	Line AIS state change
SPE4X155_EVENT_LOH_SF_AUX	Signal fail.
SPE4X155_EVENT_LOH_SD_AUX	Signal degrade.

Table 32: SPECTRA-4x155 Events for RPOH callback

Event	Description
SPE4X155_EVENT_RPOH_RDI_AUX	Path RDI state change
SPE4X155_EVENT_RPOH_AIS_AUX	Path AIS state change
SPE4X155_EVENT_RPOH_PSLU_AUX	Path signal label unstable state change
SPE4X155_EVENT_RPOH_PSLM_AUX	Path signal label mismatch state change
SPE4X155_EVENT_RPOH_LOP_AUX	Loss of pointer state change
SPE4X155_EVENT_RPOH_LOM_AUX	Loss of multiframe state change
SPE4X155_EVENT_RPOH_TIU_AUX	Trace identifier unstable mode 1 state change

Event	Description
SPE4X155_EVENT_RPOH_TIM_AUX	Trace identifier mismatch state change
SPE4X155_EVENT_RPOH_LOPC_AUX	Loss of pointer concatenation state change
SPE4X155_EVENT_RPOH_AISC_AUX	Path AIS concatenation state change
SPE4X155_EVENT_RPOH_ERDI_AUX	Path enhanced remote defect indication state change
SPE4X155_EVENT_RPOH_NEWPTR	Reception of new pointer indication
SPE4X155_EVENT_RPOH_LOPC	Path loss of pointer concatenated
SPE4X155_EVENT_RPOH_LOP	Path Loss of pointer
SPE4X155_EVENT_RPOH_AISC	Path AIS concatenated
SPE4X155_EVENT_RPOH_AIS	Path AIS
SPE4X155_EVENT_RPOH_PRDI	Path remote defect indication
SPE4X155_EVENT_RPOH_BIPE	Path BIP error
SPE4X155_EVENT_RPOH_PREI	Path remote error indication
SPE4X155_EVENT_RPOH_PERDI	Change of path enhanced remote defect indication
SPE4X155_EVENT_RPOH_ILLJREQ	Illegal pointer justification request
SPE4X155_EVENT_RPOH_CONCAT	Concatenation indicator error in STS-1/STM-0/AU-3
SPE4X155_EVENT_RPOH_DISCOPA	Change of pointer alignment event
SPE4X155_EVENT_RPOH_INVNDF	Invalid NDF code
SPE4X155_EVENT_RPOH_PSE	Positive pointer justification indication received
SPE4X155_EVENT_RPOH_NSE	Negative pointer justification indication received
SPE4X155_EVENT_RPOH_NDF	Detection of NDF_enable indication.
SPE4X155_EVENT_RPOH_LOM	Loss of multiframe

Event	Description
SPE4X155_EVENT_RPOH_COMA	Change of multiframe alignment
SPE4X155_EVENT_RPOH_ESE	Elastic store error
SPE4X155_EVENT_RPOH_PPJ	DROP bus positive pointer justification inserted.
SPE4X155_EVENT_RPOH_NPJ	DROP bus negative pointer justification inserted.
SPE4X155_EVENT_RPOH_UNEQ	Path connection unequipped
SPE4X155_EVENT_RPOH_TIU	Path trace identifier unstable
SPE4X155_EVENT_RPOH_TIM	Path trace identifier mismatch
SPE4X155_EVENT_RPOH_PSLM	Path signal label mismatch
SPE4X155_EVENT_RPOH_PSLU	Path signal label unstable
SPE4X155_EVENT_RPOH_ILLPTR	Illegal Pointer
SPE4X155_EVENT_RPOH_COPSL	Change of Path Signal Label

Table 33: SPECTRA-4x155 Events for TPOH callback

Event	Description
SPE4X155_EVENT_TPOH_LOPC_AUX	Change in transmit loss of pointer concatenation state
SPE4X155_EVENT_TPOH_AISC_AUX	Change in transmit path AIS concatenation state
SPE4X155_EVENT_TPOH_AIS_AUX	Change in transmit path AIS state.
SPE4X155_EVENT_TPOH_LOP_AUX	Change in transmit loss of pointer state
SPE4X155_EVENT_TPOH_LOM_AUX	Change in transmit loss of multiframe state
SPE4X155_EVENT_TPOH_ESE	Elastic store FIFO overflow/underflow error
SPE4X155_EVENT_TPOH_PPJ	Transmit stream positive pointer justification
SPE4X155_EVENT_TPOH_NPJ	Transmit stream negative pointer justification

Event	Description
SPE4X155_EVENT_TPOH_NEWPTR	New point indication received
SPE4X155_EVENT_TPOH_LOPC	Change in transmit AU-3 loss of pointer concatenation status
SPE4X155_EVENT_TPOH_LOP	Change in transmit loss of pointer status
SPE4X155_EVENT_TPOH_AISC	Change in transmit path AIS concatenation status
SPE4X155_EVENT_TPOH_AIS	Change in transmit path AIS status
SPE4X155_EVENT_TPOH_RDI	Change in transmit path remote defect indication
SPE4X155_EVENT_TPOH_BIPE	Change in transmit path BIP error status
SPE4X155_EVENT_TPOH_REI	Change in transmit path extended remote error indication status
SPE4X155_EVENT_TPOH_ILLJREQ	Illegal justification request event
SPE4X155_EVENT_TPOH_DISCOPA	Discontinuous pointer change
SPE4X155_EVENT_TPOH_INVNDF	Invalid NDF observed in receive stream
SPE4X155_EVENT_TPOH_ILLPTR	Illegal pointer observed in receive stream
SPE4X155_EVENT_TPOH_NSE	Responded to a dec_ind indication in the receive stream
SPE4X155_EVENT_TPOH_PSE	Responded to an inc_ind indication in receive stream
SPE4X155_EVENT_TPOH_NDF	NDF enable patter observed in receive stream
SPE4X155_EVENT_TPOH_LOM	Change in loss of multiframe status in received stream
SPE4X155_EVENT_TPOH_COMA	Change in multiframe alignment
SPE4X155_EVENT_TPOH_CONCAT	Error detected in concatenation indicators of STS-1

Table 34: SPECTRA-4x155 Events for DPGM callback

Event	Description
SPE4X155_EVENT_DPGM_GENSIG	DROP generator signal
SPE4X155_EVENT_DPGM_MONERR	DROP monitor error detected in received PRBS byte
SPE4X155_EVENT_DPGM_MONSYNC	DROP monitor synchronization state change
SPE4X155_EVENT_DPGM_MONSIG	DROP monitor signal verification state by a slave monitor

Table 35: SPECTRA-4x155 Events for APGM callback

Event	Description
SPE4X155_EVENT_APGM_GENSIG	ADD generator signal
SPE4X155_EVENT_APGM_MONERR	ADD monitor error detected in received PRBS byte
SPE4X155_EVENT_APGM_MONSYNC	ADD monitor synchronization state change
SPE4X155_EVENT_APGM_MONSIG	ADD monitor signal verification state by a slave monitor

The cause value passed to the callback contains the following values, where Aux refers to any event ending in _AUX with a mask bit ending in Aux, and Non-Aux refers to all other events.

Table 36: SPECTRA-4x155 Event Cause Values

Event Type	Cause value
IO (except TROOL)	STM-1 #
SOH	STM-1 #
LOH	STM-1 #
RPOH	Non-Aux: 0x01XY Aux: 0x02XY where X=AU-3 # and Y=STM-1 #
TPOH	Non-Aux: 0x01XY Aux: 0x02XY where X=AU-3 # and Y=STM-1 #

DPGM	0x00XY where X=AU-3 # and Y=STM-1 #
APGM	0x00XY where X=AU-3 # and Y=STM-1 #

LIST OF TERMS

APPLICATION: Refers to protocol software used in a real system as well as validation software written to validate the SPECTRA-4x155 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 that inform it of the events that caused the ISR in the first place. Because this function is operating at the task level, the user can decide on its importance in the system, relative to other functions.

DEVICE: One SPECTRA-4x155 Integrated Circuit. There can be many devices, all served by this one driver module

- DIV (Device Initialization Vector): 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): 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-4x155 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): Structure that holds the Configuration Data for this module.

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

ACRONYMS

APGM: Add Bus PRBS Generator and Monitor

API: Application Programming Interface

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: Interrupt-Service (routine) Vector

LOH: Line Overhead

LOP: Loss of Pointer

MDB: Module Data Block

MIV: Module Initialization Vector

POH: Path Overhead

PRSB: Pseudo Random Bit Sequence

RPOH: Receive Path Overhead

RTOS: Real-Time Operating System

SOH: Section Overhead

TPOH: Transmit Path Overhead

TSI: Time-Slot Interchange

INDEX

A

api functions

C

callbacks

constants

D

device

driver

E

enumeration

error

G

global variables

M

module

P

pointers

poll

porting functions

S

structures