PM5372

TSE

DRIVER MANUAL

PROPRIETARY AND CONFIDENTIAL

RELEASE

ISSUE 2: NOVEMBER, 01

ABOUT THIS MANUAL AND TSE

This manual describes the TSE 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 TSE devices
- Modify and add to the TSE driver's functions
- Port the TSE driver to a particular platform.

References

For more information about the TSE driver, see the driver's release Notes. For more information about the TSE device, see the documents listed in Table 1 and any related errata documents.

Table 1: Related Documents

Document Number	Document Name
PMC-1991258	PM5372 Transmission Switch Element 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	October 2000	Document created
Issue 2	November 2001	Added the APIs: tseSetOnePage(), tseGetOnePage(), tsePortSetMaskMode() tseSetPage() and tseGetPage() API descriptions updated. tseRmSlot() and tseClrSlot() API descriptions updated. tseInit() now uses h/w defaults for the DIV when the pdiv parameter is NULL. Documented API error codes by replacing <TSE_ERROR CODE> with explicit return values.

Legal Issues

PMC-2001402 (R2), ref PMC-991543 (P1)

Contacting PMC-Sierra

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

Tel: +1-604-415-6000
Fax: +1-604-415-6200

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 TSE driver manual describe the TSE 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 and Real-Time Operating System environments.

Section 3 of this document, Software Architecture, defines the software architecture of the TSE 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 TSE 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 TSE 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, this manual provides a brief guide to porting the TSE device driver to your hardware and RTOS platform (page 91). In addition, an extensive Appendix (beginning on page 98) and Index (page 108), provide you with useful reference information.

2 DRIVER FUNCTIONS AND FEATURES

This section describes the main functions and features supported by the TSE 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 RTOS resources needed by the driver such as timers and semaphores (except for memory, which is allocated during the Open call).

Closing the driver module involves de-allocating all RTOS resources allocated by the driver without changing the amount of memory allocated to it.

Add/Delete Device

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 first resets then initializes the device and any associated context information about it. The driver uses this context information to control and monitor the TSE device.

Device Update

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

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 API invocations.

On the contrary, de-activating a device removes it from its operating state and 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 TSE device. Routines should be invoked often enough to avoid letting the counters rollover.

Traffic control and configuration

Functions are available to control the data flow to/from the LVDS serial links.

2.2 TSE Specific Driver Functions

These functions provide control and monitoring of the various sections of the TSE device. These sections are generally enabled or disabled and configured by the MODE specified during device initialization. Changes to these registers that would violate the characteristics of the initialized mode should be disallowed.

Time Slot Interchange and Space Switch

The Time Slot Interchange (TSI) is where the space-time slot relationship across the TSE is defined and manipulated.

- Set/Get map mode
- Set/Get active memory page
- Copy memory page
- Map TSI
- Remove/Clear TSI mapping
- Get source TSI given a destination TSI
- Get destination TSIs given a source TSI
- Is a TSI multicast
- Set idle fill data for a TSI

Port Alarm, Status and Statistics

The TSE device driver has the capability to collect and report both port and device level status and statistics. The following functions enable the collection and reporting of both port level status and statistics.

- Get port statistics and status
- Get port delta statistics
- Get/Set port thresholds
- Clear port statistics

Device Alarm, Status and Statistics

The TSE device driver has the capability to collect and report both port and device level status and statistics. The following functions enable the collection and reporting of both device level status and statistics.

- Get device statistics and status
- Get device delta statistics
- Get/Set device thresholds
- Clear device statistics
- Clear port statistics

Device Configuration

The TSE device has several device level modes. The following APIs allow these modes to be reconfigured:

- Set/Get device configuration

Port Configuration

The TSE device has 64 ports in each direction (ingress and egress). The following APIs allow these ports to be reconfigured:

- Set/Get port configuration

8b/10b Decoder/Encoder

The TSE device driver has the capability to force certain errors on the 8b/10b ports. The following APIs are intended to give access to these features.

- Force out of character alignment
- Force out of frame alignment
- Force line code violation

Receive 8b/10b Frame Aligner

Functions to control all 64 frame aligners that perform 8b/10b character alignment and STS-12 frame alignment:

- Insertion of AIS high order path alarm
- Control active polarity of incoming data stream
- Report number of line code violation
- Force error operations for device diagnostics: out-of-character alignment, out-of-frame alignment

Transmit 8b/10b Disparity Encoder

Functions to configure and control the following:

- FIFO centering
- Test pattern and J0 byte insertion

Device Diagnostics

- Device register read / write test

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:

- Stuck-at condition – Monitor Inactivity in system clock activity
- Lock state change in the clock synthesis unit

- Change of active page in the space switch stage, ingress time switch element and egress time switch element
- Line code violation, out-of-character, out-of-frame alignment error, & FIFO underrun/overrun errors from receive 8b/10b frame aligner
- FIFO underrun/overrun errors from transmit 8b/10b disparity encoder

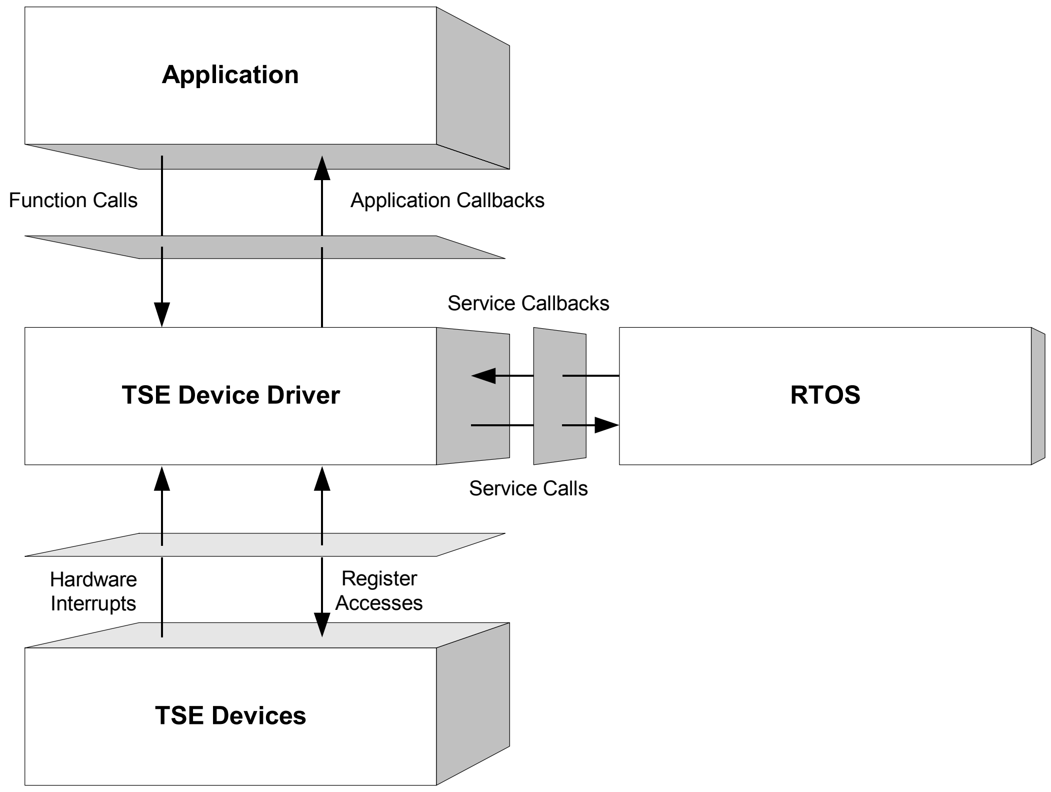
3 SOFTWARE ARCHITECTURE

This section describes the software architecture of the TSE 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 TSE 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 TSE 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 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 ISR and the DPR

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

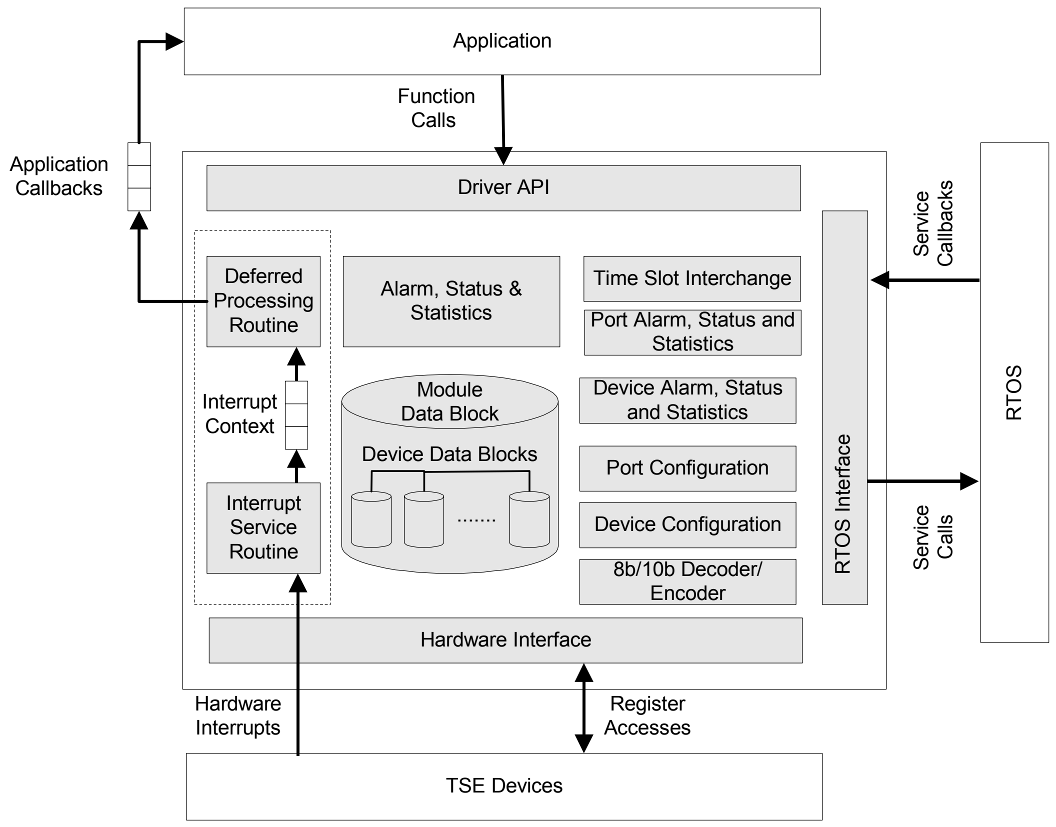
Figure 2 illustrates the top level architectural components of the TSE device driver. This architecture supports both polled and interrupt-driven operation of the driver. In polled operation, the ISR is called periodically. In interrupt operation, the interrupt directly triggers the ISR.

The driver includes ten main components:

- Module and device(s) data-blocks
- Interrupt-service routine
- Deferred-processing routine
- Alarm, status and statistics
- Time Slot Interchange
- Device Alarm, Status and Statistics
- Port Alarm, Status and Statistics
- Device Configuration
- Port Configuration

- 8b/10b Decoder/Encoder

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 TSE 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 TSE device that is registered. There is one DDB per device and there is a limit on the number of DDBs available. This 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 TSE driver provides an ISR called `tseISR` that checks if there is any valid interrupt condition present for the device. This function is 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 `tseISR` 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 TSE driver provides a DPR called `tseDPR` 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 `tseDPR`.

Example implementations of a DPR task and functions that install and remove it are provided as a reference in section 7.3. 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.

Time Slot Interchange and Space Switch

The Time Slot Interchange (TSI) is where the space-time slot relationship across the TSE is defined and manipulated.

Port Alarm, Status and Statistics

The TSE device driver has the capability to collect and report both port level status and statistics.

Device Alarm, Status and Statistics

The TSE device driver has the capability to collect and report both device level status and statistics.

Device Configuration

The TSE device has several device level modes that can be configured.

Port Configuration

The TSE device has 64 ports in each direction. These ports can be reconfigured. Configurations include controls like line inversion and enabling/disabling of ports.

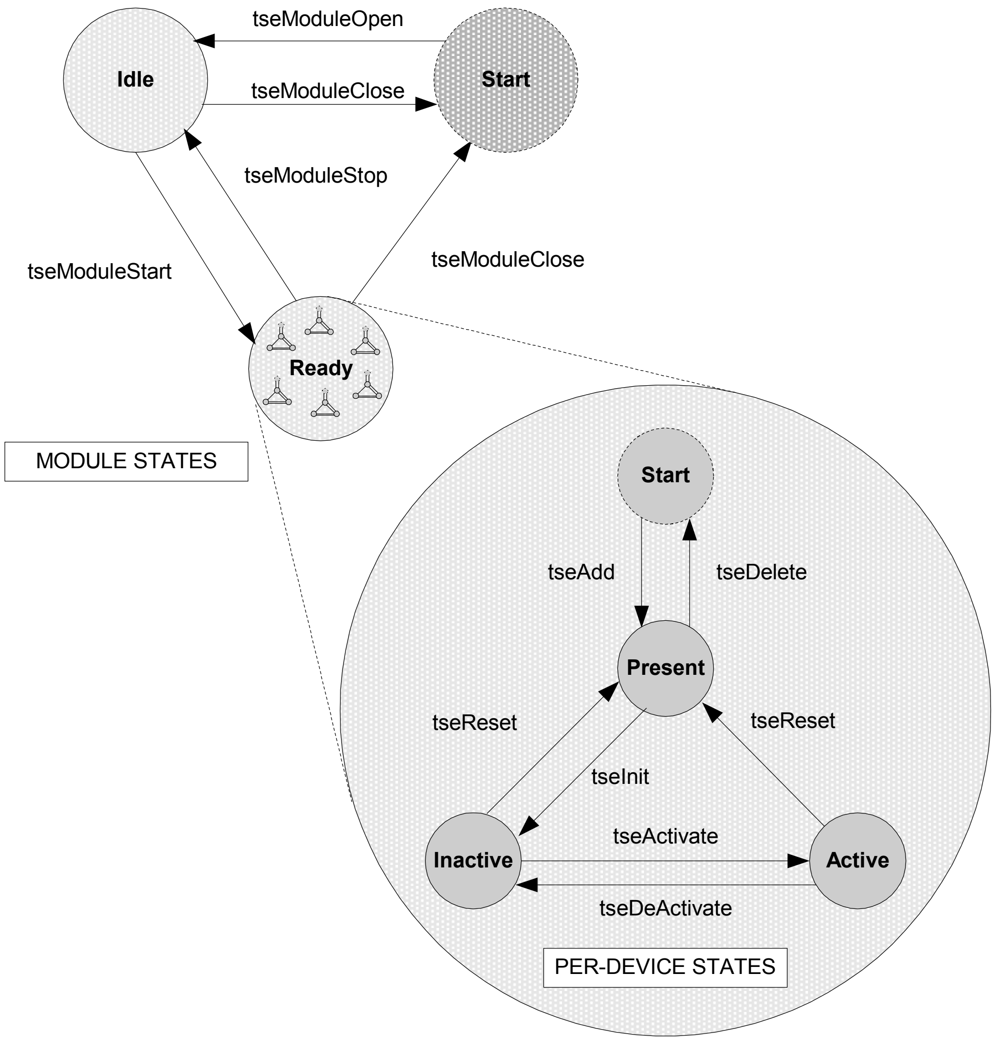
8b/10b Decoder/Encoder

The TSE device driver has the capability to force errors on the 8b/10b ports.

3.3 Software States

Figure 3 shows the software state diagram for the TSE 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 TSE 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), 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 TSE 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 0 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 with the device and updated with the user context, and a device handle has been given to the user. In this state, the driver 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

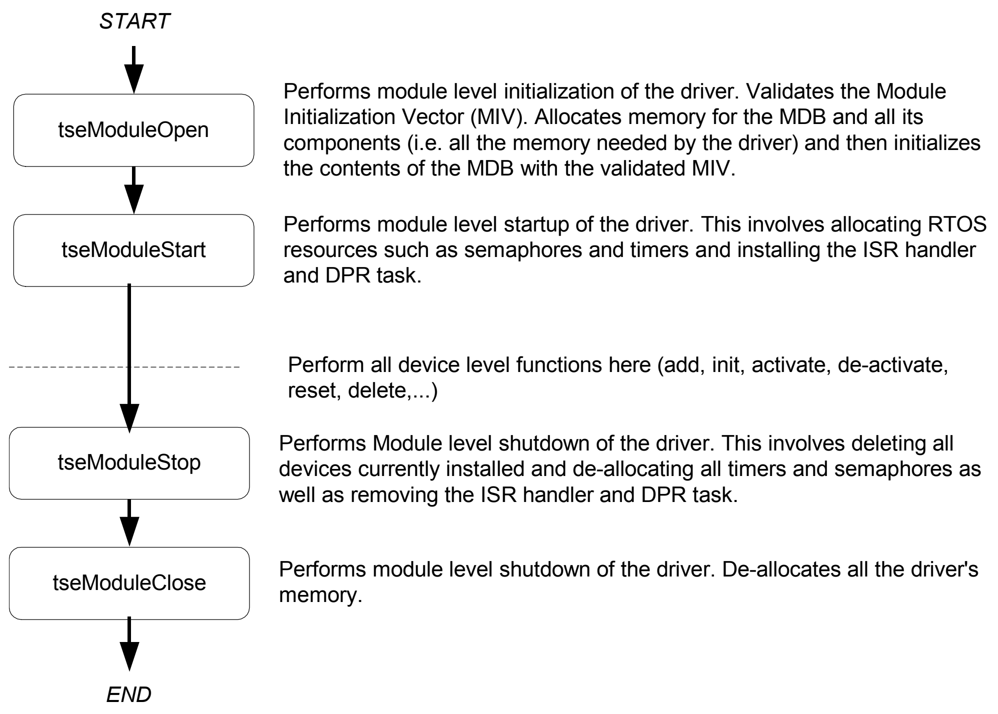
This section describes the main processing flows of the TSE driver components.

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

Module Management

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

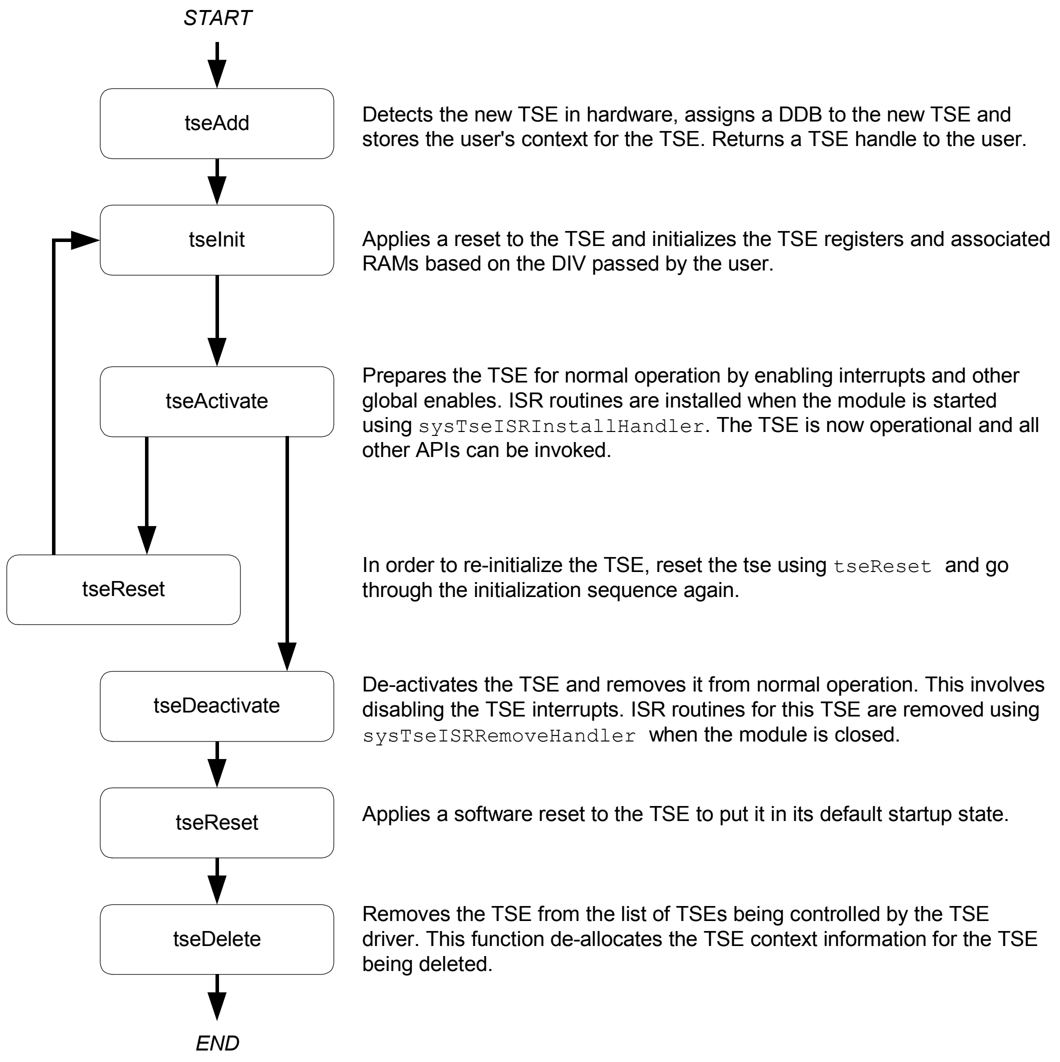
Figure 4: Module Management Flow Diagram


START
tseModuleOpen
Performs module level initialization of the driver. Validates the Module Initialization Vector (MIV). Allocates memory for the MDB and all its components (i.e. all the memory needed by the driver) and then initializes the contents of the MDB with the validated MIV.
tseModuleStart
Performs module level startup of the driver. This involves allocating RTOS resources such as semaphores and timers and installing the ISR handler and DPR task.
Perform all device level functions here (add, init, activate, de-activate, reset, delete,...)
tseModuleStop
Performs Module level shutdown of the driver. This involves deleting all devices currently installed and de-allocating all timers and semaphores as well as removing the ISR handler and DPR task.
tseModuleClose
Performs module level shutdown of the driver. De-allocates all the driver's memory.
END

Device Management

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

Figure 5: Device Management Flow Diagram


START
tseAdd
Detects the new TSE in hardware, assigns a DDB to the new TSE and stores the user's context for the TSE. Returns a TSE handle to the user.
tseInit
Applies a reset to the TSE and initializes the TSE registers and associated RAMs based on the DIV passed by the user.
tseActivate
Prepares the TSE for normal operation by enabling interrupts and other global enables. ISR routines are installed when the module is started using sysTseISRInstallHandler. The TSE is now operational and all other APIs can be invoked.
tseReset
In order to re-initialize the TSE, reset the tse using tseReset and go through the initialization sequence again.
tseDeactivate
De-activates the TSE and removes it from normal operation. This involves disabling the TSE interrupts. ISR routines for this TSE are removed using sysTseISRRemoveHandler when the module is closed.
tseReset
Applies a software reset to the TSE to put it in its default startup state.
tseDelete
Removes the TSE from the list of TSEs being controlled by the TSE driver. This function de-allocates the TSE context information for the TSE being deleted.
END

3.5 Interrupt Servicing

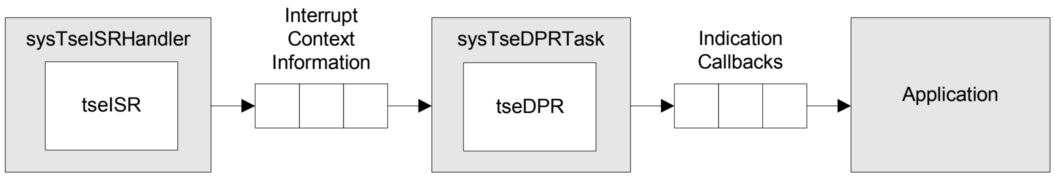
The TSE driver services device interrupts using an interrupt service routine (ISR) that traps interrupts and a deferred processing routine (DPR) that actually processes the interrupt conditions and clears them. This lets the ISR execute quickly and exit. Most of the time-consuming processing of the interrupt 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 TSE driver.

The driver provides system-independent functions, `tseISR` and `tseDPR`. You must fill in the corresponding system-specific functions, `sysTSEISRHandler` and `sysTSEDPRTask`. The system-specific functions isolate the system-specific communication mechanism (between the ISR and DPR) from the system-independent functions, `tseISR` and `tseDPR`.

Figure 6 illustrates the interrupt service model used in the TSE driver design.

Figure 6: Interrupt Service Model



Note: Instead of using an interrupt service model, you can use a polling service model in the TSE driver to process the device's event-indication registers (see page 32).

Calling tseISR

An interrupt handler function, which is system dependent, must call `tseISR`. 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 (`sysTSEISRHandler`) appears on page 85. You can customize this example implementation to suit your needs.

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

The `tseISR` function reads from the master interrupt-status of the TSE and disables the interrupt cause. If at least one valid interrupt condition is found, then `tseISR` fills an Interrupt Service Vector (ISV) with this status information as well as the current device handle. The `tseISR` function also clears and disables device interrupts as they are detected. The causes are cleared in the DPR task. The `sysTSEISRHandler` function is then responsible for sending 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 by the `sysTSEISRHandler`.

Calling tseDPR

The `sysTSEDPRTask` 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 TSE 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, `sysTSEDPRTask` calls the DPR (`tseDPR`) with the received ISV.

Then `tseDPR` processes the status information and takes appropriate action based on the specific interrupt condition detected and reads the miscellaneous interrupt-status registers and then re-enables the interrupt cause. The nature of this processing can differ from system to system. Therefore, `tseDPR` 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 `tseDelete`. Also, ensure that the callback function is non-blocking because the DPR task executes while TSE interrupts are disabled. You can customize these callbacks to suit your system. See page 82 for example implementations of the callback functions.

Note: Since the `tseISR` and `tseDPR` 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, `sysTSEISRHandler` and `sysTSEDPRTask`. When the driver calls `sysTSEISRHandlerInstall`, the application installs `sysTSEISRHandler` in the interrupt vector table of the processor, and the `sysTSEDPRTask` function is spawned as a task by the application. The `sysTSEISRHandlerInstall` function also creates the communication channel between `sysTSEISRHandler` and `sysTSEDPRTask`. This communication channel is most commonly a message queue associated with `sysTSEDPRTask`.

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

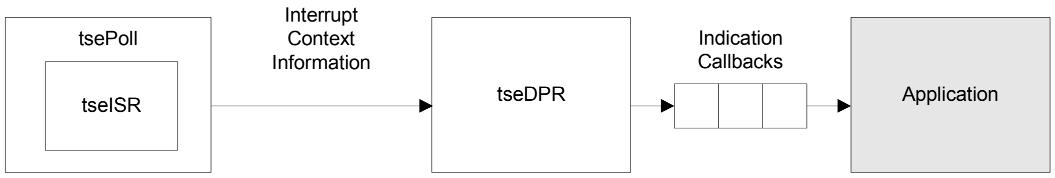
As a reference, this manual provides example implementations of the interrupt installation and removal functions on pages 85 and 90. You can customize these prototypes to suit your specific needs.

Calling tsePoll

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

Figure 7 illustrates the polling service model used in the TSE driver design.

Figure 7: Polling Service Model



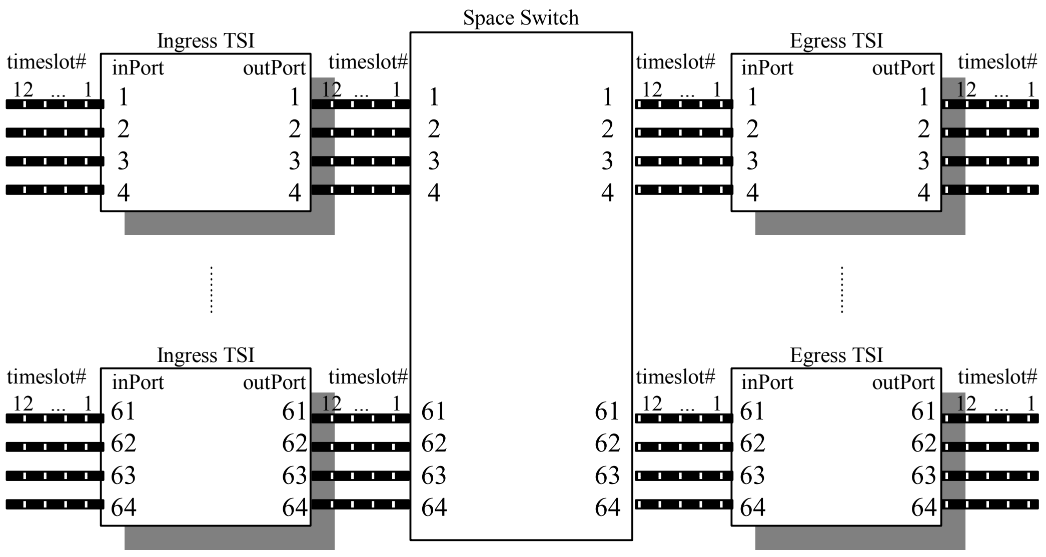
In polling mode, the application is responsible for calling `tsePoll` often enough to service any pending error or alarm conditions. When `tsePoll` is called, the `tseISR` function is called internally.

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

3.6 Theory of Operation

Time Slot Mapping

Figure 8: Time Slot Interchange and Space Switch Model

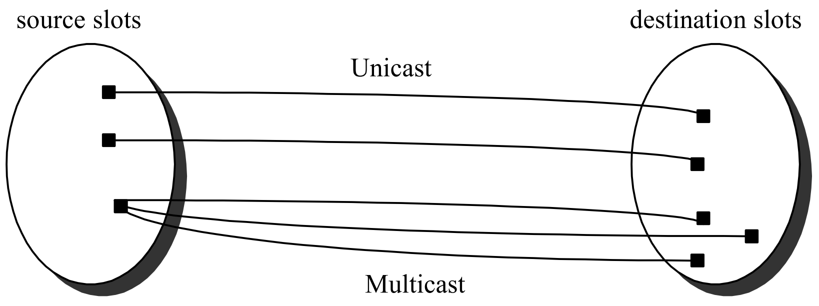


TSE has a total of 16 time slot interchange units (TSI) for time slot mapping on the ingress and egress.

Mapping is defined at STS-1 granularity; however, a valid mapping must still fit into the required time slot map in a manner mandated by the data rate of the channel. The user is responsible for maintaining data integrity when redefining the connection map.

Time slot mapping can be viewed as a process of mapping source space timeslot to destination space timeslot through both the ingress TSI and the space switch, and out the egress TSI. It is equivalent to establishing a one-to-one mapping or one-to-many mapping between the source slots and the destination slots, depending on whether the connection is unicast or multicast.

Figure 9: Space-time Slot Mapping, Multicast and Unicast



`tseMapSlot` establishes such mapping between the source space-time slot and the destination space-time slot(s). `tseRmSlot` disconnects established connection between the given source and destination slots. `tseClrSlot` clears all connections for the given source slot.

Function `tseGetDestSlot` returns the destination slot(s) given the source slot. `tseGetSrcSlot` returns the source slot given the destination slot. `tseIsMulticast` verifies if the given slot is mapped to multiple destination slots. `tseSetMapMode` sets the global mapping mode of all the TSIs in the device. There are two valid modes, user-defined or bypass. Bypass mode puts the chip in a through mode and time-slot rearrangement will not take place. If user-defined mode is selected, time slots will be re-arranged based on the connection map inside the device. `tseSetMapMode` retrieves the current mapping mode of the device.

There are two connection pages in each TSI, page 0 and 1. `tseSetPage` provides software control of the active connection memory page in the TSI. The given page is exclusive-ORed with either the hardware pin TCMP (controls ingress TSIs) or OCMP (controls egress TSIs) to determine which active page is currently active. `tseGetPage` queries the current active connection page. For connection page synchronization, `tseCopyPage` overwrites one connection page with the other within the TSI block.

Software select page	Hardware pin xCMP	Active page
0	0	0
0	1	1

Software select page	Hardware pin xCMP	Active page
1	0	1
1	1	0

Error Insertion

User may force line code violation (LCV) in the 8B/10B encoders and the disparity encoders. `tseForceLcv` may be invoked to force LCV in the 8B/10B encoders. User may insert known test patterns into the following blocks for further diagnostics: disparity encoders and transmit TSIs. For the transmit TSIs, `tseInsIdleData` introduces a known data pattern into the data stream.

User may also introduce a series of out of synchronization conditions in the 8B/10B decoders. `tseForceOutOfChar` forces out of character alignment in the block which will then attempt to realign with the alignment character (K28.5) in the data stream. In addition, out-of-frame error may be introduced by invoking `tseForceOutOfFrame`. The decoder will again attempt to resynchronize with the alignment character (K28.5).

Thresholds

The threshold mechanism allows the user to reduce the number of callbacks for a given statistic such as line code violation. A call back to the application will be made when the specified threshold is reached. The application functions defined in the DDB by the members `cbackTsePort` and `cbackTseDevice` will be invoked based on configured thresholds.

The threshold can be dynamically changed via `tsePortSetThresh` or `tseDeviceSetThresh`. The threshold values can be retrieved via `tsePortGetThresh` or `tseDeviceGetThresh`.

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 TSE device driver uses to store initialization, configuration and statistics information. For more information on naming conventions, see section 0.

4.1 Constants

The following Constants are used throughout the driver code:

- `<TSE ERROR CODES>`: error codes used throughout the driver code, returned by the API functions and used in the global error number field of the MDB and DDB.
- `TSE_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.
- `TSE_MOD_START`, `TSE_MOD_IDLE`, `TSE_MOD_READY`: are the three possible module states (stored in `stateModule`).
- `TSE_START`, `TSE_PRESENT`, `TSE_ACTIVE`, `TSE_INACTIVE`: are the four possible device states (stored in `stateDevice`).
- `TSE_MAX_PORTS`: is the maximum number of ports.
- `TSE_MAX_IE_BLOCKS`: is the maximum number of Ingress/Egress blocks.
- `TSE_MAX_TSLOTS`: is the maximum number timeslots.
- `TSE_MAX_CSU_BLOCKS`: is the maximum number of CSUs.
- `TSE_J0MASK_ALLOW, TSE_J0MASK_DENY, TSE_J0MASK_DENY_REORDER`: are the possible masking modes for the enum `eTSE_J0MASK_MODE`

4.2 Data Structures

The following are the main data structures used by the TSE driver. They are of three types:

- Structures that are passed by the application
- Structures that are in the driver's allocated memory
- Structures that are passed through RTOS buffers

Structures Passed by the Application

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

Module Initialization Vector: MIV

Passed via the `tseModuleOpen` 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 TSE_MAX_DEVS (see section 4.1).

Table 1: TSE Module Initialization Vector: sTSE_MIV

Field Name	Field Type	Field Description
perrModule	INT4 *	An output: a pointer to errModule in the MDB
maxDevs	UINT2	Maximum number of devices supported during this session

Device Initialization Vector: DIV

Passed via the tseInit call, this structure contains all the information needed by the driver to initialize a TSE device.

- pollISR is a flag that indicates the type of interrupt servicing the driver is to use. The choices are 'polling' (TSE_POLL_MODE), and 'interrupt driven' (TSE_ISR_MODE). When configured in polling the Interrupt capability of the device is NOT used, and the user is responsible for calling devicePoll periodically. The actual processing of the event information is the same for both modes.
- cbackTseDevice, cbackTsePort are used to pass the address of application functions that will be used by the DPR to inform the application code of pending events. If these fields are set as NULL, then any events that might cause the DPR to 'call back' to the application will be processed during ISR processing but ignored by the DPR.

Table 2: TSE Device Initialization Vector: sTSE_DIV

Field Name	Field Type	Field Description
pollISR	eTSE_POLL	Indicates the type of ISR / polling to do
cbackTseDevice	sTSE_CBACK	Address for the callback function for Device Events
cbackTsePort	sTSE_CBACK	Address for the callback function for Port Events
iCfgPort [TSE_MAX_PORTS]	sTSE_CFG_PORT	TSE port configuration block
iCfgDevice	sTSE_CFG_DEVICE	TSE device configuration block
iThreshPort [TSE_MAX_PORTS]	sTSE_CNTR_PORT	TSE port threshold configuration

Field Name	Field Type	Field Description
iThreshDevice	sTSE_CNTR_DEVICE	TSE device threshold configuration

TSI Connection Map: CONMAP

Used in the DDB for storing the TSI connection mapping for TSE device

Table 3: TSE TSI connection map data structure: sTSE_ CONMAP

Field Name	Field Type	Field Description
pg [TSE_MAX_PAGE]	sTSE_CONPAGE	Connection maps consist of multiple connection pages

TSI Connection Map: CONPAGE

Used in the CONMAP for storing the TSI connection mapping for TSE device

Table 4: TSE TSI connection page data structure: sTSE_ CONPAGE

Field Name	Field Type	Field Description
dstSlot [TSE_MAX_PORTS+1] [TSE_MAX_TSLOTS+1]	sTSE_SLOT	Connection pages consist of slots mapping (both arrays start indexing from 1)

TSI Connection Map: SPTSLOT

Used in the TSI APIs for representing port and timeslot relationships for TSE device.

Table 5: TSE space-time slot data structure: sTSE_SPTSLOT

Field Name	Field Type	Field Description
numPort	UINT1	Port number (1-64)
numTS	UINT1	Time slot number (1-12)
spaceSwPort	UINT1	Space switch port number (1-64) (must be within the same TSI)
spaceSwTS	UINT1	Space switch time slot number (1-12)

ISR Enable/Disable Mask

Passed via the `tseSetMask`, `tseGetMask` and `tseClearMask` calls, this structure contains all the information needed by the driver to enable and disable any of the interrupts in the TSE

Table 6: TSE ISR Mask: sTSE_MASK

Field Name	Field Type	Field Description
`csulocke[TSE_MAX_CSU_BLOCKS]`	`UINT1`	Interrupt CSU lock status interrupts (0 = disable, 1 = enable)
`sswtpage`	`UINT1`	Interrupt enable SSWT page switch (0 = disable, 1 = enable)
`r8faooc[TSE_MAX_PORTS]`	`UINT1`	Interrupt enable R8FA out of character alignment (0 = disable, 1 = enable)
`r8faoof[TSE_MAX_PORTS]`	`UINT1`	Interrupt enable R8FA out of frame alignment (0 = disable, 1 = enable)
`r8falcv[TSE_MAX_PORTS]`	`UINT1`	Interrupt enable R8FA line code violation (0 = disable, 1 = enable)
`r8fafifo[TSE_MAX_PORTS]`	`UINT1`	Interrupt enable R8FA FIFO underrun/overrun (0 = `disable`, 1 = enable)
`itsepage[TSE_MAX_IE_BLOCKS]`	`UINT1`	Interrupt enable ingress TSIs interrupts (0 = disable, 1 = enable)
`etsepage[TSE_MAX_IE_BLOCKS]`	`UINT1`	Interrupt enable egress TSIs interrupts (0 = disable, 1 = enable)
`t8defifo[TSE_MAX_PORTS]`	`UINT1`	Interrupt enable T8DE FIFO underrun/overrun (0 = disable, 1 = enable)

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), 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.

- `mSignature:` When this field contains `TSE_MDB_SIGNATURE`, it indicates that this structure has been properly initialized and may be read by the user.
- `errModule:` Most of the module API functions return a specific error code directly. When the returned code is `TSE_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.
- `stateModule:` Contains the current state of the module and could be set to: `TSE_MOD_START`, `TSE_MOD_IDLE` or `TSE_MOD_READY.`

Table 7: TSE Module Data Block: sTSE_MDB

Field Name	Field Type	Field Description
`mSignature`	`UINT4`	An indicator that this structure is initialized and valid.
`errModule`	`INT4`	Global error Indicator for module calls
`maxDevs`	`UINT2`	Maximum number of devices supported
`numDevs`	`UINT2`	Number of devices currently registered
`stateModule`	`eTSE_MOD_STATE`	Module state; can be `TSE_MOD_START`, `TSE_MOD_IDLE` or `TSE_MOD_READY`
`pmDDBSem`	`void *`	Semaphore that locks access to the DDB array (for adding and deleting)
`pddb`	`sTSE_DDB *`	(array of) Device Data Blocks (DDB) in context memory

Device Data Block: DDB

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

- `dSignature:` When this field contains `TSE_DDB_SIGNATURE`, indicates that this structure has been properly initialized and may be read by the user.
- `errDevice:` Most of the device API functions return a specific error code directly. When the returned code is `TSE_FAILURE`, this indicates that the top-level function was not able to carry the specific error code back top 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.

- `stateDevice:` Contains the current state of the device and could be set to: `TSE_START`, `TSE_PRESENT`, `TSE_ACTIVE` or `TSE_INACTIVE`.
- `usrCtxt:` 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 `tseAdd` 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 8: TSE Device Data Block: sTSE_DDB

Field Name	Field Type	Field Description
dSignature	UINT4	An indicator that this structure is initialized and valid
errDevice	INT4	Global error indicator for device calls
baseAddr	UINT2 *	Base address of the TSE device
usrCtxt	sTSE_USR_CTXT	Application defined parameter
stateDevice	eTSE_DEV_STATE	Device State; can be one of the following: TSE_START, TSE_PRESENT, TSE_ACTIVE or TSE_INACTIVE
dDiv	sTSE_DIV	Device Initialization Vector
dMask	sTSE_MASK	Interrupt enable mask
pdStatDevSem	void *	Device statistics update semaphore
pdStatPortSem	void *	Port statistics update semaphore
dStatsDevice	sTSE_STAT_DEVICE	Device statistics
dStatPort [TSE_MAX_PORTS]	sTSE_STAT_PORT	Port statistics
mapMode	eTSE_TSIMODE	Specifies bypass or user configurable mode
dSswt	sTSE_CONMAP	SSWT connection map
dItse	sTSE_CONMAP	ITSE connection map
dEtse	sTSE_CONMAP	ETSE connection map
dPageNum	UINT1	Current active page

Field Name	Field Type	Field Description
dItPrevInsIdle [TSE_MAX_PORTS]	UINT2	Previous ITSE mapping before tseInsIdleData() is called
dEtPrevInsIdle [TSE_MAX_PORTS]	UINT2	Previous ETSE mapping before tseInsIdleData() is called

Counts Block: PORT COUNTS

The sTSE_CNTR_PORT is the structure for accumulating port counts.

Table 9: TSE Port Counts Block: sTSE_CNTR_PORT

Field Name	Field Type	Field Description
cpLCVhwreg	UINT4	Rx linecode violation counts from h/w register
cpLineCodeVio	UINT4	Rx Line code violation interrupts
cpOutOfChar	UINT4	Rx Out of character interrupts
cpOutOfFrame	UINT4	Rx Out of frame interrupts
cpRxFifoErr	UINT4	Rx FIFO error interrupts
cpTxFifoErr	UINT4	Tx FIFO error interrupts

Statistics Block: PORT STATS

The sTSE_STAT_PORT is the top-level structure for port statistics.

Table 10: TSE Port Statistics Block: sTSE_STAT_PORT

Field Name	Field Type	Field Description
ipThresh	sTSE_CNTR_PORT	Thresholds
ipCount	sTSE_CNTR_PORT	Counts
ipDelta	sTSE_CNTR_PORT	Delta counts

Counts Block: DEVICE COUNTS

The sTSE_CNTR_DEVICE is the structure for accumulating device counts.

Table 11: TSE Device Counts Block: sTSE_CNTR_DEVICE

Field Name	Field Type	Field Description
`cdLocke[TSE_MAX_CSU_BLOCKS]`	`UINT4`	CSU lock interrupts
`cdSswtpage`	`UINT4`	SSWT active memory page switch interrupts
`cdItsepage[TSE_MAX_IE_BLOCKS]`	`UINT4`	ITSE page switch interrupts
`cdEtsepage[TSE_MAX_IE_BLOCKS]`	`UINT4`	ETSE page switch interrupts

Statistics Block: DEVICE STATS

The `sTSE_STAT_DEVICE` is the top-level structure for device statistics.

Table 12: TSE Port Statistics Block: sTSE_STAT_DEVICE

Field Name	Field Type	Field Description
`idThresh`	`sTSE_CNTR_DEVICE`	Thresholds
`idCount`	`sTSE_CNTR_DEVICE`	Counts
`idDelta`	`sTSE_CNTR_DEVICE`	Delta counts

Device Status

The Device Status structure stores the instantaneous device status.

Table 13: TSE Device Status: sTSE_STATUS_DEVICE

Field Name	Field Type	Field Description
`sdLockv[TSE_MAX_CSU_BLOCKS]`	`UINT1`	CSU lock status
`sdSswtpage`	`UINT1`	SSWT active page
`sdSysClkA`	`UINT1`	SYSCLK active

Port Status

The Port Status structure stores the instantaneous port status.

Table 14: TSE Port Status: sTSE_STATUS_PORT

Field Name	Field Type	Field Description
`spOutOfChar`	`UINT1`	Rx out of character alignment
`spOutOfFrame`	`UINT1`	Rx out of frame alignment

Port Configuration Block: PORT CONFIG

The `sTSE_CFG_PORT` is the top-level structure for configuring ports.

Table 15: TSE Port Configuration Block: sTSE_CFG_PORT

Field Name	Field Type	Field Description
`rxInvert`	`UINT1`	Rx, data invert
`forceAis`	`UINT1`	Rx, Force Ais (if device is in out of frame state)
`forceOutOfChar`	`UINT1`	Rx, Force out of character alignment
`forceOutOfFrame`	`UINT1`	Rx, Force out of frame alignment
`analogReset`	`UINT1`	Rx, Analog reset
`druEnable`	`UINT1`	Rx, DRU enable
`rxEnable`	`UINT1`	Receiver enable
`txEnable`	`UINT1`	Transmitter enable
`testPatEnb`	`UINT1`	Tx, Test pattern enable
`testPattern`	`UINT2`	Tx, Test pattern
`centerFifo`	`UINT1`	Tx, Center FIFO
`j0Insert`	`UINT1`	Tx, J0 insertion enable
`forceLineCodeV`	`UINT1`	Tx, force line code violation error
`pisoEnable`	`UINT1`	Tx, PISO enable

Device Configuration Block: DEVICE CONFIG

The `sTSE_CFG_DEVICE` is the top-level structure for configuring the device.

Table 16: TSE Device Configuration Block: sTSE_CFG_DEVICE

Field Name	Field Type	Field Description
`csuEnable[TSE_MAX_CSU_BLOCKS]`	`UINT1`	CSU enable
`csuReset[TSE_MAX_CSU_BLOCKS]`	`UINT1`	CSU reset
`etseDisable`	`UINT1`	Egress disable
`itseDisable`	`UINT1`	Ingress disable
`rxJ0Delay`	`UINT2`	Receive J0 delay
`txJ0Delay`	`UINT2`	Transmit J0 delay

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 `sysTSEBufferStart` function. An individual ISV buffer is then obtained by the driver via `sysTSEISVBufferGet` and returned to the 'pool' via `sysTSEISVBufferRtn`.

Table 17: TSE Interrupt Service Vector: sTSE_ISV

Field Name	Field Type	Field Description
`deviceHandle`	`sTSE_HNDL`	Handle to the device in cause
`whichints`	`sTSE_MASK`	Mask of interrupts that have occurred

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, which is typically done by its callback routines before they return.

Table 18: TSE Deferred Processing Vector: sTSE_DPV

Field Name	Field Type	Field Description
`dpEvent`	`eTSE_DPR_EVENT`	Event being reported
`dpCause`	`UINT2`	Depends on the event. Either a port number, CSU number, ITSE number or an ETSE number

4.3 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.

- `tseMdb`: 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 `TSE_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 dSignature field is correct. 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 `TSE_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 TSE 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 cannot invoke the same API at the same time. However the driver protects its data structures from concurrent accesses by the application and the 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 will take care of initializing the driver, allocating memory and all the other 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: tseModuleOpen

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 tseModuleOpen(sTSE_MIV *pmiv)`
Inputs	`pmiv` : (pointer to) Module Initialization Vector
Outputs	Places the address of the MDB into the MIV passed by the application.

Returns	Success = `TSE_SUCCESS`
	Failure = `TSE_ERR_MODULE_ALREADY_INIT`
	`TSE_ERR_INVALID_MIV`
	`TSE_ERR_MEM_ALLOC`

Valid States	`TSE_MOD_START`
Side Effects	Changes the MODULE state to `TSE_MOD_IDLE`

Closing the Driver Module: tseModuleClose

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

Prototype	`INT4 tseModuleClose(void)`
Inputs	None
Outputs	None

Returns	Success = `TSE_SUCCESS`
	Failure = `TSE_ERR_MODULE_NOT_INIT`
	`TSE_FAILURE`

Valid States	ALL STATES
Side Effects	Changes the MODULE state to `TSE_MOD_START`

Starting the Driver Module: tseModuleStart

This function connects the RTOS resources to the driver. This involves allocating semaphores and timers, initializing buffers and installing the ISR handler and DPR task. Upon successful return from this function the driver is ready to add devices.

Prototype	`INT4 tseModuleStart(void)`
Inputs	None
Outputs	None

Returns	Success = `TSE_SUCCESS`
	Failure = `TSE_ERR_MODULE_NOT_INIT`
	`TSE_ERR_WRONG_STATE`
	`TSE_ERR_SEMAPHORE`
	`TSE_ERR_STAT_INSTALL`

Valid States	`TSE_MOD_IDLE`
Side Effects	Changes the MODULE state to `TSE_MOD_READY`

Stopping the Driver Module: tseModuleStop

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

Prototype	`INT4 tseModuleStop(void)`
Inputs	None
Outputs	None

Returns	Success = `TSE_SUCCESS`
	Failure = `TSE_ERR_MODULE_NOT_INIT`
	`TSE_ERR_WRONG_STATE`
	`TSE_ERR_DEVS_EMPTY`
	`TSE_ERR_SEMAPHORE`

Valid States	`TSE_MOD_READY`
Side Effects	Changes the MODULE state to `TSE_MOD_IDLE`

5.2 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: tseAdd

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

```
sTSE_HNDL tseAdd(void *usrCtxt, UINT2 *baseAddr,
INT4 **pperrDevice)
```

Inputs

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

Outputs

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

ERROR code written to the tseMdb->errModule on failure

TSE_ERR_INVALID_ARG
TSE_ERR_NULL_BASE_ADDR
TSE_ERR_MODULE_NOT_INIT
TSE_ERR_DEVS_FULL
TSE_ERR_SEMAPHORE
TSE_ERR_DEV_ALREADY_ADDED
TSE_ERR_DEV_NOT_DETECTED

Returns Pointer to device handle or NULL in case of failure.

Valid States `TSE_MOD_READY`

Side Effects Changes the DEVICE state to `TSE_PRESENT`

Deleting a Device: tseDelete

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

Prototype `INT4 tseDelete(sTSE_HNDL deviceHandle)`

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

Outputs None

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_DIV`
`TSE_ERR_INVALID_ARG`
`TSE_ERR_SEMAPHORE`

Valid States `TSE_PRESENT, TSE_ACTIVE, TSE_INACTIVE`

Side Effects Changes the DEVICE state to `TSE_PRESENT`

Initializing a Device: tseInit

This function initializes the Device Data Block (DDB) associated with that device during `tseAdd`, applies a soft reset to the device and configures it according to the DIV passed by the Application. This routine may take up to 100ms to execute because it waits for CSUs to lock.

If the `pdiv` parameter is NULL, then h/w defaults are used instead for the DIV.

Prototype `INT4 tseInit(sTSE_HNDL deviceHandle, sTSE_DIV *pdiv, UINT1 profileNum))`

Inputs

`deviceHandle`	: device handle (from tseAdd)
`pdiv`	: pointer to Device Initialization Vector
`profileNum`	: the profile number is ignored by this driver

Outputs None

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_DIV`
`TSE_ERR_INVALID_ARG`

Valid States	`TSE_PRESENT`
Side Effects	Changes the DEVICE state to `TSE_INACTIVE`

Updating the Configuration of a Device: tseUpdate

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 `tseUpdate` and `tseInit` is that no soft reset will be applied to the device.

Prototype

```
INT4 tseUpdate(sTSE_HNDL deviceHandle, sTSE_DIV *pdiv,
UINT1 profileNum)
```

Inputs

`deviceHandle`	: device handle (from `tseAdd`)
`pdiv`	: pointer to Device Initialization Vector
`profileNum`	: the profile number is ignored by this driver

Outputs None

Returns

Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_DIV`
`TSE_ERR_INVALID_ARG`

Valid States `TSE_ACTIVE, TSE_INACTIVE`

Side Effects None

Resetting a Device: tseReset

This function applies a software reset to the TSE device. Also resets all the DDB contents (except for the user context). This function is typically called before re-initializing the device (via `tseInit`).

Prototype

```
INT4 tseReset(sTSE_HNDL deviceHandle)
```

Inputs

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

Outputs None

Returns

Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`

Valid States	`TSE_PRESENT, TSE_ACTIVE, TSE_INACTIVE`
Side Effects	Changes the DEVICE state to `TSE_PRESENT`

Activating a Device: tseActivate

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

Prototype	`INT4 tseActivate(sTSE_HNDL deviceHandle)`
Inputs	`deviceHandle` : device handle (from `tseAdd`)
Outputs	None

Returns	Success = `TSE_SUCCESS`
	Failure = `TSE_ERR_MODULE_NOT_INIT`
	`TSE_ERR_INVALID_DEV`
	`TSE_ERR_INVALID_STATE`

Valid States	`TSE_INACTIVE`
Side Effects	Changes the DEVICE state to `TSE_ACTIVE`

De-Activating a Device: tseDeActivate

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 tseDeActivate(sTSE_HNDL deviceHandle)`
Inputs	`deviceHandle` : device handle (from `tseAdd`)
Outputs	None

Returns	Success = `TSE_SUCCESS`
	Failure = `TSE_ERR_MODULE_NOT_INIT`
	`TSE_ERR_INVALID_DEV`
	`TSE_ERR_INVALID_STATE`

Valid States	`TSE_ACTIVE`
Side Effects	Changes the DEVICE state to `TSE_INACTIVE`

5.3 Device Read and Write

Reading from Device Registers: tseRead

This function is used to read a register of a specific TSE 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, `sysTSERead`. Note that a failure to read returns a zero and any error indication is written to the associated DDB.

Prototype	`UINT2 tseRead(sTSE_HNDL deviceHandle, UINT2 regNum)`
Inputs	`deviceHandle` : device handle (from `tseAdd`) `regNum` : register number

Outputs	The DDB field .errDevice contains
	
	Success = `TSE_SUCCESS`
	Failure = `TSE_ERR_INVALID_DEV`
	`TSE_ERR_INVALID_REG`

Returns	Value read
Valid States	`TSE_PRESENT, TSE_ACTIVE, TSE_INACTIVE`
Side Effects	May affect registers that change after a read operation

Writing to Device Registers: tseWrite

This function is used to write to a register of a specific TSE 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, `sysTSEWrite`.

Prototype	`UINT2 tseWrite(sTSE_HNDL deviceHandle, UINT2 regNum, UINT2 value)`
Inputs	`deviceHandle` : device handle (from `tseAdd`) `regNum` : register number `value` : value to be written

Outputs	The DDB field .errDevice contains
	
	Success = `TSE_SUCCESS`
	Failure = `TSE_ERR_INVALID_DEV`
	`TSE_ERR_INVALID_REG`

Returns	Value written
Valid States	`TSE_PRESENT, TSE_ACTIVE, TSE_INACTIVE`
Side Effects	May change the configuration of the device

Reading from a block of Device Registers: tseReadBlock

This function is used to read a register block of a specific TSE 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, `sysTSERead`. It is the user's responsibility to allocate enough memory for the block read.

Prototype

```
UINT2 tseReadBlock(sTSE_HNDL deviceHandle, UINT2
startRegNum, UINT2 numu2, UINT2 *pblock)
```

Inputs

`deviceHandle`	: device handle (from `tseAdd`)
`startRegNum`	: starting register number
`numu2`	: number of registers to read
`pblock`	: (pointer to) the block to read

Outputs

`pblock`	: (pointer to) the block read

The DDB field .errDevice contains:

Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`
`TSE_ERR_INVALID_REG`

Returns	none
Valid States	`TSE_PRESENT, TSE_ACTIVE, TSE_INACTIVE`
Side Effects	May affect registers that change after a read operation

Writing to a Block of Device Registers: tseWriteBlock

This function is used to write to a register block of a specific TSE 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, `sysTSEWrite`. A bit from the passed block is only modified in the device's registers if the corresponding bit is set in the passed mask.

Prototype

```
UINT2 tseWriteBlock(sTSE_HNDL deviceHandle, UINT2
startRegNum, UINT2 numu2, UINT2 *pblock, UINT2 *pmask)
```

Inputs

`deviceHandle`	: device handle (from `tseAdd`)
`startRegNum`	: starting register number
`numu2`	: number of registers to write
`pblock`	: (pointer to) block to write
`pmask`	: (pointer to) mask

Outputs The DDB field `.errDevice` contains

Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`
`TSE_ERR_INVALID_REG`

Returns None

Valid States `TSE_PRESENT, TSE_ACTIVE, TSE_INACTIVE`

Side Effects May change the configuration of the device

Indirect reading from a Device Register: tseReadIndirect

This function is used to perform an indirect read from an indirect register in the TSE device by providing the register location and the indirect address to be read from. This function derives the actual start address location based on the device handle. It then reads the data pointed to by the indirect address using calls to the system-specific macro, `sysTseRead`. Note that a failure to read returns a zero and any error indication is written to the DDB.

Prototype

```
INT4 tseReadIndirect(sTSE_HNDL deviceHandle, UINT2
iaddrReg, UINT2 iaddr, UINT2 *pData)
```

Inputs

`deviceHandle`	: device handle (from `tseAdd`)
`iaddrReg`	: indirect address register number
`iaddr`	: indirect address to read

	`pData`	: (pointer to) the data to read
Outputs	`pData`	: (pointer to) the block read

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_ARG`
`TSE_ERR_POLL_TIMEOUT`

Valid States `TSE_PRESENT, TSE_ACTIVE, TSE_INACTIVE`

Side Effects May affect registers that change after a read operation

Indirect writing to a Device Registers: tseWriteIndirect

This function is used to perform an indirect write to an indirect access register in the TSE device by providing the register location and the indirect address to be written to. This function derives the actual start address location based on the device handle. It then writes the data to the location pointed to by the indirect address using calls to the system-specific macro, `sysTseWrite`. Note that a failure to write returns a zero and any error indication is written to the DDB.

Prototype `INT4 tseWriteIndirect(sTSE_HNDL deviceHandle, UINT2 iaddrReg, UINT2 iaddr, UINT2 data)`

Inputs	`deviceHandle`	: device handle (from `tseAdd`)
	`iaddrReg`	: indirect address register number
	`iaddr`	: indirect address to read
	`data`	: new data

Outputs None

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_ARG`
`TSE_ERR_POLL_TIMEOUT`

Valid States `TSE_PRESENT, TSE_ACTIVE, TSE_INACTIVE`

Side Effects May affect registers that change after a read operation

5.4 Time Slot Interchange and Space Switch

Each TSI has two connection memory pages, 0 and 1. Mapping is defined at the STS-1 granularity; however, a valid mapping must still fit into the required time slot map mandated by the data rate of the channel. The higher lever application is responsible for maintaining data integrity when redefining the connection map.

Time slot interchange is seen as a process of mapping source space timeslots to destination space timeslots. This is used to establish a one-to-one mapping or one-to-many mapping between the source slots and the destination slots, depending on whether it is a unicast or a multicast connection.

The following functions allow for the configuration of the Time Slot Interchange and Space Switch.

Setting global mapping mode: tseSetMapMode

This function is used to set global mapping mode of all the TSIs in the device. There are two valid modes: user-defined and bypass. Bypass mode puts the chip in a through mode and timeslot rearrangement will not occur. If user-defined mode is selected a valid time slot map will be required.

Prototype `INT4 tseSetMapMode (sTSE_HNDL deviceHandle, eTSE_TSIMODE mode)`

Inputs

`deviceHandle`	: device handle (from `tseAdd`)
`mode`	: TSI mode

Outputs None

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`
`TSE_ERR_POLL_TIMEOUT`

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

Getting global mapping mode: tseGetMapMode

This function is used to get global mapping mode of all the TSIs in the device. There are two valid modes: user-defined and bypass. Bypass mode puts the chip in a through mode and timeslot rearrangement will not occur. If user-defined mode is selected a valid time slot map will be required.

Prototype	`INT4 tseGetMapMode (sTSE_HNDL deviceHandle, eTSE_TSIMODE *pmode)`	
Inputs	`deviceHandle`	: device handle (from `tseAdd`)
Outputs	`pmode`	: pointer to the TSI mode

Returns	Success = `TSE_SUCCESS`
	Failure = `TSE_ERR_MODULE_NOT_INIT`
	`TSE_ERR_INVALID_DEV`
	`TSE_ERR_INVALID_STATE`	

Valid States	`TSE_INACTIVE or TSE_ACTIVE`	
Side Effects	None	

Setting active connection page: tseSetPage

This function sets the specified page number as the active connection memory pages for the TSIs.

A total of thirty-three pages numbers will be changed; one Space Switch (SSWT), sixteen Ingress Time Switch Elements (ITSE) and sixteen Egress Time Switch Elements (ETSE).

Prototype	`INT4 tseSetPage (sTSE_HNDL deviceHandle, UINT2 pgNum)`	
Inputs	`deviceHandle` `pgNum`	: device handle (from `tseAdd`) : active page number (0 or 1)
Outputs	None	

Returns	Success = `TSE_SUCCESS`
	Failure = `TSE_ERR_MODULE_NOT_INIT`
	`TSE_ERR_INVALID_DEV`
	`TSE_ERR_INVALID_STATE`
	`TSE_ERR_INVALID_ARG`	

Valid States	`TSE_INACTIVE or TSE_ACTIVE`	
Side Effects	None	

Getting active connection page: tseGetPage

This function gets the active connection memory page that is stored in the driver's database. See the routine tseGetOnePage to get an individual TSI memory page number directly from the device.

Prototype `INT4 tseGetPage (sTSE_HNDL deviceHandle, UINT2 *ppgNum)`

Inputs

`deviceHandle`	: device Handle (from `tseAdd`)
`ppgNum`	: location to store page number

Outputs

`ppgNum`	: pointer to the active page number

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
TSE_ERR_INVALID_ARG

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

Setting active connection page: tseSetOnePage

This function sets an individual TSI connection memory page.

One of the following TSIs may be changed: Space Switch (SSWT), one of the sixteen Ingress Time Switch Elements (ITSE) or one of the sixteen Egress Time Switch Elements (ETSE).

Prototype

```
INT4 tseSetOnePage (sTSE_HNDL deviceHandle, eTSE_BLOCK
blockType, UINT1 blockNum, UINT2 pgNum)
```

Inputs

`deviceHandle`	: device Handle (from `tseAdd`)
`blockType`	: the block to read page number from (SSWT, ITSE, ETSE)
`blockNum`	: block within the TSI to read page number from (ignored for SSWT, otherwise it must be 1 to TSE_MAX_IE_BLOCKS)
`pgNum`	: active page number (0 or 1)

Outputs None

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

Getting active connection page: tseGetOnePage

This function gets an individual TSI connection memory from the device.

Prototype

```
INT4 tseGetOnePage (sTSE_HNDL deviceHandle, eTSE_BLOCK
blockType, UINT1 blockNum, UINT2 *ppgNum)
```

Inputs

`deviceHandle`	: device Handle (from `tseAdd`)
`blockType`	: the block to read page number from (SSWT, ITSE, ETSE)
`blockNum`	: block within the TSI to read page number from (ignored for SSWT, otherwise it must be 1 to TSE_MAX_IE_BLOCKS)
`ppgNum`	: location to store page number

Outputs

`ppgNum`	: the active page number (0 or 1)

Returns

Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

Copying connection map from one page to another in: tseCopyPage

This function is used to synchronize (overwrite) the given page using the other connection page.

Prototype `INT4 tseCopyPage (sTSE_HNDL deviceHandle, UINT2 src, UINT2 dest)`

Inputs

`deviceHandle`	: device handle (from `tseAdd`)
`src`	: source page number
`dest`	: destination page number

Outputs None

Returns Success = `TSE_SUCCESS`

Failure =
```
TSE_ERR_MODULE_NOT_INIT
TSE_ERR_INVALID_DEV
TSE_ERR_INVALID_STATE
TSE_ERR_INVALID_ARG
TSE_ERR_INVALID_MODE
TSE_ERR_POLL_TIMEOUT
```

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

Mapping the source to destination slots(s): tseMapSlot

This function is used to map the source space-time slot to the destination space-time slots. If the number of destinations is less than or equal to one the connection is unicast; otherwise it is multicast.

Prototype `INT4 tseMapSlot (sTSE_HNDL deviceHandle, UINT2 pgNum, sTSE_SPTSLOT *psrcSlot, sTSE_SPTSLOT destSlot[], UINT4 numSlots)`

Inputs

`deviceHandle`	: device handle (from `tseAdd`)
`pgNum`	: connection page
`psrcSlot`	: pointer to source space-time slot
`destSlot`	: array of destination space-time slots
`numSlots`	: number of destination space-time slots

Outputs None

Returns Success = `TSE_SUCCESS`

Failure =
```
TSE_ERR_MODULE_NOT_INIT
TSE_ERR_INVALID_DEV
TSE_ERR_INVALID_STATE
TSE_ERR_INVALID_ARG
TSE_ERR_INVALID_MODE
```

```
TSE_ERR_CONNECT_EXIST
TSE_ERR_POLL_TIMEOUT
```

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

Removing established connection: tseRmSlot

This function is used to remove a connection between the source space-time slot to the destination space-time slots.

Note: the connection is removed from the driver's database only. It is not possible to remove a connection from the device because it does not support the concept of "no-connection", an output is always connected to an input.

Prototype

```
INT4 tseRmSlot (sTSE_HNDL deviceHandle, UINT2 pgNum,
sTSE_SPTSLOT *psrcSlot, sTSE_SPTSLOT destSlot[], UINT4
numSlots)
```

Inputs

`deviceHandle`	: device handle (from `tseAdd`)
`pgNum`	: connection page
`psrcSlot`	: pointer to source space-time slot
`destSlot`	: array of destination space-time slots
`numSlots`	: number of destination space-timeslots

Outputs None

Returns

Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`

```
TSE_ERR_INVALID_DEV
TSE_ERR_INVALID_STATE
TSE_ERR_INVALID_ARG
TSE_ERR_INVALID_MODE
TSE_ERR_CONNECT_EXIST
```

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

Clearing all connections: tseClrSlot

This function removes all given mapping for the source space-time.

Note: the connection is removed from the driver's database only. It is not possible to remove a connection from the device because it does not support the concept of "no-connection", an output is always connected to an input.

Prototype `INT4 tseClrSlot (sTSE_HNDL deviceHandle, UINT2 pgNum, sTSE_SPTSLOT *psrcSlot)`

Inputs

`deviceHandle`	: device handle (from `tseAdd`)
`pgNum`	: connection page
`psrcSlot`	: pointer to source space-time slot

Outputs None

Returns

Success = `TSE_SUCCESS`

Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`
`TSE_ERR_INVALID_MODE`
`TSE_ERR_CONNECT_EXIST`

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

Getting source space-time Slot: tseGetSrcSlot

This function is used to get the space-time slot mappings given a specific destination space-time slot. If the number of destinations is less than or equal to one the connection is unicast; otherwise it is multicast.

Prototype `INT4 tseGetSrcSlot (sTSE_HNDL deviceHandle, UINT2 pgNum, sTSE_SPTSLOT *psrcSlot, sTSE_SPTSLOT *pdestSlot)`

Inputs	`deviceHandle`	: device handle (from `tseAdd`)
	`pgNum`	: connection page
	`pdestSlot`	: pointer to destination space-time slots
Outputs	`psrcSlot`	: pointer to source space-time slot

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
TSE_ERR_INVALID_ARG

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

Getting destination space-time Slot: tseGetDestSlot

This function is used to get the space-time slot mappings given a specific source space-time slot. If the number of destinations is less than or equal to one, the connection is unicast; otherwise, it is multicast.

Prototype `INT4 tseGetDestSlot (sTSE_HNDL deviceHandle, UINT2 pgNum, sTSE_SPTSLOT *psrcSlot, sTSE_SPTSLOT destSlot[], UINT4 *pNumSlots)`

Inputs	`deviceHandle`	: device handle (from `tseAdd`)
	`pgNum`	: connection page
	`psrcSlot`	: pointer to source space-time slot
Outputs	`destSlot`	: array of pointers to destination space-time slots
	`pNumSlots`	: pointer to number of destination space- time slots

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

Verifying a multicast connection: tseIsMulticast

This function is given a source space-time slot and discovers if this connection is a multicast.

Prototype

```
INT4 tseIsMulticast (sTSE_HNDL deviceHandle, UINT2
pgNum, sTSE_SPTSLOT *psrcSlot)
```

Inputs

`deviceHandle`	: device handle (from `tseAdd`)
`pgNum`	: connection page
`psrcSlot`	: pointer to source space-time slot

Outputs None

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

Inserting Idle Data: tseInsIdleData

This function is given a destination space-time slot and inserts idle data into that time slot.

Prototype

```
INT4 tseInsIdleData (sTSE_HNDL deviceHandle, UINT2 pgNum,
UINT2 blk, sTSE_SPTSLOT *pSlot, BOOLEAN insert, UINT2
idleData)
```

Inputs

`deviceHandle`	: device handle (from `tseAdd`)
`pgNum`	: connection page
`blk`	: 0 = ITSE, 1 = ETSE
`pSlot`	: pointer to source space-time slot
`insert`	: enable/disable insertion
`idleData`	: idle data to be inserted

Outputs None

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`

TSE_ERR_INVALID_STATE
TSE_ERR_INVALID_ARG
TSE_ERR_POLL_TIMEOUT

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

5.5 Port Alarm, Status and Statistics

The TSE device driver has the capability to collect and report both port and device level status and statistics. The functions described in this section are for the collection and reporting of port level status and statistics. For the device level information, see section 5.6.

Getting port cumulative statistics: tsePortGetStats

This function retrieves all the port statistical counts that are kept in the Device Data Block (DDB).

Prototype `INT4 tsePortGetStats (sTSE_HNDL deviceHandle, UINT2 port, sTSE_CNTR_PORT *pCnts)`

Inputs

deviceHandle	: device handle (from tseAdd)
port	: port number

Outputs

pCnts	: pointer to the statistics

Returns Success = TSE_SUCCESS
Failure = TSE_ERR_MODULE_NOT_INIT
TSE_ERR_INVALID_DEV
TSE_ERR_INVALID_STATE
TSE_ERR_INVALID_ARG
TSE_ERR_SEMAPHORE

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

Getting port status: tsePortGetStatus

This function retrieves all the port status information directly from the device.

Prototype `INT4 tsePortGetStatus (sTSE_HNDL deviceHandle, UINT2 port, sTSE_STATUS_PORT *pStatus)`

Inputs

deviceHandle	: device handle (from tseAdd)
port	: port number

Outputs	`pStatus`	: pointer to the status

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

Getting port delta statistics: tsePortGetDelta

This function retrieves all the port delta statistical counts that are kept in the Device Data Block (DDB).

Prototype

```
INT4 tsePortGetDelta (sTSE_HNDL deviceHandle, UINT2 port,
sTSE_CNTR_PORT *pCnts)
```

Inputs	`deviceHandle`	: device handle (from `tseAdd`)
	`port`	: port number
Outputs	`pCnts`	: pointer to the statistics

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`
`TSE_ERR_SEMAPHORE`

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

Getting port interrupt callback threshold: tsePortGetThresh

This function retrieves all the port thresholds that are kept in the Device Data Block (DDB).

Prototype

```
INT4 tsePortGetThresh(sTSE_HNDL deviceHandle, UINT2 port,
sTSE_CNTR_PORT *pCnts)
```

Inputs	`deviceHandle`	: device handle (from `tseAdd`)
	`port`	: port number
Outputs	`pCnts`	: pointer to the thresholds

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

Setting port interrupt callback threshold: tsePortSetThresh

This function configures all the port thresholds that are kept in the Device Data Block (DDB). The threshold controls how often the driver calls back to the application for an event. A threshold of 2 means the driver will call back once for every 2 such events.

Prototype `INT4 tsePortSetThresh(sTSE_HNDL deviceHandle, UINT2 port, sTSE_CNTR_PORT *pCnts)`

Inputs

`deviceHandle`	: device handle (from `tseAdd`)
`port`	: port number
`pCnts`	: pointer to the thresholds

Outputs None

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

Clear all port statistics: tsePortClrStats

This function clears the cumulative and delta counts for a port that are kept in the Device Data Block (DDB).

Prototype `INT4 tsePortClrStats(sTSE_HNDL deviceHandle, UINT2 port)`

Inputs

`deviceHandle`	: device handle (from `tseAdd`)
`port`	: port number

Outputs None

Returns Success = `TSE_SUCCESS`

Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`
`TSE_ERR_SEMAPHORE`

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

5.6 Device Alarm, Status and Statistics

The TSE device driver has the capability to collect and report both port and device level status and statistics. The functions described in this section are for the collection and reporting of device level status and statistics: For the port level information, see section 5.5.

Getting device cumulative statistics: tseDeviceGetStats

This function retrieves all the device statistical counts that are kept in the Device Data Block (DDB).

Prototype

```
INT4 tseDeviceGetStats (sTSE_HNDL deviceHandle,
sTSE_CNTR_DEVICE *pCnts)
```

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

Outputs `pCnts` : pointer to the statistics

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`
`TSE_ERR_SEMAPHORE`

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

Getting device status: tseDeviceGetStatus

This function retrieves all the device status information directly from the device.

Prototype	`INT4 tseDeviceGetStatus (sTSE_HNDL deviceHandle, sTSE_STATUS_DEVICE *pStatus)`	
Inputs	`deviceHandle`	: device handle (from `tseAdd`)
Outputs	`pStatus`	: pointer to the status

Returns	Success = `TSE_SUCCESS`
	Failure = `TSE_ERR_MODULE_NOT_INIT`
	`TSE_ERR_INVALID_DEV`
	`TSE_ERR_INVALID_STATE`
	`TSE_ERR_INVALID_ARG`	

Valid States	`TSE_INACTIVE or TSE_ACTIVE`	
Side Effects	None	

Getting device delta statistics: tseDeviceGetDelta

This function retrieves all the device delta statistical counts that are kept in the Device Data Block (DDB).

Prototype	`INT4 tseDeviceGetDelta (sTSE_HNDL deviceHandle, sTSE_CNTR_DEVICE *pCnts)`	
Inputs	`deviceHandle`	: device handle (from `tseAdd`)
Outputs	`pCnts`	: pointer to the statistics

Returns	Success = `TSE_SUCCESS`
	Failure = `TSE_ERR_MODULE_NOT_INIT`
	`TSE_ERR_INVALID_DEV`
	`TSE_ERR_INVALID_STATE`
	`TSE_ERR_INVALID_ARG`
	`TSE_ERR_SEMAPHORE`	

Valid States	`TSE_INACTIVE or TSE_ACTIVE`	
Side Effects	None	

Getting device interrupt callback threshold: tseDeviceGetThresh

This function retrieves all the device thresholds that are kept in the Device Data Block (DDB).

Prototype	`INT4 tseDeviceGetThresh(sTSE_HNDL deviceHandle, sTSE_CNTR_DEVICE *pCnts)`	
Inputs	`deviceHandle`	: device handle (from `tseAdd`)
Outputs	`pCnts`	: pointer to the thresholds

Returns	Success = `TSE_SUCCESS`
	Failure = `TSE_ERR_MODULE_NOT_INIT`
	`TSE_ERR_INVALID_DEV`
	`TSE_ERR_INVALID_STATE`
	`TSE_ERR_INVALID_ARG`	

Valid States	`TSE_INACTIVE or TSE_ACTIVE`	
Side Effects	None	

Setting device interrupt callback threshold: tseDeviceSetThresh

This function configures all the device thresholds that are kept in the Device Data Block (DDB). The threshold controls how often the driver calls back to the application for an event. A threshold of 2 means the driver will call back once for every 2 such events.

Prototype	`INT4 tseDeviceSetThresh(sTSE_HNDL deviceHandle, sTSE_CNTR_DEVICE *pCnts)`	
Inputs	`deviceHandle` `pCnts`	: device handle (from `tseAdd`) : pointer to the thresholds
Outputs	None	

Returns	Success = `TSE_SUCCESS`
	Failure = `TSE_ERR_MODULE_NOT_INIT`
	`TSE_ERR_INVALID_DEV`
	`TSE_ERR_INVALID_STATE`
	`TSE_ERR_INVALID_ARG`	

Valid States	`TSE_INACTIVE or TSE_ACTIVE`	
Side Effects	None	

Clear all device statistics: tseDeviceClrStats

This function clears all the device statistics that are kept in the Device Data Block (DDB).

Prototype	`INT4 tseDeviceClrStats(sTSE_HNDL deviceHandle)`	
Inputs	`deviceHandle`	: device handle (from `tseAdd`)
Outputs	None	

Returns	Success = `TSE_SUCCESS`
	Failure = `TSE_ERR_MODULE_NOT_INIT`
	`TSE_ERR_INVALID_DEV`
	`TSE_ERR_INVALID_STATE`
	`TSE_ERR_INVALID_ARG`
	`TSE_ERR_SEMAPHORE`	

Valid States	`TSE_INACTIVE or TSE_ACTIVE`	
Side Effects	None	

5.7 Device Configuration

The TSE device has several device level modes. The following APIs allow these modes to be configured.

Setting device configuration: tseDeviceSetConfig

This function allows the configuration of a TSE device to be dynamically changed.

Prototype	`INT4 tseDeviceSetConfig (sTSE_HNDL , sTSE_CFG_DEVICE *pDevConfig)`	
Inputs	`deviceHandle`	: device handle (from `tseAdd`)
	`pDevConfig`	: pointer to the device configuration
Outputs	None	

Returns	Success = `TSE_SUCCESS`
	Failure = `TSE_ERR_MODULE_NOT_INIT`
	`TSE_ERR_INVALID_DEV`
	`TSE_ERR_INVALID_STATE`
	`TSE_ERR_INVALID_ARG`	

Valid States	`TSE_INACTIVE or TSE_ACTIVE`	
Side Effects	None	

Getting device configuration: tseDeviceGetConfig

This function allows the configuration of an 8b/10b device.

Prototype

```
INT4 tseDeviceGetConfig (sTSE_HNDL deviceHandle,
sTSE_CFG_DEVICE *pDevConfig)
```

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

Outputs `pDevConfig` : pointer to the device configuration

Returns

Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`

Valid States `TSE_ACTIVE or TSE_INACTIVE`

Side Effects None

5.8 Port Configuration

The TSE device has 64 ports in each direction (ingress and egress). The following APIs allow these ports to be configured.

Setting port configuration: tsePortSetConfig

This function allows the configuration of an 8b/10b port.

Prototype `INT4 tsePortSetConfig (sTSE_HNDL deviceHandle, UINT2 port, sTSE_CFG_PORT *pconfig)`

Inputs	`deviceHandle`	: device handle (from `tseAdd`)
	`port`	: port number
	`pconfig`	: pointer to the port configuration

Outputs None

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

Getting port configuration: tsePortGetConfig

This function allows the configuration of an 8b/10b port.

Prototype `INT4 tsePortGetConfig (sTSE_HNDL deviceHandle, UINT2 port, sTSE_CFG_PORT *pconfig)`

Inputs	`deviceHandle`	: device handle (from `tseAdd`)
	`port`	: port number
Outputs	`pconfig`	: pointer to the port configuration

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`

Valid States `TSE_ACTIVE or TSE_INACTIVE`

Side Effects None

Setting J0 Masking mode: tsePortSetMaskMode

This function defines one of three modes for filtering J0 characters from the time-space-time switch core.

`TSE_J0MASK_ALLOW` mode will allow good J0's into the switch core and is available on all revisions of the device. It is also the default mode of the device when it is brought out of reset.

`TSE_J0MASK_DENY` mode will not allow good J0 characters into the switch. This mode is only available on revision D, or later, TSE devices.

`TSE_J0MASK_DENY_REORDER` mode will not allow good J0 characters into the switch core and the J0/Z0 bytes are re-ordered in the ITSE/ETSE. This mode is only available on revision D, or later, TSE devices.

Prototype `INT4 tsePortSetMaskMode (sTSE_HNDL deviceHandle, UINT2 portSet, eTSE_MASK_MODE mode)`

Inputs

`deviceHandle`	: device Handle (from `tseAdd`)
`portSet`	: the first port number of a set of 4 ports (1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61)
`mode`	: masking mode (Allow, Deny, Deny_Reorder)

Outputs None

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`
`TSE_ERR_DEV_VERSION`

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

Notes Ports can only be put into these modes 4 at a time because the ITSE/ETSE applies masking configuration globally to the ports feeding them.

Depending on the mode, the following bits will be set or cleared:

`J0MASK`	in 4 R8FA ports
`IJ0RORDR`	in 1 ITSE block
`EJ0RORDR`	in 1 ETSE block
`J0INS`	in 4 T8DE ports

5.9 8b/10b Decoder/Encoder

The TSE device driver has the capability to force certain errors on the 8b/10b ports. The following functions are intended to give access to these features.

Forcing port OOC alignment: tseForceOutOfChar

This function allows the forcing of an out of character alignment on an 8b/10b port.

Prototype `INT4 tseForceOutOfChar (sTSE_HNDL deviceHandle, UINT2 port, UINT2 force)`

Inputs

`deviceHandle`	: device handle (from `tseAdd`)
`port`	: port number
`force`	: 1 = force Out Of Character with a 0->1 transistion of the R8FA_FOCA bit 0 = no not force, write a 0 to R8FA_FOCA bit

Outputs None

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

Forcing port OOF alignment: tseForceOutOfFrame

This function allows the forcing of an out of frame alignment on an 8b/10b port.

Prototype `INT4 tseForceOutOfFrame(sTSE_HNDL deviceHandle, UINT2 port, UINT2 force)`

Inputs

`deviceHandle`	: device handle (from `tseAdd`)
`port`	: port number
`force`	: 1 = force Out Of Frame with a 0->1 transistion of the R8FA_FOFA bit 0 = no not force, write a 0 to R8FA_FOFA bit

Outputs None

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

Forcing port AIS: tseForceAIS

This function allows the forcing of an out of frame Alarm Indication Signal.

As documented in the TSE Engineering document PMC-1990713, the AIS will only be inserted if the corresponding R8FA block is in the out-of-frame state.

Prototype `INT4 tseForceAIS(sTSE_HNDL deviceHandle, UINT2 port, UINT2 force)`

Inputs	`deviceHandle`	: device handle (from `tseAdd`)
	`port`	: port number
	`force`	: force AIS (1) or not (0)

Outputs None

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`

Valid States `TSE_INACTIVE or TSE_ACTIVE`

Side Effects None

Forcing port LCV: tseForceLcv

This function allows the forcing a line character violation on an 8b/10b port.

Prototype `INT4 tseForceLcv(sTSE_HNDL deviceHandle, UINT2 port, UINT2 force)`

Inputs	`deviceHandle`	: device handle (from `tseAdd`)
	`port`	: port number
	`force`	: force LCV (1) or not (0)

Outputs None

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`

Valid States `TSE_INACTIVE or TSE_ACTIVE`

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: tseISRConfig

This function allows the user to configure how ISR processing is to be handled: polling (`TSE_POLL_MODE`) or interrupt driven (`TSE_ISR_MODE`). If polling is selected, the user is responsible for periodically calling devicePoll to collect exception data from the Device.

Prototype `INT4 tseISRConfig(sTSE_HNDL deviceHandle, UINT2 mode)`

Inputs

`deviceHandle`	: device handle (from `tseAdd`)
`mode`	: mode of operation (`TSE_ISR_MODE` or `TSE_POLL_MODE`)

Outputs None

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`
`TSE_ERR_SEMAPHORE`

Valid States `TSE_PRESENT, TSE_ACTIVE, TSE_INACTIVE`

Side Effects None

Getting the Interrupt Status Mask: tseGetMask

This function returns the contents of the interrupt mask registers of the TSE device.

Prototype `INT4 tseGetMask(sTSE_HNDL deviceHandle, sTSE_MASK *pmask)`

Inputs

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

Outputs

`pmask`	: (pointer to) updated mask structure

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`

`TSE_ERR_INVALID_ARG`

Valid States `TSE_ACTIVE, TSE_INACTIVE`

Side Effects None

Setting the Interrupt Enable Mask: tseSetMask

This function sets the contents of the interrupt mask registers of the TSE device. Any bits that are set in the passed structure are set in the associated TSE registers. Any bits that are not set are left as is on the TSE device.

Prototype `INT4 tseSetMask(sTSE_HNDL deviceHandle, sTSE_MASK *pmask)`

Inputs

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

Outputs None

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`

Valid States `TSE_ACTIVE, TSE_INACTIVE`

Side Effects May change the operation of the ISR/DPR

Clearing the Interrupt Enable Mask: tseClearMask

This function clears individual interrupt bits and registers in the TSE device. Any bits that are set in the passed structure are cleared in the associated TSE registers. Any bits that are not set are left as is on the TSE device.

Prototype `INT4 tseClearMask(sTSE_HNDL deviceHandle, sTSE_MASK *pmask)`

Inputs

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

Outputs None

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`

Valid States `TSE_ACTIVE, TSE_INACTIVE`

Side Effects May change the operation of the ISR/DPR

Polling the Interrupt Status Registers: tsePoll

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

Prototype `INT4 tsePoll(sTSE_HNDL deviceHandle)`

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

Outputs None

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_INVALID_ARG`
`TSE_ERR_MODE`
`TSE_FAILURE`

Valid States `TSE_ACTIVE`

Side Effects None

Interrupt-Service Routine: tseISR

This function reads the state of the interrupt registers in the TSE 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 `sysTSEISRHandler`.

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. Note: When designing these functions, keep in mind all possible issues that may arise when multiple devices are present and some are in polling mode and some are in ISR mode.

Prototype	`void * tseISR(sTSE_HNDL deviceHandle)`	
Inputs	`deviceHandle`	: device handle (from `tseAdd`)
Outputs	None	
Returns	(pointer to) ISV buffer (to send to the DPR) or NULL (pointer)	
Valid States	`TSE_ACTIVE`	
Side Effects	None	

Deferred-Processing Routine: tseDPR

This function acts on data contained in the passed ISV, allocates one or more DPV buffers (via `sysTSEDPVBufferGet`) and invokes one or more callbacks (if defined and enabled). This routine is called by the application code, within `sysTSEDPRTask`. Note that the callbacks are responsible for releasing the passed DPV. It is recommended that this be done as soon as possible to avoid running out of DPV buffers. Note: When designing these functions, keep in mind all possible issues that may arise when multiple devices are present and some are in polling mode and some are in ISR mode.

Prototype	`void tseDPR(void *pivec)`	
Inputs	`pivec`	: (pointer to) ISV buffer
Outputs	None	
Returns	None	
Valid States	`TSE_ACTIVE`	

Side Effects None

5.11 Device Diagnostics

Testing Register Accesses: tseTestReg

This function verifies the hardware access to the device registers by writing and reading back values. The original register contents are restored after the test.

Prototype

```
INT4 tseTestReg(sTSE_HNDL deviceHandle, UINT2 *
pErrorRegNum, UINT2 * pErrorWrite, UINT2 * pErrorRead,
UINT2 * pErrorMask)
```

Inputs

`deviceHandle`	: device handle (from `tseAdd`)
`pErrorRegNum`	: pointer to register number test failed at
`pErrorWrite`	: pointer to value written
`pErrorRead`	: pointer to value read
`pErrorMask`	: pointer to mask applied to test value

Outputs None

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
`TSE_ERR_FAILRAMTEST`

Valid States `TSE_PRESENT, TSE_INACTIVE`

Side Effects None

Testing RAM Accesses: tseTestRAM

This function performs a RAM test at the read/write registers inside the device's memory space to verify the address and data bus connections between the CPU and the device.

Prototype

```
INT4 tseTestRAM(sTSE_HNDL deviceHandle, UINT2 iaddrReg,
UINT1 timeSlotStart, UINT1 timeSlotEnd, UINT1 bpStart,
UINT1 bpEnd)
```

Inputs

`deviceHandle`	: device handle (from `tseAdd`)
`iaddrReg`	: indirect address register
`timeSlotStart`	: time slot to start test (1-12)
`timeSlotEnd`	: time slot to end test (1-12)
`bpStart`	: block or port to start test
`bpEnd`	: block or port to end test

Outputs None

Returns Success = `TSE_SUCCESS`
Failure = `TSE_ERR_MODULE_NOT_INIT`
`TSE_ERR_INVALID_DEV`
`TSE_ERR_INVALID_STATE`
TSE_ERR_FAILRAMTEST

Valid States `TSE_PRESENT, TSE_INACTIVE`

Side Effects None

5.12 Callback Functions

The TSE driver has the capability to call back 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 `tseDPR` function are passed during the `tseInit` 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 `sysTSEDPVBufferRtn` either within the callback function or later inside the application code.

Once the number of events exceeds its specified threshold (in the DIV) the callback to the application is made. The thresholds can be changed dynamically by calling `tseDeviceSetThresh()` or `tsePortSetThresh()`. Setting a threshold count to 0 or 1 means that a callback will occur for every such event.

Calling Back to the Application due to device level events: cbackTSEDevice

This callback function is provided by the user and is used by the DPR to report significant device level 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 `tseInit` call. If the address of the callback function was passed as a NULL at initialization, no callback will be made.

Prototype	`void cbackTSEDevice(sTSE_USR_CTXT usrCtxt, void *pdpv)`	
Inputs	`usrCtxt`	: user context (from `tseAdd`)
	`pdpv`	: (pointer to) DPV that describes this event
Outputs	None	
Returns	None	
Valid States	`TSE_ACTIVE`	
Side Effects	None	

Calling Back to the Application due to port level events: cbackTSEPort

This callback function is provided by the user and is used by the DPR to report significant port level 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 `tseInit` call. If the address of the callback function was passed as a NULL at initialization, no callback will be made.

Prototype	`void cbackTSEPort(sTSE_USR_CTXT usrCtxt, void *pdpv)`
Inputs	`usrCtxt` : user context (from `tseAdd`) `pdpv` : (pointer to) DPV that describes this event
Outputs	None
Returns	None
Valid States	`TSE_ACTIVE`
Side Effects	None

6 HARDWARE INTERFACE

The TSE 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: sysTSERead

`sysTseRead` provides the most basic kind of hardware access; it reads the contents of a specific register location. This macro should be defined by the user according to the target system's addressing logic. There is no need for error recovery in this function.

Format	`#define sysTSERead(base, offset)`	
Prototype	`UINT2 sysTSERead(UINT2 *base, UINT2 offset)`	
Inputs	`base`	: base address of device
	`offset`	: offset of register from base address
Outputs	None	
Returns	value read from the addressed register location	

Writing to a Device Register: sysTSEWrite

`sysTseWrite` provides the most basic kind of hardware access; it writes the supplied value to the specific register location. This macro should be defined by the user according to the target system's addressing logic. There is no need for error recovery in this function.

Format	`#define sysTSEWrite(base, offset, value)`	
Prototype	`void sysTSEWrite(UINT2 *base, UINT2 offset, UINT2 value)`	
Inputs	`base`	: base address of device
	`offset`	: offset of register from base address
	`data`	: data to be written
Outputs	None	
Returns	None	

6.2 System-Specific Interrupt Servicing

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

During execution of the API function `tseModuleStart` / `tseModuleStop` the driver informs the application that it is time to install / uninstall this shell via `sysTSEISRHandlerInstall` / `sysTSEISRHandlerRemove`, which 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 (`tsePoll`) that in turn calls `tseISR`.

Installing the ISR Handler: sysTSEISRHandlerInstall

This routine installs the user-supplied Interrupt-Service Routine (ISR), `sysTSEISRHandler` into the processor's interrupt vector table. Also the Deferred Processing task (DPR), `sysTSEDPRTask` and the ISR to DPR message queue is installed.

Format	`#define sysTSEISRHandlerInstall(void)`
Prototype	`INT4 sysTSEISRHandlerInstall(void)`
Inputs	None
Outputs	None

Returns	Success = 0
	Failure = <any other value>

ISR Handler: sysTSEISRHandler

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

Format	`#define sysTSEISRHandler(irq)`	
Prototype	`void sysTSEISRHandler(INT4 irq)`	
Inputs	`irq`	: IRQ number for handler
Outputs	None	
Returns	None	

Removing the ISR Handler: sysTSEISRHandlerRemove

This routine disables interrupt processing for this device. Removes the user-supplied Interrupt Service routine (ISR), `sysTSEISRHandler`, from the processor's interrupt vector table. Also the Deferred Processing task (DPR), `sysTSEDPRTask` and the ISR to DPR message queue are removed.

Format	`#define sysTSEISRHandlerRemove ()`
Prototype	`void sysTSEISRHandlerRemove(void)`
Inputs	None
Outputs	None
Returns	None

7 RTOS INTERFACE

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

7.1 Memory Allocation/De-Allocation

Allocating Memory: sysTSEMemAlloc

This function allocates specified number of bytes of memory.

Format	`#define sysTSEMemAlloc(numBytes)`
Prototype	`UINT1 *sysTSEMemAlloc(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: sysTSEMemFree

This function frees memory allocated using `sysTSEMemAlloc`.

Format	`#define sysTSEMemFree(pfirstByte)`
Prototype	`void sysTSEMemFree(UINT1 *pfirstByte)`
Inputs	`pfirstByte` : pointer to first byte of the memory region being de-allocated
Outputs	None
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 `sysTSEBufferStart` call.

Starting Buffer Management: sysTSEBufferStart

This function alerts the RTOS that the time has come to make sure ISV buffers and DPV buffers are available and sized correctly. This may involve the creation of new buffer pools, or it may not involve anything, depending on the RTOS.

Format	`#define sysTSEBufferStart()`
Prototype	`INT4 sysTSEBufferStart(void)`
Inputs	None
Outputs	None

Returns	Success = 0
	Failure = <any other value>

Getting an ISV Buffer: sysTSEISVBufferGet

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

Format	`#define sysTSEISVBufferGet()`
Prototype	`sTSE_ISV *sysTSEISVBufferGet(void)`
Inputs	None
Outputs	None

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

Returning an ISV Buffer: sysTSEISVBufferRtn

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

Format	`#define sysTSEISVBufferRtn(pisv)`	
Prototype	`void sysTSEISVBufferRtn(sTSE_ISV *pisv)`	
Inputs	`pisv`	: (pointer to) a ISV buffer
Outputs	None	
Returns	None	

Getting a DPV Buffer: sysTSEDPVBufferGet

This function gets a buffer from the RTOS that will be used by the 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 sysTSEDPVBufferGet()`
Prototype	`sTSE_DPV *sysTSEDPVBufferGet(void)`
Inputs	None
Outputs	None

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

Returning a DPV Buffer: sysTSEDPVBufferRtn

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

Format	`#define sysTSEDPVBufferRtn(pdpv)`	
Prototype	`void sysTSEDPVBufferRtn(sTSE_DPV *pdpv)`	
Inputs	`pdpv`	: (pointer to) a DPV buffer
Outputs	None	
Returns	None	

Stopping Buffer Management: sysTSEBufferStop

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 sysTSEBufferStop()`
Prototype	`void sysTSEBufferStop(void)`
Inputs	None
Outputs	None
Returns	None

7.3 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' (`sysTSEDPRTask`) that in turn calls `tseDPR` with an ISV to perform the ISR related processing that is required by each interrupting device.

During execution of the API function `tseModuleStart` / `tseModuleStop`, the driver informs the application that it is time to install / uninstall this shell via `sysTSEISRHandlerInstall` / `sysTSEISRHandlerRemove`, which needs to be supplied by the user.

DPR Task: sysTSEDPRTask

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

Prototype	`void sysTSEDPRTask(void)`
Inputs	None
Outputs	None
Returns	None

8 PORTING THE TSE DRIVER

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

8.1 Driver Source Files

The C source files listed below contain the code for the TSE 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 `src` files contain the functions and the `inc` files contain the constants and macros.

Directory	File	Description
`src`	`tse_api1.c`	General driver API
`src`	`tse_apl2.c`	TSE specific API
`src`	`tse_hw.c`	Hardware interface routines
`src`	`tse_isr.c`	Interrupt service routines
`src`	`tse_rtos.c`	RTOS interface routines
`src`	`tse_util.c`	General utility routines
`inc`	`tse_api.h`	API prototypes
`inc`	`tse_defs.h`	TSE definitions
`inc`	`tse_err.h`	Error return values
`inc`	`tse_fns.h`	Prototypes for `tse_isr.c` and `tse_util.c`
`inc`	`tse_hw.h`	Prototypes for `tse_hw.c`
`inc`	`tse_rtos.h`	RTOS interface macros and prototypes
`inc`	`tse_strs.h`	TSE data structures
`inc`	`tse_typs.h`	TSE data types
`example`	`tse_debug.h`	Example debug definitions
`example`	`tse_app.c`	Example application callback routines

Directory	File	Description
`example`	`tse_debug.c`	Example debug task that prints API register accesses

8.2 Driver Porting Procedures

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

To port the TSE driver to your platform:

Step 1: Port the driver's OS extensions (page 92)

Step 2: Port the driver to your hardware platform (page 95)

Step 3: Port the driver's application-specific elements (page 96)

Step 4: Build the driver (page 97)

Step 1: Porting Driver OS Extensions

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

- Task management
- Message queues
- Events
- Memory Management

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

To port the driver's OS extensions:

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

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

Service Type	Macro Name	Description
Memory	`sysTSEMemAlloc`	Allocates the memory block
	`sysTSEMemFree`	Frees the memory block
	`sysTSEMemCpy`	Copies the memory block from src to dest
	`sysTSEMemSet`	Sets each character in the memory buffer
Semaphores	`sysTSESemCreate`	Create semaphore object
	`sysTSESemTake`	Take semaphore object
	`sysTSESemGive`	Give semaphore object
	`sysTSESemDelete`	Delete semaphore object
Miscellaneous	`sysTseAssert`	ANSI assert

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

Service Type	**Function Name**	**Description**
Memory	`sysTSEBufferStart`	Allocates buffers for ISR and DSP
	`sysTSEBufferStop`	Deallocates buffers for ISR and DSP
Interrupt	`sysTSEISRHandlerInstall`	Installs the interrupt handler for the OS
	`sysTSEISRHandlerRemove`	Removes the interrupt handler from the OS
	`sysTSEISRHandler`	Interrupt handler for the TSE device
	`sysTSEDPRTask`	Deferred interrupt-processing routine (DPR)

Step 2: Porting Drivers to Hardware Platforms

This section describes how to modify the TSE driver for your hardware platform.

This section describes how to modify the TSE driver for your hardware platform.

To port the driver to your hardware platform:

1. Modify the hardware specific macros in tse_hw.h:

Service Type	Function Name	Description
Device I/O	sysTseRead	Reads from a device register
	sysTseWrite	Writes to a device register

2. Modify the hardware specific functions in tse_hw.c:

Service Type	Function Name	Description
Interrupt	`sysTseISRHandlerInstall`	Installs the interrupt handler into the processor's interrupt vector table and spawns the DPR task
	`sysTseISRHandlerRemove`	Removes the interrupt handler from the RTOS and deletes the DPR task
	`sysTseISRHandler`	Interrupt handler for the TSE device
	`sysTseBufferSend`	Send ISV message to DPR task
	`sysTseDPRTask`	Task that calls the TSE DPR
Statistics collection	`sysTseStatTask`	Statistics polling task
	`sysTseStatTaskEnable`	Enable Statistics task count collection
	`sysTseStatTaskDisable`	Disable Statistics task count collection
Device I/O	`sysTseBusyBitPoll`	Polls a device register given its real address in memory

Step 3: Porting Driver Application Specific Elements

Application specific elements are configuration constants used by the API for developing an application. This section describes how to modify the application specific elements in the TSE driver.

To port the driver's application specific elements:

1. Edit TSE_ERR_BASE in tse_err.h so that the driver's error codes do not collide with any other error codes in your system.
2. Define the following driver task-related constants for your RTOS specific services in file tse_rtos.h:

Task Constant	Description	Default
`TSE_DPR_TASK_PRIORITY`	Deferred Task (DPR) task priority	85
`TSE_DPR_TASK_STACK_SZ`	DPR task stack size, in bytes	8192
`TSE_STAT_TASK_PRIORITY`	Statistics task priority	95
`TSE_STAT_TASK_STACK_SZ`	Statistics task stack size, in bytes	8192
`TSE_TASK_SHUTDOWN_DELAY`	Delay time in milliseconds. When clearing the DPR loop active flag in the DPR task, this delay is used to gracefully shutdown the DPR task before deleting it	100

3. Code the callback functions according to your application. There are two sample callback functions in the `tse_app.c fi`le. You can customize them before using the driver, then write their addresses into the `cbackTseDevice` and `cbackTsePort` members of the DIV passed to `tseInit()`. These functions must free the DPR before returning and should conform to the following prototypes:

- `void cbackTseDevice (sTSE_CTXT usrCtxt, sTSE_DPV *pdpv)`
- `void cbackTsePort (sTSE_CTXT usrCtxt, sTSE_DPV *pdpv)`

Note: The port callback routine in `tse_app.c` illustrates how you must disable the Rx Fifo interrupt when it first occurs. See `tse_app.c` for a full explanation.

Step 4: Building the Driver

This section describes how to build the TSE driver.

To build the driver:

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

2. Compile the source files and build the TSE driver using your make utility.

3. Link the TSE driver to your application code.

APPENDIX A: CODING CONVENTIONS

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

Variable Type Definitions

Table 19: 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 30 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 20: Naming Conventions

Type	Case	Naming convention	Examples
Macros	Uppercase	prefix with "`m`" and device abbreviation	`mTSE_WRITE`
Constants	Uppercase	prefix with device abbreviation	`TSE_REG`
Enumeration	Hungarian	prefix with "`e`" and device abbreviation	`eTSE_MOD_STATS`
Structures	Hungarian Notation	prefix with "`s`" and device abbreviation	`sTSE_DDB`
API Functions	Hungarian Notation	prefix with device name	`tseAdd()`
Porting Functions	Hungarian Notation	prefix with "`sys`" and device name	`sysTSEReadReg()`
Other Functions	Hungarian Notation		`myOwnFunction()`
Variables	Hungarian Notation		`maxDevs`
Pointers to variables	Hungarian Notation	prefix variable name with "`p`"	`pmaxDevs`
Global variables	Hungarian Notation	prefix with device name	`tseMdb`

Macros

- Macro names must be all uppercase.
- Words are 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: `mTSE_WRITE` is a valid name for a macro.

Constants

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

Structures

- Structure names must be all uppercase.
- Words are separated by an underscore.
- The letter `'s'` in lowercase must be used as a prefix to specify that it is a structure, then the device abbreviation must appear.
- Example: `sTSE_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 is used as a prefix.
- Example: `tseAdd()` 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 `'sys'` prefix is used to indicate a porting function.
- The device's name starting with an uppercase must follow the prefix.
- Example: `sysTSEReadReg()` is a hardware specific function.

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: `myOwnFunction()` is a valid name for such a function.

Variables

- Naming of variables must follow the Hungarian notation.
- A pointer to a variable shall use `'p'` as a prefix followed by the variable name unchanged. If the variable name already starts with a `'p'`, the first letter of the variable name may be capitalized, but this is not a requirement. Double pointers might be prefixed with `'pp'`, but this is not required.
- Global variables 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 `tseBaseAddress` 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 21 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, separate it into two or more files and add a number at the end of the file name (e.g. `tse_api1.c` or `tse_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 21: File Naming Conventions

File Type	File Name
API	`tse_api1.c, tse_api.h`
Hardware Dependent	`tse_hw.c, tse_hw.h`
RTOS Dependent	`tse_rtos.c, tse_rtos.h`
Other	`tse_init.c, tse_init.h`

API Files

- The name of the API files must start with the device abbreviation followed by an underscore and `'api'`. Eventually a number might be added at the end of the name.
- Example: `tse_api1.c` is the only valid name for the file that contains the first part of the API functions.

- Example: `tse_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'`.
- Example: `tse_hw.c` is the only valid name for the file that contains all of the hardware dependent functions.
- Example: `tse_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.
- Example: `tse_rtos.c` is the only valid name for the file that contains all of the RTOS dependent functions.
- Example: `tse_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: `tse_isr.c` is a valid name for a file that would deal with initialization of the device.
- Examples: `tse_isr.h` is a valid name for the corresponding header file.

APPENDIX B: ERROR CODES

This section of the manual describes the error codes used in the TSE device driver.

Table 22: TSE Error Codes

Error Code	Description
TSE_SUCCESS	Success
TSE_FAILURE	Failure
TSE_ERR_MEM_ALLOC	Not enough memory for allocation
TSE_ERR_INVALID_ARG	Invalid function parameter
TSE_ERR_MODULE_NOT_INIT	Module not initialized
TSE_ERR_MODULE_ALREADY_INIT	Module already initialized
TSE_ERR_INVALID_MIV	Invalid Module Initialization Vector
TSE_ERR_SEMAPHORE	Semaphore error
TSE_ERR_WRONG_STATE	Wrong module state
TSE_ERR_INT_INSTALL	Unable to install interrupt handler
TSE_ERR_STAT_INSTALL	Unable to install statistics task
TSE_ERR_CONNECT_NONEXISTENT	Connection does not exist
TSE_ERR_CONNECT_EXIST	Connection already exists
TSE_ERR_DEVS_FULL	Device table is full
TSE_ERR_DEVS_EMPTY	No devices in table
TSE_ERR_DEV_NOT_DETECTED	Failed to see manufacturer and device ID on the chip
TSE_ERR_DEV_ALREADY_ADDED	Device is already in table
TSE_ERR_INVALID_DEV	Invalid device handle passed to driver API
TSE_ERR_INVALID_STATE	Invalid device state

Error Code	Description
`TSE_ERR_INVALID_DIV`	Invalid Device Initialization Vector
`TSE_ERR_INVALID_MODE`	Invalid TSI mode
`TSE_ERR_NULL_BASE_ADDR`	NULL base address passed to tseAdd
`TSE_ERR_INVALID_REG`	Invalid register number
`TSE_ERR_POLL_TIMEOUT`	Indirect read/write busy bit timeout
`TSE_ERR_FAILRAMTEST`	RAM test failed
`TSE_ERR_CSUNOTLOCKED`	CSUs did not lock after a reset
`TSE_ERR_MODE`	Wrong mode, interrupt or polling
`TSE_NODEBUG`	Debug not installed, use `TSE_CSW_DEBUG` compile switch
`TSE_ERR_DEV_VERSION`	Driver not tested with this device rev, or a mode is not supported on the rev being using.
`TSE_ERR_INVALID_INDIR_VAL`	Invalid indirect register value

APPENDIX C: EVENT CODES

Table 23 below describes the interrupt event codes used in the TSE device driver.

Note that specific callback is defined by the "event" and "cause" fields of the `sTSE_DPV` structure (for the structure's definition, please refer to Table 18). Information encoded in these two fields explicitly defines the cause of the callback. The "event" field encodes the nature of the callback (e.g., `TSE_EVENT_DEVICE_CSU` represent a CSU changing lock state); the "cause" further indicates the absolute cause(s) of the callback event and will be either a port number or a block number.

Table 23: TSE Event Codes

Event Code	Description	Cause
`TSE_EVENT_DEVICE_CSU`	Clock Synthesis Unit	CSU number (1-4)
`TSE_EVENT_DEVICE_SSWT`	Space Switch	0
`TSE_EVENT_DEVICE_ITSE`	ITSE page switch	ITSE number (1-16)
`TSE_EVENT_DEVICE_ETSE`	ETSE page switch	ETSE number (1-16)
`TSE_EVENT_PORT_ROOC`	R8FA out of character alignment	Port number (1-64)
`TSE_EVENT_PORT_ROOF`	R8FA out of frame alignment	Port number (1-64)
`TSE_EVENT_PORT_RLCV`	R8FA line code violation	Port number (1-64)
`TSE_EVENT_PORT_RFIFO`	R8FA FIFO underrun/overrun	Port number (1-64)
`TSE_EVENT_PORT_TFIFO`	T8DE FIFO underrun/overrun	Port number (1-64)

List of Terms

APPLICATION: Refers to protocol software used in a real system as well as validation software written to validate the TSE 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 TSE 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: The module is all of the code that is part of this driver. There is only one instance of this module connected to one or more TSE 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

API:	Application Programming Interface
DDB:	Device Data Block
DIV:	Device Initialization Vector
DPR:	Deferred-Processing Routine
DPV:	Deferred-Processing (routine) Vector
FIFO:	First In, First Out
MDB:	Module Data Block
MIV:	Module Initialization Vector
ISR:	Interrupt-Service Routine
ISV:	Interrupt-Service (routine) Vector
RTOS:	Real-time operating system

Index