PM5310

TBS

DRIVER MANUAL

PROPRIETARY AND CONFIDENTIAL

RELEASE

ISSUE 3: NOVEMBER, 2001

ABOUT THIS MANUAL AND TBS

This manual describes the TBS device driver. It describes the driver's functions, data structures, and architecture. This manual focuses on the driver's interfaces to your application, real-time operating system, and to the 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 TBS devices
- Modify and add to the TBS driver's functions
- Port the TBS driver to a particular platform.

References

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

Table 1: Related Documents

Document Number	Document Name
PMC-1991257	TelecomBus serializer (TBS) 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	August 2000	Document created
Issue 2	October, 2000	Add description to a new overwrite mode in TSI mapping

Issue No.	Issue Date	Details of Change
Issue 3	November 2001	1) expand definition of function tbsIsMulticast 2) update tbsReadIndirect and tbsWriteIndirect prototypes 3) changes in sTBS_EVT_RX8D, sTBS_EVT_PRBS, sTBS_CNTR, sTBS_CFG_DEVICE data structures. 4) update DDB field to include PRBS STS-1 Path configuration and DIV 5) add one argument to API function tbsPayloadCfg to allow configuration block retrieval 6) add descriptions for data structures sTBS_CFG_PRBS and sTBS_CFG_PRBSPORT 7) add mutual exclusion semaphore in DDB to protect statistics access 8) add (a) tbsUpdate() function, (b) errStat in DDB 9) API tbsIX8EforceLcv() and tbsIX8EcenterFIFO() are retired 10) API tbsPrbsGenEnable() and tbsPrbsMonEnable() are renamed to tbsPrbsGenCfg() and tbsPrbsMonCfg() 11) remove ot8d_ofaais field and tbsGenAIS() configures only Rx8D blocks 12) remove fields sysclki, refclki changei and change in sTBS_STATUS_IO structure 13) add new field to DDB, BOOL tsiOverwrite to control the operating mode of the driver in TSI mapping.

Legal Issues

PMC-2001251, (P2), ref PMC-1991228 (P1)

Contacting PMC-Sierra

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

Tel: (604) 415-6000
Fax: (604) 415-6200

Document Information: document@pmc-sierra.com
Corporate Information: info@pmc-sierra.com
Technical Support: apps@pmc-sierra.com
Web Site: http://www.pmc-sierra.com

TABLE OF CONTENTS

About this Manual and TBS 2
Audience 2
References 2
Revision History 2
Legal Issues 4
Contacting PMC-Sierra 4

Table of Contents 5

1 Introduction 14

2 Driver Functions and Features 15

2.1 General Driver Functions 15
Open/Close Driver Module 15
Start/Stop Driver Module 15
Add/Delete Device 15
Device Initialization 15
Activate/De-Activate Device 15
Read/Write Device Registers 16
Interrupt Servicing/Polling 16
Statistics Collection 16

2.2 TBS Specific Driver Functions 16
Time Slot Interchange 16
Incoming 8B/10B Encoder 17
Transmit Disparity Encoder 17
Receive 8B/10B Decoder 17
PRBS Processors 17
Device Diagnostics 17
Statistics and Alarm Monitoring 18
Specific Callback Functions 18

3 Software Architecture 19

3.1 Driver External Interfaces 19
Application Programming Interface 19
Real-Time Operating System (RTOS) Interface 20
Hardware Interface 20

3.2 Main Components 20
Module Data-Block and Device(s) Data-Blocks 21
Interrupt-Service Routine 22
Deferred-Processing Routine 22
Alarms, Status, and Statistics 22

Input/Output Configuration 22
Time Slot Interchange 23
Incoming 8B/10B Encoder 23
Receive 8B/10B Decoder 23
Disparity Decoder 23
PRBS Processors 23

3.3 Software States 23
Module States 24
Device States 25

3.4 Processing Flows 25
Module Management 26
Device Management 26

3.5 Interrupt Servicing 27
Calling tbsISR 28
Calling tbsDPR 29
Calling tbsPoll 29

3.6 Theory of Operation 30
TBS Overview 30
Time Slot Mapping 32
FIFO Centering 35
Alarms, Status and Statistics 35
Diagnostics 35

4 Data Structures 38

4.1 Constants 38

4.2 Data Structures 38
Structures Passed by the Application 39
Module Initialization Vector: MIV 39
Device Initialization Vector: DIV 39
Device Configuration Block: DEVICE 42
Payload Configuration Block: PYLD 43
TSI Connection Map: CONMAP 44
TSI Connection Page: CONPAGE 44
TSI Space-Time Slot: SPTSLOT 45
ISR Enable/Disable Mask 45
8B/10B Decoder Events: EVT_RX8D 46
PRBS Processor Events: EVT_PRBS 47
Structures in the Driver's Allocated Memory 47
Module Data Block: MDB 47
Device Data Block: DDB 48
Statistics Block: STAT 50
Event/Statistics Counter Structure: CNTR 50
Device Status Block: STATUS 52

I/O Status Block: STATUS_IO 52
8B/10B Decoder Status Block: STATUS_RX8D 53
PRBS Monitor Status Block: STATUS_PRBS 53
PRBS Generator/Monitor Configuration Block: CFG_PRBS PORT 53
PRBS Generator/Monitor STS-1 Configuration Parameters: CFG_PRBS 54
Structures Passed through RTOS Buffers 55
Interrupt Service Vector: ISV 55
Deferred Processing Vector: DPV 55

4.3 Global Variable 56

5 Application Programming Interface 57

5.1 Module Management 57
Opening the Driver Module: tbsModuleOpen 57
Closing the Driver Module: tbsModuleClose 57
Starting the Driver Module: tbsModuleStart 58
Stopping the Driver Module: tbsModuleStop 58

5.2 Device Management 59
Adding a Device: tbsAdd 59
Deleting a Device: tbsDelete 60
Initializing a Device: tbsInit 60
Updating the Configuration of a Device: tbsUpdate 61
Resetting a Device: tbsReset 61
Activating a Device: tbsActivate 61
De-Activating a Device: tbsDeActivate 62

5.3 Device Read and Write 62
Reading from Device Registers: tbsRead 62
Writing to Device Registers: tbsWrite 63
Reading from a block of Device Registers: tbsReadBlock 63
Writing to a Block of Device Registers: tbsWriteBlock 64
Indirect reading from Device Registers: tbsReadIndirect 65
Indirect writing to Device Registers: tbsWriteIndirect 65

5.4 Device Configuration 66
Setting Device Configuration Block: tbsDeviceSetConfig 66
Getting Device Configuration Block: tbsDeviceGetConfig 66

5.5 Time Slot Interchange 67
Setting global TSI mapping mode: tbsSetMapMode 67
Getting global TSI mapping mode: tbsGetMapMode 68
Setting active connection page in TSI: tbsSetPage 68
Getting active connection page in TSI: tbsGetPage 69
Mapping the source to destination slot(s) in TSI: tbsMapSlot 69
Retrieving source space-time Slot in TSI: tbsGetSrcSlot 70
Retrieving destination space-time Slot in TSI: tbsGetDestSlot 70
Copying connection map from one page to another in TSI: tbsCopyPage 71

Inserting Idle Data in TSI: tbsInsIdleData 71
Removing established connection in TSI: tbsRmSlot 72
Clearing all established connections in TSI: tbsClrSlot 73
Verifying a multicast connection in TSI: tbsIsMulticast 73
Verifying the connection map setting in TSI: tbsIsValidMap 74

5.6 8B/10B Decoder 75
Forcing out of character alignment in 8B/10B decoder: tbsForceOutofChar 75
Forcing out of frame alignment in 8B/10B decoder: tbsForceOutofFrm 75

5.7 Disparity Encoder 76
Forcing line code violation in disparity encoder: tbsTXDEForceLcv 76
Centering FIFO in disparity encoder: tbsTXDECenterFIFO 76
Inserting test pattern in disparity encoder: tbsInsertTP 77

5.8 PRBS Processors 77
Configuring and retrieving payload for PRBS processor: tbsPayloadCfg 77
Configuring PRBS generator: tbsPrbsGenCfg 78
Forcing a bit error in PRBS generator: tbsPrbsForceBitErr 79
Configuring PRBS monitor: tbsPrbsMonCfg 79
Resynchronizing PRBS monitor: tbsPrbsResync 80

5.9 Interrupt Service Functions 80
Configuring ISR Processing: tbsISRConfig 81
Getting the Interrupt Status Mask: tbsGetMask 81
Setting the Interrupt Enable Mask: tbsSetMask 82
Clearing the Interrupt Enable Mask: tbsClearMask 82
Polling the Interrupt Status Registers: tbsPoll 82
Interrupt Service Routine: tbsISR 83
Deferred Processing Routine: tbsDPR 83

5.10 Alarm, Status, and Statistics Functions 84
Getting the Cumulative Device Statistics: tbsGetStats 84
Clearing the Device Statistics: tbsClrStats 84
Getting Status of the Device : tbsGetStatus 85
Getting Delta Statistics Counter of the Device : tbsGetDelta 85
Getting Event Threshold of the Device : tbsGetThresh 86
Setting Event Threshold of the Device : tbsSetThresh 86
Controlling AIS Generation of the Device : tbsGenAIS 87

5.11 Device Diagnostics 87
Testing Register Accesses: tbsTestReg 87
Testing RAM Accesses: tbsTestRAM 88
Enabling outgoing to incoming parallel TelecomBus Loopbacks: tbsLoopOut2InTCB 88
Enabling incoming to outgoing parallel TelecomBus Loopbacks: tbsLoopIn2OutTCB 88
Enabling receive to transmit serial TelecomBus Loopbacks: tbsLoopRx2TxLVDS 89
Enabling transmit to receive serial TelecomBus Loopbacks: tbsLoopTx2RxLVDS 89

5.12 Callback Functions 90

Calling Back to the Application due to IO events: cbackIO 90
Calling Back to the Application due to TSI events: cbackTSI 91
Calling Back to the Application due to PRBS events: cbackPRBS 91
Calling Back to the Application due to TXDE events: cbackTXDE 92
Calling Back to the Application due to RX8D events: cbackRX8D 92

6 Hardware Interface 93

6.1 Device I/O 93
Reading from a Device Register: sysTbsRead 93
Writing to a Device Register: sysTbsWrite 93

6.2 System-Specific Interrupt Servicing 94
Installing the ISR Handler : sysTbsISRHandlerInstall 94
ISR Handler: sysTbsISRHandler 94
Removing the ISR Handler : sysTbsISRHandlerRemove 95

7 RTOS Interface 96

7.1 Memory Allocation / De-Allocation 96
Allocating Memory: sysTbsMemAlloc 96
Freeing Memory: sysTbsMemFree 96

7.2 Buffer Management 97
Starting Buffer Management : sysTbsBufferStart 97
Getting an ISV Buffer: sysTbsISVBufferGet 97
Returning an ISV Buffer: sysTbsISVBufferRtn 97
Getting a DPV Buffer: sysTbsDPVBufferGet 98
Returning a DPV Buffer: sysTbsDPVBufferRtn 98
Stopping Buffer Management : sysTbsBufferStop 98

7.3 Timers 99
Sleeping a Task: sysTbsTimerSleep 99

7.4 Semaphores 99
Creating a Semaphore: sysTbsSemCreate 99
Taking a Semaphore: sysTbsSemTake 100
Giving a Semaphore: sysTbsSemGive 100
Deleting a Semaphore: sysTbsSemDelete 100

7.5 Preemption 101
Disabling Preemption : sysTbsPreemptDisable 101
Re-Enabling Preemption : sysTbsPreemptEnable 101

7.6 System-Specific DPR Routine 101
Installing the DPR Task: sysTbsDPRTaskInstall 102
DPR Task: sysTbsDPRTask 102
Removing the DPR Task: sysTbsDPRTaskRemove 103

8 Porting the TBS Driver 104

8.1 Driver Source Files 104

8.2 Driver Porting Procedures 104
Procedure 1: Porting Driver OS Extensions 105
Procedure 2: Porting Drivers to Hardware Platforms 106
Procedure 3: Porting Driver Application-Specific Elements 106
Procedure 4: Building the Driver 108

Appendix A: Coding Conventions 109
Variable Type Definitions 109
Naming Conventions 109
Macros 110
Constants 110
Structures 110
Functions 111
Variables 111
File Organization 112
API Files 112
Hardware Dependent Files 112
RTOS Dependent Files 113
Other Driver Files 113

Appendix B: Error Codes 114

Appendix C: Event Codes 116

List of Terms 121

Acronyms 122

Index 123

LIST OF FIGURES

Figure 1: Driver External Interfaces 19

Figure 2: Driver Architecture 21

Figure 3: Driver Software States 24

Figure 4: Module Management Flow Diagram 26

Figure 5: Device Management Flow Diagram 27

Figure 6: Interrupt Service Model 28

Figure 7: Polling Service Model 30

Figure 8: Device Traffic Flow Illustration 30

Figure 9: Time Slot Interchange Model 32

Figure 10: Space-time Slot Mapping, Multicast and Unicast 33

Figure 11: PRBS Processor Model 36

LIST OF TABLES

Table 1: TBS Block Description 31

Table 2: TBS Module Initialization Vector: sTBS_MIV 39

Table 3: TBS Device Initialization Vector: sTBS_DIV 40

Table 4: TBS Device Configuration Data Structure: sTBS_CFG_DEVICE 42

Table 5: TBS Payload Configuration Block: sTBS_CFG_PYLD 43

Table 6: TBS TSI Connection Map Data Structure: sTBS_TSI_CONMAP........ 44

Table 7: TBS TSI Connection Page Data Structure: sTBS_TSI_CONPAGE........ 44

Table 8: TBS Space-time Slot Data Structure: sTBS_SPTSLOT 45

Table 9: TBS ISR Mask: sTBS_MASK........ 45

Table 10: TBS 8B/10B Decoder Event: sTBS_EVT_RX8D........ 46

Table 11: TBS PRBS Monitor Event: sTBS_EVT_PRBS........ 47

Table 12: TBS Module Data Block: sTBS_MDB........ 48

Table 13: TBS Device Data Block: sTBS_DDB........ 49

Table 14: TBS Event Statistics Block: sTBS_STAT 50

Table 15: TBS Events Counter Block: sTBS_CNTR 51

Table 16: TBS Device Status Block : sTBS_STATUS........ 52

Table 17: TBS I/O Block Status: sTBS_STATUS _IO 52

Table 18: TBS 8B/10B Decoder Block Status: sTBS_STATUS _RX8D........ 53

Table 19: TBS PRBS Monitor Status: sTBS_STATUS _PRBS........ 53

Table 20: TBS PRBS Generator/Monitor Configuration Block Per Port: sTBS_CFG_PRBS
PORT 54

Table 21: TBS PRBS Generator/Monitor Configuration Parameters Per Time Slot :
sTBS_CFG_PRBS........ 54

Table 22: TBS Interrupt Service Vector: sTBS_ISV........ 55

Table 23: TBS Deferred Processing Vector: sTBS_DPV ... 55
Table 24: Variable Type Definitions ... 109
Table 25: Naming Conventions ... 109
Table 26: File Naming Conventions ... 112
Table 27: TBS Error Codes ... 114
Table 28: TBS Event Codes ... 116

1 INTRODUCTION

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

Section 3 of this document, Software Architecture, defines the software architecture of the TBS 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 configure or control its behavior. Included here are the constants, variables and structures that the TBS 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 TBS driver Application Programming Interface (API). The section outlines function calls that hide device-specific details and application callbacks that notify the user of significant device events.

For your convenience, Section 8 of this manual provides a brief guide for porting the TBS device driver to your hardware and RTOS platform. In addition, an extensive Appendix (page 109) and Index (page 123) provides you with useful reference information.

2 DRIVER FUNCTIONS AND FEATURES

This section summarizes the main functions and features supported by the TBS driver. A more detailed description will follow in later sections.

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

Activate/De-Activate Device

Activating a device puts it into its normal mode of operation by enabling interrupts and other global registers. A successful device activation also enables other API invocations.

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

2.2 TBS Specific Driver Functions

These functions provide control and monitoring of the various sections of the TBS 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.

The following sub-sections list out all the major blocks in the TBS and their key features. Functions will be designed to configure, control and/or monitor those main blocks. The last sub-section is devoted to callback functions specific to the TBS for application interface.

Time Slot Interchange

Functions are provided to control/configure all time-slot interchange blocks in the TBS including the transmit working, protection, and auxiliary TSI, as well as the receive working, protection, and auxiliary TSI. These functions perform:

- read/write to and from the dual connection memory pages (pages 0 & 1) for each block
- switch between the two connection pages (active and inactive) via software control

Incoming 8B/10B Encoder

A function is provided to control/configure all the 8B/10B encoders in TBS:

- configure the mode of incoming TelecomBus data stream (MST, or HPT)

Transmit Disparity Encoder

Functions are available for controlling all three transmit disparity encoders, namely the working, protection, and auxiliary. These functions perform:

- insertion of line code violation into data stream
- insertion of test patterns
- idle data insertion

Receive 8B/10B Decoder

This section is composed of all three receive 8B/10B decoders, namely the working, protection, and auxiliary. Functions are available for:

- reporting out-of-character alignment and out-of-frame alignment conditions
- controlling insertion of AIS alarm
- reporting line code violation counts

PRBS Processors

This section consists of all PRBS processors in the TBS including all four incoming TelecomBus PRBS processors and the working, protection, and auxiliary receive PRBS processors. These functions perform the following:

- configure and control the PRBS Generator (to facilitate downstream equipment diagnostics)
- insert the PRBS data pattern, B1/E1 byte, single bit error
- configure the data stream payload (STS-48c, STS-36c, STS-24c, STS-12c, STS-3c or STS-1)
- configure and monitor the PRBS Detector (to facilitate upstream equipment diagnostics)
- resynchronize the PRBS sequence, control the PRBS pattern, compare B1/E1 byte, report error count
- enable/disable event/error monitor interrupts, such as B1/E1 mismatch, byte error interrupts, and synchronized status change

Device Diagnostics

- device register read/write test
- outgoing to incoming TelecomBus loopback;
- incoming to outgoing TelecomBus loopback;

- Rx to Tx LVDS loopback
- Tx and Rx LVDS loopback

Statistics and Alarm Monitoring

Functions are provided to gather statistics and do alarm monitoring; they are:

- get/clear cumulative statistics
- get delta statistics
- get/set thresholds

Specific Callback Functions

Callback functions are available to the application for event notification from the device driver. The application will be notified via the callback functions for selected events of interest, such as I/O events.

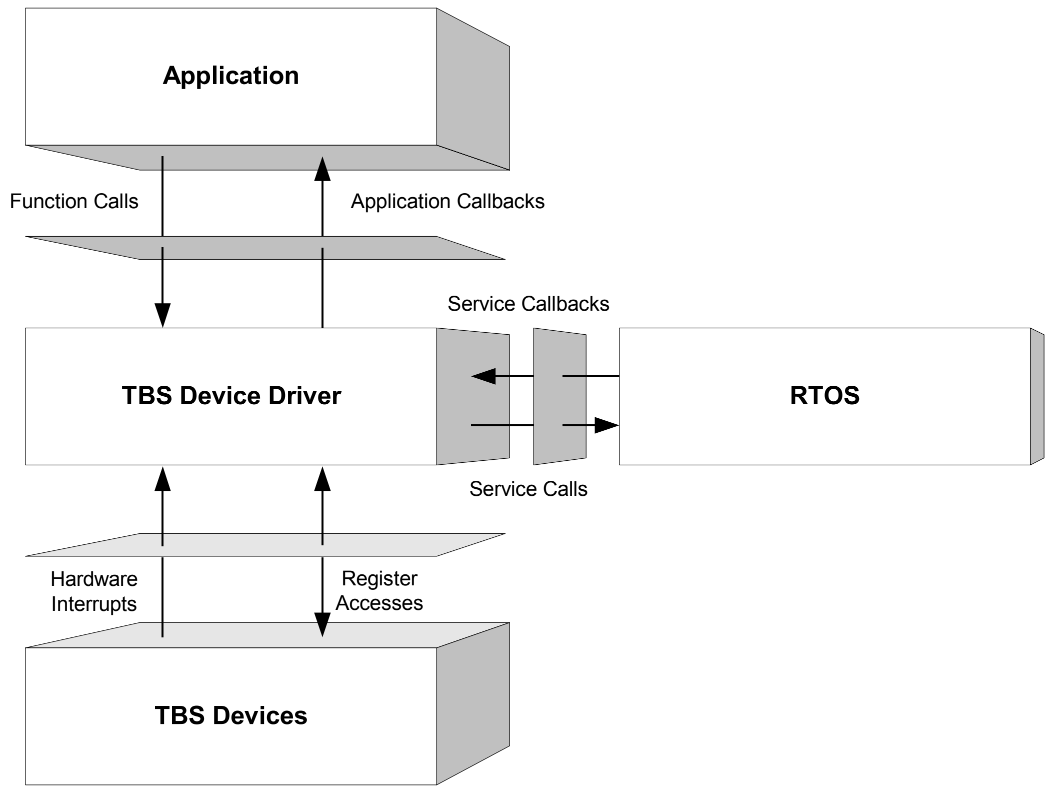
3 SOFTWARE ARCHITECTURE

This section describes the software architecture of the TBS 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 TBS 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 the TBS devices. The API functions perform operations that are more meaningful from a systems perspective. The API includes functions such as:

- 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 Operating System (RTOS) 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
- take and give semaphores
- enable and disable preemption

The RTOS interface also includes service callbacks. These are functions installed by the driver using RTOS service calls, such as installing interrupts and starting timers. These service callbacks are invoked when an interrupt occurs or a timer expires.

Note: Users must modify the RTOS interface code to suit their RTOS.

Hardware Interface

The hardware interface provides functions that read from and write to the device registers; it 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

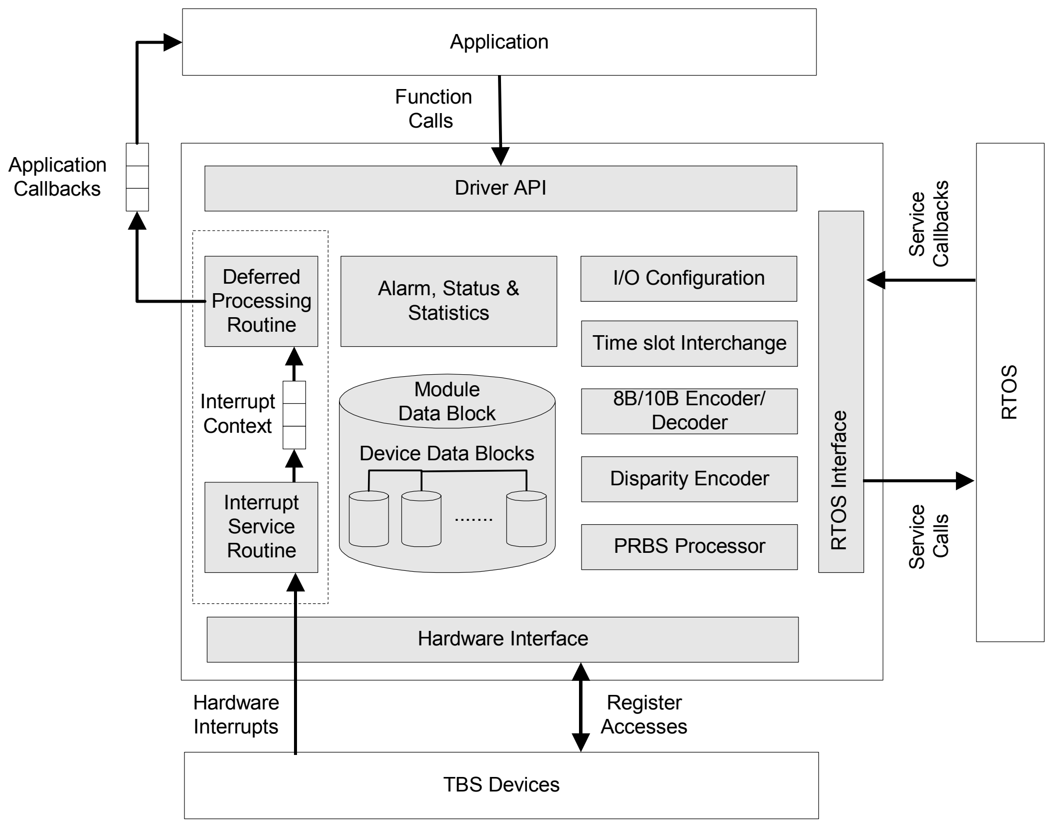
Figure 2 illustrates the top-level architectural components of the TBS device driver. This applies in both polled and interrupt driven operation. In polled operation, the ISR is called periodically. In interrupt operation, the interrupt directly triggers the ISR.

The driver includes the following main components:

- Module and device(s) data-blocks
- Interrupt-service routine

- Deferred-processing routine
- Alarm, status, and statistics
- I/O Configuration
- Time Slot Interchange
- 8B/10B Encoder/Decoder
- Disparity Encoder
- PRBS Processor

Figure 2: Driver Architecture



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

The Module Data-Block (MDB) is the top-layer data structure; it is created by the TBS 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 is initialized by the driver module for each TBS device that is registered. There is one DDB per device, and there is a limit on the number of DDBs; that limit is set by the user when the module is initialized. The DDB is used to store context information about one device, such as:

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

Interrupt-Service Routine

The TBS driver provides an ISR called `tbsISR` that checks to see if there is any valid interrupt condition present for the device. This function can be used by a system-specific interrupt-handler function to service interrupts raised by the device.

The low-level interrupt-handler function that traps the hardware interrupt and calls `tbsISR` is system and RTOS dependent. Therefore, it is outside the scope of the driver. Example implementations of an interrupt handler, as well as 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 the interrupt-servicing model.

Deferred-Processing Routine

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

Example implementations of a DPR task, as well as functions that install and remove it, are provided as a reference in section 7.6. You can customize these example implementations to suit your specific needs.

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

Alarms, Status, and Statistics

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

Input/Output Configuration

This section controls the data traffic on both the transmit and the receive side in the TelecomBus and the LVDS serial links. Attributes such as data parity, traffic concatenation level (STS-48c, STS-12c, STS-3c or STS-1), and J0 byte processing delay are to be set up correctly for proper operation.

Time Slot Interchange

The section controls all time-slot interchange modules in the TBS including the transmit working, protection, and auxiliary TSI, as well as the receive working, protection and auxiliary TSI. A TSI operates as an independent entity and possesses a dual-connection memory page mechanism. This two-page setup allows users to switch to an alternate timeslot mapping without disrupting the current operation. Memory page switching can be achieved by either software control or via external hardware pins.

Incoming 8B/10B Encoder

This section controls all the 8B/10B encoders in the TBS to configure the mode of the incoming TelecomBus data stream (MST, or HPT

Receive 8B/10B Decoder

This section controls all three receive 8B/10B decoders, namely the working, protection, and auxiliary. The decoders are responsible for AIS alarm insertion, character and frame alignment, out-of-character or out-of-frame alignment monitoring, and line code violation counting.

Disparity Decoder

This section controls all three transmit disparity encoders, namely the working, protection and auxiliary in the TBS; it handles insertion of line code violations, test patterns, and idle data.

PRBS Processors

This section controls all PRBS processors in the TBS, including all four incoming TelecomBus PRBS processors, as well as the working, protection, and auxiliary receive PRBS processors. A PRBS processor can be further subdivided into a PRBS generator and a PRBS monitor.

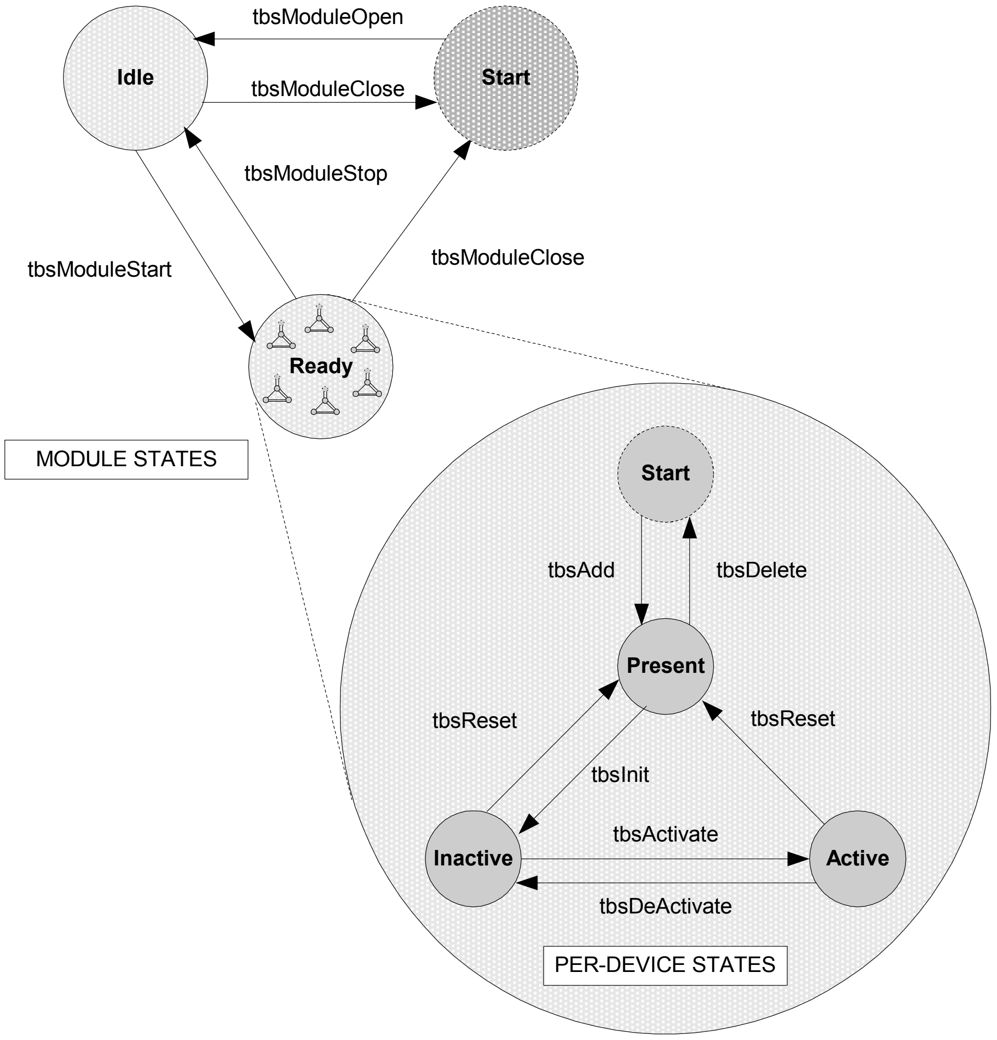
The PRBS generator is used to facilitate downstream equipment diagnostics. It handles the PRBS data pattern, the B1/E1 byte, and single-bit error insertion. Proper payload configuration (STS-48c, STS-36c, STS-24c, STS-12c, STS-3c, or STS-1) is necessary to ensure correct operation.

The PRBS monitor is used for upstream equipment diagnostics; it does this through operations such as PRBS pattern monitoring, comparison of B1/E1 byte, and PRBS sequence resynchronization. It also requires correct payload configuration to function properly.

3.3 Software States

Figure 3 shows the software state diagram for the TBS 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 TBS 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 TBS per-device states. The state that is mentioned here is the software state as maintained by the driver; it is not the software state maintained inside the device itself. See section 5.2 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 to the driver and performs no actions. There is a separate flow for each device that can be added, and they all start here.

Present

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

Inactive

In this state, the device is configured but all data functions are de-activated, including interrupts and alarms, and 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

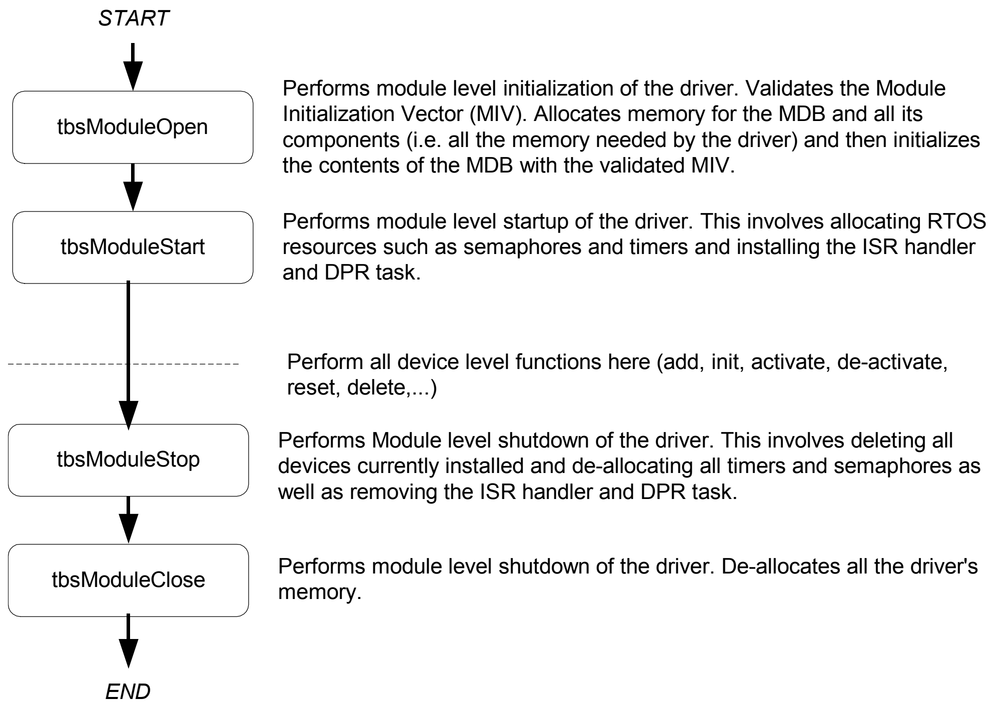
This section describes the main processing flows of the TBS's driver components.

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 TBS driver module.

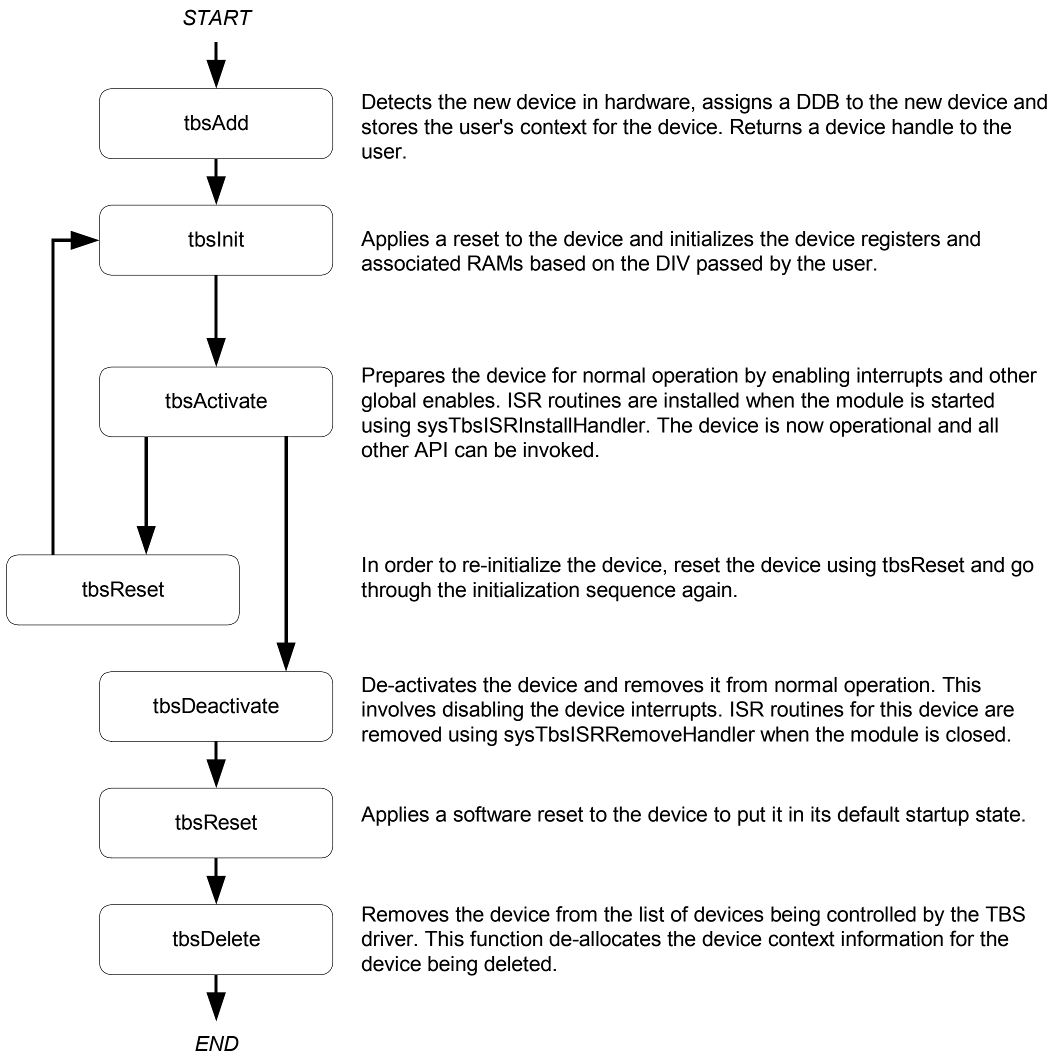
Figure 4: Module Management Flow Diagram


START
tbsModuleOpen
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.
tbsModuleStart
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,...)
tbsModuleStop
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.
tbsModuleClose
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 TBS device.

Figure 5: Device Management Flow Diagram


START
tbsAdd
Detects the new device in hardware, assigns a DDB to the new device and stores the user's context for the device. Returns a device handle to the user.
tbsInit
Applies a reset to the device and initializes the device registers and associated RAMs based on the DIV passed by the user.
tbsActivate
Prepares the device for normal operation by enabling interrupts and other global enables. ISR routines are installed when the module is started using sysTbsISRInstallHandler. The device is now operational and all other API can be invoked.
tbsReset
In order to re-initialize the device, reset the device using tbsReset and go through the initialization sequence again.
tbsDeactivate
De-activates the device and removes it from normal operation. This involves disabling the device interrupts. ISR routines for this device are removed using sysTbsISRRemoveHandler when the module is closed.
tbsReset
Applies a software reset to the device to put it in its default startup state.
tbsDelete
Removes the device from the list of devices being controlled by the TBS driver. This function de-allocates the device context information for the device being deleted.
END

3.5 Interrupt Servicing

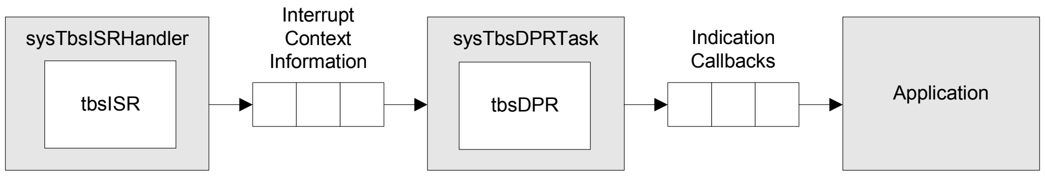
The TBS driver services device interrupts using an interrupt service routine (ISR) that traps interrupts; the driver also uses a deferred processing routine (DPR) that processes the interrupt conditions and clears them; this lets the ISR execute quickly and exit. Most of the time-consuming processing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

The driver provides system-independent functions, `tbsISR` and `tbsDPR`. You must fill in the corresponding system-specific functions, `sysTbsISRHandler` and `sysTbsDPRTask`. The system-specific functions isolate the system-specific communication mechanism (between the ISR and DPR) from the system-independent functions, `tbsISR` and `tbsDPR`.

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

Figure 6: Interrupt Service Model



Note: Instead of using an interrupt service model, you can use a polling service model in the TBS driver to process the device's event-indication registers (see page 39).

Calling tbsISR

An interrupt handler function, which is system dependent, must call `tbsISR`. 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 (`sysTbsISRHandler`) appears on page 94. You can customize this example implementation to suit your needs.

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

The `tbsISR` function reads from the master interrupt-status registers and disables the interrupt cause. If at least one valid interrupt condition is found, then `tbsISR` fills an Interrupt Service Vector (ISV) with this status information as well as the current device handle. The `tbsISR` function also clears and disables all the device's detected interrupts. The `sysTbsISRHandler` 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 using the `sysTbsISRHandler`.

Calling tbsDPR

The `sysTbsDPRTask` 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 that of any application tasks that are interacting with the TBS 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, `sysTbsDPRTask` calls the DPR (`tbsDPR`) with the received ISV.

At that point, `tbsDPR` processes the status information and takes appropriate action based on the specific interrupt condition detected; it also 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, `tbsDPR` 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 `tbsDelete`. Also, ensure that the callback function is non-blocking because the DPR task executes while TBS interrupts are disabled. You can customize these callbacks to suit your system. See page 88 for example implementations of the callback functions.

Note: Since the `tbsISR` and `tbsDPR` 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, `sysTbsISRHandler` and `sysTbsDPRTask.` When the driver calls `sysTbsISRHandlerInstall,` the application installs `sysTbsISRHandler` in the interrupt vector table of the processor, and the `sysTbsDPRTask` function is spawned as a task by the application. The `sysTbsISRHandlerInstall` function also creates the communication channel between `sysTbsISRHandler` and `sysTbsDPRTask`. This communication channel is most commonly a message queue associated with the `sysTbsDPRTask`.

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

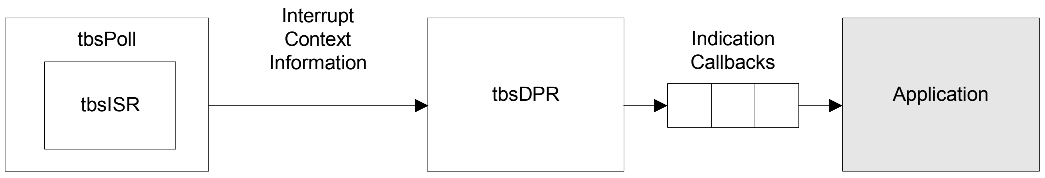
As a reference, this manual provides example implementations of the interrupt installation and removal functions on pages 94 and 101. Users can customize these prototypes to suit their specific needs.

Calling tbsPoll

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

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

Figure 7: Polling Service Model



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

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

3.6 Theory of Operation

TBS Overview

Figure 8: Device Traffic Flow Illustration

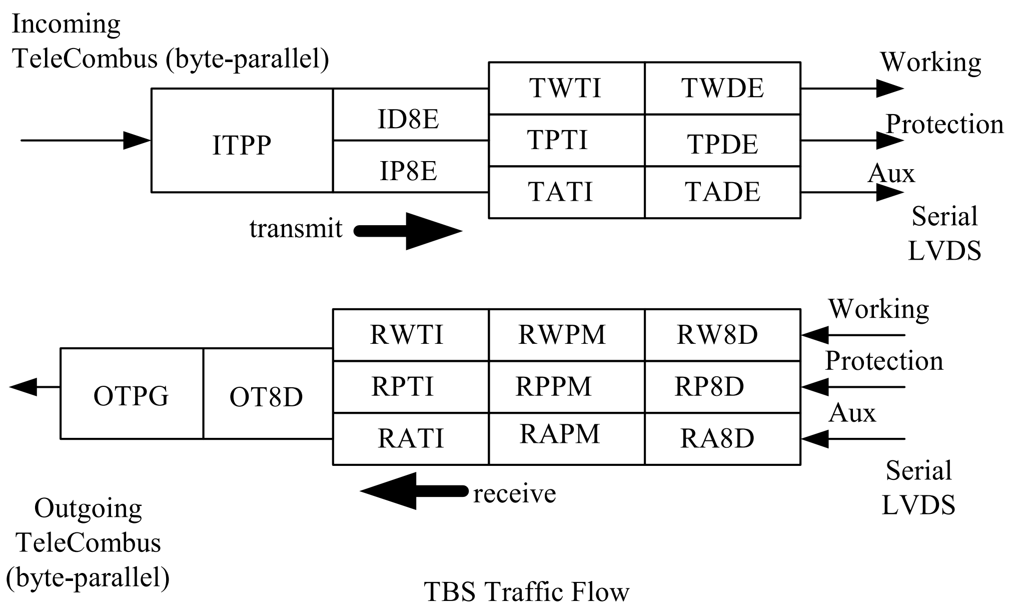


Table 1: TBS Block Description

Logical Block Name	Software Block Name	Definition
ITPP	PRBS[TRANSMIT][x]*	Incoming TelecomBus PRBS processor
ID8E & IP8E	IX8E	Incoming 8B/10B encoders
TWTI	TSI[TRANSMIT] [WORKING]	Transmit working TSI
TPTI	TSI[TRANSMIT] [PROTECTION]	Transmit protection TSI
TATI	TSI[TRANSMIT][AUX]	Transmit auxiliary TSI
TWDE	TXDE[WORKING]	Transmit disparity encoder
TPDE	TXDE[PROTECTION]	Transmit disparity encoder
TADE	TXDE[AUX]	Transmit disparity encoder
RW8D	RX8D[WORKING]	Receive working 8B/10B decoder
RP8D	RX8D[PROTECTION]	Receive protection 8B/10B decoder
RA8D	RX8D[AUX]	Receive auxiliary 8B/10B decoder
OT8D	OT8D	Outgoing 8B/10B decoder
RWPM	PRBS[RECEIVE] [WORKING]	Receive working PRBS monitor
RPPM	PRBS[RECEIVE] [PROTECTION]	Receive protection PRBS monitor
RAPM	PRBS[RECEIVE][AUX]	Receive auxiliary PRBS monitor
RWTI	TSI[RECEIVE] [WORKING]	Receive working TSI
RPTI	TSI[RECEIVE] [PROTECTION]	Receive protection TSI
RATI	TSI[RECEIVE][AUX]	Receive auxiliary TSI
OTPG	PRBS[RECEIVE][x]*	Outgoing PRBS generator

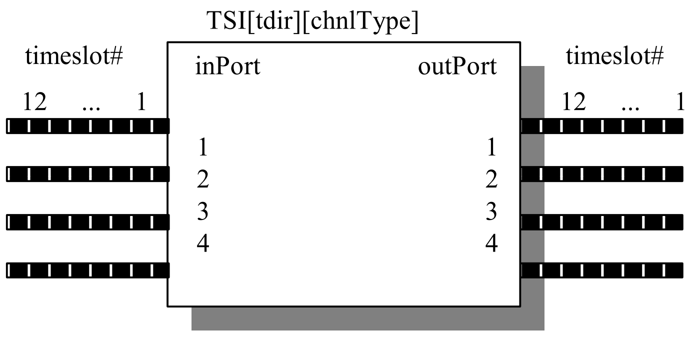
Notes: *x denotes don't-care terms in the channel type; they can be either WORKING, PROTECTION, or AUX.

The TBS implements byte-parallel TelecomBus and bit-serial 8B/10B-based TelecomBus conversion. On the transmit (ingress) side, the TBS connects an incoming, parallel TelecomBus data stream to a set of three (working, protection, and auxiliary) serial LVDS (TelecomBus) bit streams. 8B/10B encoders (IX8E) are present to encode the incoming data into an extended set of 8B/10B characters. Special 8B/10B characters encode transport and payload frame boundaries, pointer justification events, and alarm conditions. The PRBS processor (ITPP) is present to monitor incoming payloads and overwrite PRBS patterns for diagnosis by downstream equipment. A set of three time-slot interchanges (TSI) is designed for arbitrary mapping of time slots at STS-1 granularity. Multicast is supported. Disparity encoders (TXDE) are present to handle a polarity change after a possible time slot re-mapping.

On the receive (egress) side, the TBS connects three independent serial TelecomBus links to the outgoing byte-parallel TelecomBus. 8B/10B decoders (Rx8D) are employed to decode received 8B/10B characters and control signals. There are a set of three TSIs that are used to handle arbitrary time slot remapping from the received data to the outgoing TelecomBus. The PRBS processors (OTPG and RxPM) are also available to monitor the decoded payload and to generate PRBS traffic patterns for diagnosis by downstream equipment.

Time Slot Mapping

Figure 9: Time Slot Interchange Model

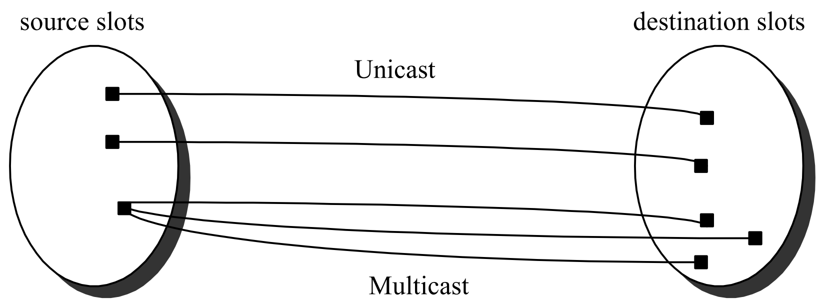


The TBS has a total of 6 time-slot interchange units (TSI) for time-slot mapping; each of these is capable of handling a STS-48 data stream. A TSI is described by 2 parameters, `tdir` and `chnlType`; these denote the traffic direction (TRANSMIT or RECEIVE) and the channel type (WORKING, PROTECTION, or AUX) respectively. 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 channel's data rate. The user has the responsibility to maintain data integrity when redefining the connection map.

Time-slot interchange can be viewed as a process of mapping source time slots to destination time slots. 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.

The TSI model appears to be switching in both space and time (i.e., one can map an STS-1 timeslot from port#1, timeslot#1 to port#3, timeslot#10). It is important to point out that the TSI is actually performing `only` timeslot remapping. The apparent space switching is due to the STS-48 stream multiplexing across 4 STS-12 links (one can re-label the timeslots from #1 to #48 instead of 4 ports each with timeslots from #1 to #12). We will, however, retain this space- and time-switching view; a view that matches the hardware layout.

Figure 10: Space-time Slot Mapping, Multicast and Unicast



`tbsMapSlot` establishes the mapping between the source space-time slot and the destination space-time slot(s). `tbsRmSlot` disconnects the established connection between the given source and destination slots. `tbsClrSlot` clears all connections for the given source slot.

The user must be in one of two modes when mapping slots: normal or overwrite. The user can switch modes at will by calling the API function `tbsDeviceSetConfig`.

In *normal* mode, the driver keeps track of all connections. Before connecting to an already occupied time slot, the user must first remove the old connection via `tbsRmSlot` or `tbsClrSlot`. This provides additional protection in setting up and tearing down connections.

In *overwrite* mode, the user has more autonomy in connection management. The driver does not check existing connections prior to setting up new ones, because `tbsMapSlot` essentially overwrites any existing connections. The user is fully responsible for keeping track. Note that if it is necessary to update active TSI pages in the application, using overwrite mode guarantees hitless updates.

The function `tbsGetDestSlot` returns the destination slot(s) when given the source slot. `tbsGetSrcSlot` returns the source slot when given the destination slot. `tbsIsMulticast` verifies if the given slot is mapped to multiple destination slots. `tbsSetMapMode` 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; when in this mode, time-slot rearrangement will not take place. If the user-defined mode is selected, time slots will be re-arranged based on the connection map inside the device. `tbsGetMapMode` retrieves the current mapping mode of the device.

There are two connection pages in each TSI, page 0 and 1. `tbsSetPage` 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. `tbsGetPage` queries the current active connection page. For connection page synchronization, `tbsCopyPage` 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
1	0	1
1	1	0

For each time slot in the outgoing TelecomBus, it can accept data from one and only one source time slot from either the working, protection, or auxiliary TSI. `tbsIsValidMap` verifies the connection map setting in all the receive TSIs to see if more than one source time slot (from the receive TSIs) has been assigned to an outgoing TelecomBus time slot. It is important to point out that even if multiple RxTSEN bits are high, the RWTSEN bit will take precedence and the data from the working channel will be selected over that of other channels. In essence, `tbsIsValidMap` looks for multiple ones settings of the RxTSEN bits in the TSIs.

When mapping time slots on the transmit side, there is an extra parameter to set for selecting where the data source is (either from ID8E or IP8E blocks). On the transmit side, the data stream flows from the ITPP block to both the ID8E and the IP8E block. As a result, the TxTI blocks have a choice of selecting data either from the ID8E or IP8E block on a per time-slot basis. Hence, there is an extra field, srcSel, in the data structure `sTBS_SPTSLOT`, governing where the data is drawn from when mapping slots in TxTI. In the implementation, srcSel=0 selects ID8E and a non-zero value for srcSel chooses IP8E as the data source. This extra control allows users to mix PRBS data and/or clear data coming out from the ITPP in one single TxTI block. In addition, `tbsGetSrcSlot` and `tbsGetDestSlot` use this to specify if the timeslot belongs to the ID8E or IP8E block on the transmit side. The parameter is ignored when performing RxTI mapping.

FIFO Centering

`tbsTXDECenterFIFO` forces the FIFO depth in the transmit disparity encoders to be 4 8B/10B characters deep if the current FIFO depth is not in the range of 3, 4 or 5 characters. If the current FIFO depth is in the 3, 4 or 5 range, this function has no effect. This function should be invoked when either: (1) the external system clock stabilizes or (2) after a FIFO overrun/underrun error is detected as part of a recovery procedure.

Alarms, Status and Statistics

This section handles alarm generation, retrieval and maintenance of statistics/event counters, and status retrieval.

Statistics are accumulated and events are counted inside the status block. For each statistic/event, cumulative as well as delta counters are kept. The delta counter contains the statistic/event count since the last query. `tbsGetStats` and `tbsGetDelta` retrieve the cumulative and delta statistics counters respectively. The Delta statistics counters are cleared after a query. `tbsClrStats` clears the cumulative statistics counters when invoked. Instead of having callbacks for each and every event (or statistic), the user may wish to only have callbacks after a user-specified count/occurrence is detected/encountered. `tbsGetThresh` and `tbsSetThresh` are for dynamically retrieving and setting the event/statistic thresholds, respectively. In addition, if zero or one is set as threshold for a particular event, callbacks will be made for every event occurrence.

Device status can be retrieved by invoking `tbsGetStatus`. Many signals − such as synchronization state of an 8B/10B decoder, B1/E1 byte received by a PRBS monitor, and bus activity indicator − can be instantly read back using this function.

tbs`GenAIS` enables/disables the insertion of AIS alarms in the event of out-of-frame alignment being detected in the receive 8B/10B decoders.

Diagnostics

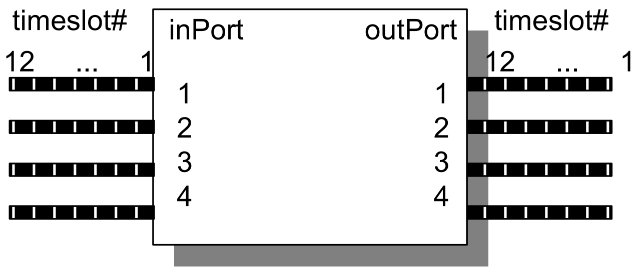
The purpose of this section is to perform system/device diagnostics. These diagnostics would include loopbacks, generation of line-code violations on a TelecomBus, PRBS traffic insertion, and error insertion.

Loopback Functions

The loopback functions allow several blocks inside the TBS to be placed in diagnostic loopback operation. `tbsLoopOut2InTCB` loops back the outgoing TelecomBus to the incoming one; while `tbsLoopIn2OutTCB` loops back the incoming TelecomBus to the outgoing one. `tbsLoopRx2TxLVDS` loops back the receive serial LVDS links to the transmit serial LVDS and `tbsLoopTx2RxLVDS` loops back the transmit serial LVDS links to the receive serial LVDS.

PRBS Processor

Figure 11: PRBS Processor Model



The PRBS processors (generators and monitors) are present to perform device/system diagnostics. There are a total of two PRBS generators, one on the transmit side and one on the receive side. There are a total of four PRBS monitors, one on the transmit side and three on the receive side (i.e., working, protection and auxiliary). Individual PRBS generators or monitors are specified by one or two of the following arguments: `tdir` and `chnlType`.

Each port in the PRBS processor has to be configured separately to handle different traffic payloads. A PRBS processor can handle traffic from STS-1 to STS-48c. `tbsPayloadCfg` configures each port in the PRBS processor for different traffic payload configuration.

`tbsPrbsGenCfg` configures and controls the PRBS generator. The user may command the generator to operate in autonomous or TelecomBus mode, generate PRBS or sequential patterns, invert PRBS bytes prior to insertion, replace B1/E1 bytes, and access the LFSR. In addition, the current PRBS generator setting can also be retrieved using this function. Moreover, the user can also force bit errors in the generator using `tbsPrbsForceBitErr`.

Similarly, `tbsPrbsMonCfg` functions in a similar fashion for the PRBS monitor; it can be setup to operate in the autonomous or TelecomBus mode, invert PRBS byte for monitoring, monitor B1/E1 bytes, and monitor either PRBS or sequential patterns. In addition, the user may force the monitor to re-synchronize using `tbsPrbsResync`.

Error Insertion

The user can force a line-code violation (LCV) in the transmit side disparity encoders by calling `tbsTXDEForceLcv`.

In addition, the user can insert known test patterns into the following blocks for further diagnostics: disparity encoders and transmit TSIs. A user-defined test pattern can be inserted into the data stream via the disparity encoder using `tbsInsertTP`. For the transmit TSIs, `tbsInsIdleData` introduces a known data pattern (should be an 8B/10B character) into the data stream.

Finally, the user can also introduce a series of out-of-synchronization conditions in the 8B/10B decoders. `tbsForceOutOfChar` 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 errors may be introduced by invoking `tbsForceOutOfFrm`. The decoder will again attempt to resynchronize with the alignment character (K28.5).

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 TBS device driver uses to store initialization, configuration, and statistics information. For more information on our naming convention, the reader is referred to section 0.

4.1 Constants

The following Constants are used throughout the driver code:

- `<TBS 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.
- `TBS_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.
- `TBS_MOD_START`, `TBS_MOD_IDLE`, `TBS_MOD_READY`: the three possible module states (stored in `stateModule).`
- `TBS_START`, `TBS_PRESENT`, `TBS_ACTIVE`, `TBS_INACTIVE`: the four possible device states (stored in `stateDevice).`
- `TBS_NUM_TSIPAGE`: defines the number of connection memory pages present in a TSI block.
- `TBS_NUM_CHNLTYPE`: defines the number of channels (working, protection, …) on the egress side in the device.
- `TBS_NUM_TSLOTS`: the value of this constant minus one defines the number of time slots in one data stream.
- `TBS_NUM_TCBSTM`: the value of this constant minus one defines the number of data streams on both the ingress side and the egress side per channel.
- `eTBS_TSIMODE:` this is an enumerated type that defines all the TSI mapping modes.
- `eTBS_PTMODE:` this is an enumerated type that defines all the path termination modes.
- `eTBS_TRAFFICDIR:` this is an enumerated type that defines the traffic flow direction.
- `eTBS_CHNLTYPE:` this is an enumerated type that defines all the channel types (working, protection, and auxiliary).

4.2 Data Structures

The following are the main data structures used by the TBS driver. There are 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 arguments to functions within the driver. The structures are: the module Initialization Vector (MIV), the Device Initialization Vector (DIV), and the ISR mask.

Module Initialization Vector: MIV

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

- `perrModule` is a pointer to the errModule which indicates the last error encountered in an operation.
- `maxDevs` is used to inform the driver of 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 `TBS_MAX_DEVS` (see section 4.1).

Table 2: TBS Module Initialization Vector: sTBS_MIV

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

Device Initialization Vector: DIV

Passed via the `tbsInit` call, this structure contains all the information needed by the driver to initialize a TBS device. If a NULL pointer is supplied, the device will be left at the power up hardware reset state and no software initialization will take place. For a detailed description of the hardware reset state, please refer to the TBS Engineering Document (PMC-990522).

- `valid`: Reserved. Driver accesses this field. Do not write to it
- `pollISR` is a flag that indicates the type of interrupt servicing the driver is to use. The choices are 'polling' (`TBS_POLL_MODE`), and 'interrupt driven' (`TBS_ISR_MODE`). When configured in 'polling,' the Interrupt capability of the device is NOT used, and the user is responsible for calling devicePoll periodically. The actual processing of the event information is the same for both modes.

- `cbackIO, cbackTSI, cbackPRBS, cbackTXDE, and cbackRX8D` 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.
- `cfgDev` is the device configuration block.
- `intrmask` is the interrupt mask.
- `initPayloadCfg` is a Boolean variable indicating whether or not PRBS processor payload configuration will take place. If TRUE, the following PRBS payload configuration parameters will be used to configure all the PRBS generators and monitors. Otherwise, the following parameter blocks will be ignored.
  - `cfgIgPrbsGen [TBS_NUM_TCBSTM]` is the ingress PRBS generator's payload configuration blocks
  - `cfgIgPrbsMon [TBS_NUM_TCBSTM]` is the ingress PRBS monitor's payload configuration blocks.
  - `cfgEgPrbsGen [TBS_NUM_TCBSTM]` is the egress PRBS generator's payload configuration blocks.
  - `cfgEgPrbsMon [TBS_NUM_CHNLTYPE][TBS_NUM_TCBSTM]` is the egress PRBS monitor's payload configuration blocks.
- `initTSImap` is a Boolean variable indicating whether TSI connection maps will be initialized. If TRUE, all the TSI connection maps will be initialized with the following connection maps. Otherwise, there will not be TSI map initialization and the following connection maps are ignored.
- `txtiConMap[TBS_NUM_CHNLTYPE`] is the connection map for all the transmit TSIs
- `rxtiConMap [TBS_NUM_CHNLTYPE`] is the connection map for all the receive TSIs

Table 3: TBS Device Initialization Vector: sTBS_DIV

Field Name	Field Type	Field Description
`valid`	UINT2	Unused
`pollISR`	`TBS_POLL`	Indicates the type of ISR / polling to do
`cbackIO`	`sTBS_CBACK`	Address for the callback function for I/O events
`cbackTSI`	`sTBS_CBACK`	Address for the callback function for time slot interchange events
`cbackPRBS`	`sTBS_CBACK`	Address for the callback function for PRBS processor events

Field Name	Field Type	Field Description
`cbackTXDE`	`sTBS_CBACK`	Address for the callback function for disparity encoder events
`cbackRX8D`	`sTBS_CBACK`	Address for the callback function for 8B/10B decoder events
`cfgDev`	`sTBS_CFG_DEVICE`	Device configuration block
`intrmask`	`sTBS_MASK`	Interrupt mask
`initPayloadCfg`	BOOL	Payload configuration indicator. 0 = not configured, 1 = configured with given payload configuration blocks
`cfgIgPrbsGen [TBS_NUM_TCBSTM]`	`sTBS_CFG_PYLD`	Ingress(transmit) PRBS generator payload configuration block
`cfgIgPrbsMon [TBS_NUM_TCBSTM]`	`sTBS_CFG_PYLD`	Ingress(transmit) PRBS monitor payload configuration block
`cfgEgPrbsGen [TBS_NUM_TCBSTM]`	`sTBS_CFG_PYLD`	Egress(receive) PRBS generator payload configuration block
`cfgEgPrbsMon [TBS_NUM_CHNLTYPE] [TBS_NUM_TCBSTM]`	`sTBS_CFG_PYLD`	Egress(receive) PRBS monitor payload configuration block
`initTSImap`	BOOL	TSI connection map initialization indicator. 0 = not initialized, 1 = initialized with given map
`txtiConMap [TBS_NUM_CHNLTYPE]`	`sTBS_TSI_CONMAP`	Transmit TSI connection map
`rxtiConMap [TBS_NUM_CHNLTYPE]`	`sTBS_TSI_CONMAP`	Receive TSI connection map

Device Configuration Block: DEVICE

Used in the DIV for storing the device configuration.

Table 4: TBS Device Configuration Data Structure: sTBS_CFG_DEVICE

Field Name	Field Type	Field Description
iop	UINT1	Incoming odd parity bit inclusion, 0 = even, 1 = odd
incij0j1	UINT1	Incoming composite frame pulse bit inclusion, 0 = not included, 1 = included
incipl	UINT1	Incoming payload active bit inclusion, 0 = not included, 1 = included
oop	UINT1	Outgoing odd parity bit inclusion, 0 = even, 1 = odd
incoj0j1	UINT1	Outgoing composite frame pulse bit inclusion, 0 = not included, 1 = included
incopl	UINT1	Outgoing payload active bit inclusion, 0 = not included, 1 = included
rwsel_en	UINT1	Device pin rwsel enable, 0 = disable, 1 = enable
rj0dly	UINT2	J0 byte processing delay in cycles
rx8d_ofaais [TBS_NUM_CHNLTYPE] [TBS_NUM_TCBSTM]	UINT1	AIS insertion enable in Rx8D blocks, 0 = no insertion, 1 = insertion
id8e_tmode [TBS_NUM_TCBSTM] [TBS_NUM_TSLOTS]	eTBS_PTMODE	ID8E block termination mode for each time slot: 0 = MST, 1 = HPT
ip8e_tmode [TBS_NUM_TCBSTM] [TBS_NUM_TSLOTS]	eTBS_PTMODE	IP8E block termination mode for each time slot: 0 = MST, 1 = HPT
csu_ena	UINT1	CSU enable control bit. This bit is active low and a logic zero enables the CSU.

Field Name	Field Type	Field Description
`csu_rstb`	UINT1	CSU reset, holds this bit low for at least 100 microseconds for proper CSU reset. This bit provides a mean to force a CSU reset by the user.
`notepad`	UINT2	Software ID register that retains a user word even after a reset.
`dll_lock`	UINT1	Controls the DLL to track the phase offset between SYSCLK and REFCLK when this field is low, ignores the phase offset when set to high
`tsiOverwrite`	BOOL	TSI mapping mode: 0 = normal, 1 = overwrite

Payload Configuration Block: PYLD

This is used in the DIV and DDB to store the payload configuration for the PRBS generators and monitors.

Table 5: TBS Payload Configuration Block: sTBS_CFG_PYLD

Field Name	Field Type	Field Description
`sts12csl`	UINT1	Selects the slave payload configuration. 0 = all time slots are part of concatenated master payload, 1 = all time slots are part of a slave payload
`sts12c`	UINT1	Selects payload configuration. 0 = STS-1 paths are defined by sts3c bits, 1 = all time slots are part of the same concatenated payload defined by `msslen` bits.
`msslen`	UINT1	Selects payload configuration to be processed. 0x0 = STS-12c or below 0x1 = STS-24c 0x2 = STS-36c 0x3 = STS-48c

`sts3c`	UINT1	STS-3c payload configuration bits. The 4 LSBs are used (sts3c[3..0]).
		sts3c[x] = 0: STS-1 time slot #(1+x), #(5+x), #(9+x) are part of STS-3c payload
		sts3c[x] = 1: STS-1 time slot #(1+x), #(5+x), #(9+x) are independent STS-1 payload

TSI Connection Map: CONMAP

Used in the DIV and DDB for storing the TSI connection maps.

Table 6: TBS TSI Connection Map Data Structure: sTBS_TSI_CONMAP

Field Name	Field Type	Field Description
`pg [TBS_NUM_TSIPAGE]`	`sTBS_TSI_CONPAGE`	Connection maps consist of multiple connection pages

TSI Connection Page: CONPAGE

Used in the DIV and CONMAP for storing the TSI connection maps.

Table 7: TBS TSI Connection Page Data Structure: sTBS_TSI_CONPAGE

Field Name	Field Type	Field Description
`destSlot [TBS_NUM_TCBSTM] [TBS_NUM_TSLOTS]`	`sTBS_SPTSLOT`	Connection pages consist of slots mapping

TSI Space-Time Slot: SPTSLOT

Used in the DIV and CONPAGE to represent space-time slot mapping.

Table 8: TBS Space-time Slot Data Structure: sTBS_SPTSLOT

Field Name	Field Type	Field Description
`numPort`	UINT2	Port number
`numTS`	UINT2	Time slot number
`srcSel`	UINT1	Data source control (applicable only when dealing with TxTI blocks)

ISR Enable/Disable Mask

Passed via the `tbsSetMask`, `tbsGetMask` and `tbsClearMask` calls, this structure contains all the information needed by the driver to either enable or disable any of the interrupts in the TBS

Table 9: TBS ISR Mask: sTBS_MASK

Field Name	Field Type	Field Description
`io_ipe [TBS_NUM_TCBSTM]`	UINT2	Interrupt enable for incoming data parity error interrupts (0 = disable, 1 = enable)
`txti_coap [TBS_NUM_CHNLTYPE]`	UINT2	Interrupt enable for change in active connection page in transmit TSIs interrupts (0 = disable, 1 = enable)
`rxti_coap [TBS_NUM_CHNLTYPE]`	UINT2	Interrupt enable for change in active connection page in receive TSIs interrupts (0 = disable, 1 = enable)
`txde_fifoerr [TBS_NUM_CHNLTYPE][TBS_NUM_TCBSTM]`	UINT2	Interrupt enable for transmit disparity encoder FIFO error interrupts (0 = disable, 1 = enable)
`rx8d [TBS_NUM_CHNLTYPE]`	`sTBS_EVT_RX8D`	Interrupt enable for receive 8B/10B decoder block. Individual interrupt that corresponds to each event in the data structure is enabled/disabled separately. (0 = disable, 1 = enable)

Field Name	Field Type	Field Description
`itpp`	`sTBS_EVT_PRBS`	Interrupt enable for incoming PRBS monitor block. Individual interrupt that corresponds to each event in the data structure is enabled/disabled separately. (0 = disable, 1 = enable)
`rxpm [TBS_NUM_CHNLTYPE]`	`sTBS_EVT_PRBS`	Interrupt enable for receive PRBS monitor block. Individual interrupt that corresponds to each event in the data structure is enabled/disabled separately. (0 = disable, 1 = enable)
`csuLocke`	UINT2	Interrupt enable for CSU lock state change interrupt (0 = disable, 1 = enable)
`dllError`	UINT2	Interrupt enable for DLL block (0 = disable, 1 = enable)

8B/10B Decoder Events: EVT_RX8D

This structure encompasses all events in a receive 8B/10B decoder block. All the fields can be interpreted in two ways. In the context of interrupt masking, the fields set/clear the corresponding hardware interrupt bits. In the context of event counting, the fields serve as occurrence counters and threshold levels for a particular event/interrupt. Data field `lcvCt[]` is the only exception. Since there are no hardware interrupts associated with line code violation counters. This field is not applicable in interrupt masking.

Table 10: TBS 8B/10B Decoder Event: sTBS_EVT_RX8D

Field Name	Field Type	Field Description
`oca [TBS_NUM_TCBSTM]`	UINT4	Out of character alignment interrupt per port.
`ofa [TBS_NUM_TCBSTM]`	UINT4	Out of frame alignment interrupt per port.
`fuo [TBS_NUM_TCBSTM]`	UINT4	FIFO underrun/overrun interrupt per port
`lcv [TBS_NUM_TCBSTM]`	UINT4	Line code violation interrupt per port.

Field Name	Field Type	Field Description
`lcvCt [TBS_NUM_TCBSTM]`	UINT4	Line code violation counter per port. Not defined in the context of interrupt mask. Only defined in event counting.

PRBS Processor Events: EVT_PRBS

This structure includes all the events in a PRBS monitor block. All data fields can be interpreted in two ways. In the context of interrupt masking, the fields set/clear the corresponding hardware interrupt bits. In the context of event counting, the fields serve as occurrence counters and threshold levels for a particular event/interrupt. Data field byteErrCt[][] is the only exception, since there are no hardware interrupts associated with line code violation counters. This field is not applicable in interrupt masking.

Table 11: TBS PRBS Monitor Event: sTBS_EVT_PRBS

Field Name	Field Type	Field Description
`monerr [TBS_NUM_TCBSTM] [TBS_NUM_TSLOTS]`	UINT4	Monitor byte error interrupt per slot
`monb1e1 [TBS_NUM_TCBSTM] [TBS_NUM_TSLOTS]`	UINT4	Monitors B1/E1 byte mismatch interrupt per slot
`monsync [TBS_NUM_TCBSTM] [TBS_NUM_TSLOTS]`	UINT4	Monitors synchronization state change interrupt per slot
`byteErrCt [TBS_NUM_TCBSTM] [TBS_NUM_TSLOTS]`	UINT4	Monitors byte error counter detected in the block defined in event counting, but not applicable in the context of interrupt masking.

Structures in the Driver's Allocated Memory

These structures are defined and used by the driver; they are part of the context memory allocated when the driver is opened. The structures are: the Module Data Block (MDB), and the Device Data Block (DDB).

Module Data Block: MDB

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

- `errModule` – Most of the module API functions return a specific error code directly. When the returned code is `TBS_FAILURE`, this indicates that the top-level function was not able to carry the specified error code back to the application. Under those circumstances, the proper error code is recorded in this element. The element is the first in the structure so that the user can cast the MDB pointer into a INT4 pointer and retrieve the local error (this eliminates the need to include the MDB template into the application code).
- `valid` indicates that this structure has been properly initialized and may be read by the user.
- `stateModule` contains the current state of the module and could be set to: `TBS_MOD_START`, `TBS_MOD_IDLE` or `TBS_MOD_READY`.
- `maxDev` indicates the maximum number of devices supported by the driver
- `numDevs` indicates the number of devices currently registered in the driver
- `pddb` is a pointer to an array of device data blocks (DDB) inside the driver.

Table 12: TBS Module Data Block: sTBS_MDB

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

Device Data Block: DDB

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

- `errDevice -` Most of the device API functions return a specific error code directly. When the returned code is `TBS_FAILURE`, this indicates that the top-level function was not able to carry the specific error code back to the application. In addition, some device functions do not return an error code. Under those circumstances, the proper error code is recorded in this element. The element is the first in the structure so that the user can cast the DDB pointer to a INT4 pointer and retrieve the local error (this eliminates the need to include the DDB template in the application code).
- `valid` indicates that this structure has been properly initialized and may be read by the user.

- `stateDevice` contains the current state of the Device and could be set to: `TBS_START`, `TBS_PRESENT`, `TBS_ACTIVE` or `TBS_INACTIVE`.
- `usrCtxt` is a value that can be used by the user to identify the device during the execution of the callback functions. It is passed to the driver when `tbsAdd` 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.
- For the rest of the members inside the structure, please refer to their respective description fields inside the table.

Table 13: TBS Device Data Block: sTBS_DDB

Field Name	**Field Type**	**Field Description**
`errDevice`	INT4	Global error indicator for device calls
`valid`	UINT2	Indicates that this structure has been initialized
`stateDevice`	`eTBS_DEV_STATE`	Device State can be one of the following PRESENT, ACTIVE or INACTIVE
`baseAddr`	void*	Base address of the Device
`usrCtxt`	`sTBS_USR_CTXT`	Stores the user's context for the device. It is passed as an input parameter when the driver invokes an application callback
`pollISR`	`sTBS_POLL`	Indicates the current type of ISR / polling
`cbackIO`	`sTBS_CBACK`	Address for the callback function for I/O events
`cbackTSI`	`sTBS_CBACK`	Address for the callback function for time slot interchange events
`cbackPRBS`	`sTBS_CBACK`	Address for the callback function for PRBS processor events
`cbackTXDE`	`sTBS_CBACK`	Address for the callback function for disparity encoder events
`cbackRX8D`	`sTBS_CBACK`	Address for the callback function for 8B/10B decoder events

Field Name	Field Type	Field Description
div	sTBS_DIV	DIV copy given at initialization
txtiMap [TBS_NUM_CHNLTYPE]	sTBS_TSI_CONMAP	Mirror copy of the transmit TSIs connection map residing in the driver memory
rxtiMap [TBS_NUM_CHNLTYPE]	sTBS_TSI_CONMAP	Mirror copy of the receive TSIs connection map residing in the driver memory
muxSemStat	void*	Pointer to mutual exclusion semaphore for event statistics block protection
errStat	INT4	Error indicator for statistics retrieval
intrmask	sTBS_MASK	Current interrupt mask
evtCntr	sTBS_STAT	Device event statistics block
tsiOverwrite	BOOL	TSI mapping mode: 0 = normal, 1 = overwrite

Statistics Block: STAT

This is the top level structure for statistics.

Table 14: TBS Event Statistics Block: sTBS_STAT

Field Name	Field Type	Field Description
actual	sTBS_CNTR	Cumulative count of all the events
delta	sTBS_CNTR	Delta count of all the events
thresh	sTBS_CNTR	Threshold setting of all the events
threshCtr	sTBS_CNTR	Threshold counter for all events (internal use)

Event/Statistics Counter Structure: CNTR

This structure contains the counter for all TBS events and statistics.

Table 15: TBS Events Counter Block: sTBS_CNTR

Field Name	Field Type	Field Description
`sysclka`	UINT4	System clock inactivity event (an I/O event)
`io_ipe [TBS_NUM_TCBSTM]`	UINT4	Interrupt event for data parity error for each incoming TelecomBus byte stream (an I/O event)
`csuLocke`	UINT4	Interrupt event for the CSU lock state change (an I/O event)
`dllError`	UINT4	Interrupt event for the DLL block error condition
`itca [TBS_NUM_TCBSTM]`	UINT4	Tributary control inactivity for each incoming TelecomBus byte stream
`ipca [TBS_NUM_TCBSTM]`	UINT4	High order path control inactivity for each incoming TelecomBus byte stream
`ida [TBS_NUM_TCBSTM]`	UINT4	Data bus inactivity for each incoming TelecomBus byte stream
`txti_coap [TBS_NUM_CHNLTYPE]`	UINT4	Interrupt event for change in active connection page for each transmit TSI block (a TSI event)
`rxti_coap [TBS_NUM_CHNLTYPE]`	UINT4	Interrupt event for change in active memory page for each receive TSI block (a TSI event)
`txde_fifoerr [TBS_NUM_CHNLTYPE] [TBS_NUM_TCBSTM]`	UINT4	FIFO error interrupt event for each transmit disparity encoder (a TXDE event)
`rx8d [TBS_NUM_CHNLTYPE]`	`sTBS_EVT_RX8D`	Events for each receive 8B/10B decoder block (a RX8D event)
`itpp`	`sTBS_EVT_PRBS`	Events for the transmit PRBS monitor (a PRBS event)
`rxpm [TBS_NUM_CHNLTYPE]`	`sTBS_EVT_PRBS`	Events for the receive PRBS monitor (a PRBS event)

Device Status Block: STATUS

This structure encompasses all the relevant status of the device for instant read back.

Table 16: TBS Device Status Block : sTBS_STATUS

Field Name	Field Type	Field Description
`io`	`sTBS_STATUS_ IO`	Current status of the I/O block
`rx8d [TBS_NUM_CHNLTYPE]`	`sTBS_STATUS_ RX8D`	Current status of the receive 8B/10B decoders
`rxpm [TBS_NUM_CHNLTYPE]`	`sTBS_STATUS_ PRBS`	Current status of the receive PRBS monitor blocks
`itpp`	`sTBS_STATUS_ PRBS`	Current status of the incoming PRBS monitor block
`csuLockv`	UINT2	Current status of the CSU reference clock lock. 0 = not locked, 1 = locked

I/O Status Block: STATUS_IO

This sub-structure contains the I/O block status.

Table 17: TBS I/O Block Status: sTBS_STATUS _IO

Field Name	Field Type	Field Description
`sysclka`	UINT1	System clock activity (0 = inactive, 1 = active)
`itca [TBS_NUM_TCBSTM]`	UINT1	Tributary control active bit for each incoming TelecomBus stream (0 = inactive, 1 = active)
`ipca [TBS_NUM_TCBSTM]`	UINT1	High order path control active bit for each incoming TelecomBus stream (0 = inactive, 1 = active)
`ida [TBS_NUM_TCBSTM]`	UINT1	Data bus active bit for each incoming TelecomBus stream (0 = inactive, 1 = active)
`dll_run`	UINT1	DLL block lock status. If the phase detector has a lock, it reads a logic high; otherwise, it reads a logic low.

Field Name	Field Type	Field Description
dll_error	UINT1	DLL error indicator. It normally reads a logic low unless the DLL runs out of dynamic range.

8B/10B Decoder Status Block: STATUS_RX8D

This sub-structure contains the 8B/10B decoder status.

Table 18: TBS 8B/10B Decoder Block Status: sTBS_STATUS _RX8D

Field Name	Field Type	Field Description
ocav [TBS_NUM_TCBSTM]	UINT1	Current out-of-character alignment status, 0 = aligned, 1 = not aligned
ofav [TBS_NUM_TCBSTM]	UINT1	Current out-of-frame alignment status, 0 = aligned, 1 = not aligned

PRBS Monitor Status Block: STATUS_PRBS

This sub-structure encompasses the PRBS monitor's status.

Table 19: TBS PRBS Monitor Status: sTBS_STATUS _PRBS

Field Name	Field Type	Field Description
monsyncv [TBS_NUM_TCBSTM] [TBS_NUM_TSLOTS]	UINT1	Current monitor synchronization status, 0 = unsynchronized, 1 = synchronized
prbs_lfsr [TBS_NUM_TCBSTM] [TBS_NUM_TSLOTS]	UINT4	Current PRBS linear feedback shifted register (LFSR) content
rec_b1 [TBS_NUM_TCBSTM] [TBS_NUM_TSLOTS]	UINT1	Received B1 bytes
rec_e1 [TBS_NUM_TCBSTM] [TBS_NUM_TSLOTS]	UINT1	Received E1 bytes

PRBS Generator/Monitor Configuration Block: CFG_PRBS PORT

This sub-structure encompasses the PRBS generator/monitor configuration block.

Table 20: TBS PRBS Generator/Monitor Configuration Block Per Port: sTBS_CFG_PRBS PORT

Field Name	Field Type	Field Description
`sts1PathCfg [TBS_NUM_TSLOTS]`	`sTBS_CFG_PRBS`	STS-1 path configuration parameter per time slot

PRBS Generator/Monitor STS-1 Configuration Parameters: CFG_PRBS

This sub-structure encompasses the PRBS generator/monitor STS-1 path configuration parameters.

Table 21: TBS PRBS Generator/Monitor Configuration Parameters Per Time Slot : sTBS_CFG_PRBS

Field Name	Field Type	Field Description
`amode`	UINT1	TelecomBus Mode, 0 = TelecomBus mode, 1 = autonomous mode
`inv_prbs`	UINT1	PRBS inversion. 0 = unmodified, 1 = inverted
`b1e1_enb`	UINT1	B1/E1 byte replacement for generator and monitoring for monitor. 0 = inactive, 1 = active
`gpo`	UINT1	General purpose output, only used in OTPG (OCOUT bit)
`seq_prbs`	UINT1	Pattern selection. 0 = PRBS, 1 = sequential
`prbs_lfsr`	UINT4	Linear feedback shift register (LFSR) for PRBS
`b1`	UINT1	B1 byte to be inserted for generator and to be monitored for monitor
`s`	UINT1	S bit value to be inserted to H1 byte bit 2 & 3, only used in autonomous mode and when processing concatenated payload (for generator use only)
`prbs_ena`	UINT1	Enable/disable PRBS pattern insertion in generator and monitoring in monitor. 0 = disable, 1 = enable This field provides additional control for the PRBS generator (ITPP) on the transmit side. 0 = disable, 1 = Prbs data sent to ID8E, 2 = Prbs data sent to IP8E, 3 = Prbs data sent to both ID8E and IP8E blocks

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 `sysTbsBufferStart` function. An individual ISV buffer is then obtained by the driver via `sysTbsISVBufferGet` and returned to the 'pool' via `sysTbsISVBufferRtn`.

Table 22: TBS Interrupt Service Vector: sTBS_ISV

Field Name	Field Type	Field Description
`deviceHandle`	`sTBS_HNDL`	Handle to the device in cause
`parityErrIntrStat`	UINT2	Master accumulation transfer and parity error interrupt status
`masterIntrStat_1`	UINT2	Master interrupt status#1
`masterIntrStat_2`	UINT2	Master interrupt status#2
`masterIntrStat_3`	UINT2	Master interrupt status#3
`masterIntrStat_4`	UINT2	Master interrupt status#4
`tsiIntrStat`	UINT2	Master TSI interrupt status

Deferred Processing Vector: DPV

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

The DPR divides the TBS into 5 sections: IO, TSI, PRBS, TXDE, and RX8D. Five user-supplied callback routines (one per section) are used to inform the application which section of the device has caused the event being reported. The size of this buffer should be kept as short as possible.

Table 23: TBS Deferred Processing Vector: sTBS_DPV

Field Name	Field Type	Field Description

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

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 it.

`tbsMdb`: This is 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 `TBS_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 Block s. The user is cautioned that a DDB is only valid if the `valid` flag is set. Note that the array of DDBs is in no particular order.
    - `errDevice:` This structure element is used to store an error code that specifies the reason for an API function's failure. The field is only valid for functions that do not return an error code or when a value of `TBS_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 TBS 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, as well as 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 24. For a typical module management flow diagram see page 26.

Opening the Driver Module: tbsModuleOpen

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 tbsModuleOpen( sTBS_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 = `TBS_SUCCESS`
Failure = `TBS_ERR_MODULE_ALREADY_OPEN`
`TBS_ERR_INVALID_MIV`
`TBS_ERR_MEM_ALLOC`

Valid States `TBS_MOD_START`

Side Effects Changes the MODULE state to `TBS_MOD_IDLE`

Closing the Driver Module: tbsModuleClose

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

Prototype `INT4 tbsModuleClose( void)`

Inputs None

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_MODULE_NOT_OPEN`
`TBS_FAILURE`

Valid States ALL STATES

Side Effects Changes the MODULE state to `TBS_MOD_START`

Starting the Driver Module: tbsModuleStart

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 tbsModuleStart(void)`

Inputs None

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_MODULE_NOT_OPEN`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_MEM_ALLOC`
`TBS_ERR_INT_INSTALL`
`TBS_FAILURE`

Valid States `TBS_MOD_IDLE`

Side Effects Changes the MODULE state to `TBS_MOD_READY`

Stopping the Driver Module: tbsModuleStop

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, `tbsDelete` is called for each.

Prototype `INT4 tbsModuleStop( void)`

Inputs None

Outputs None

Returns Success = `TBS_SUCCESS`

	Failure = `TBS_ERR_MODULE_NOT_OPEN`
	`TBS_ERR_INVALID_STATE`
	`TBS_FAILURE`

Valid States	`TBS_MOD_READY`
Side Effects	Changes the MODULE state to `TBS_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's general activity, as well as enabling interrupt processing for that device. These functions are also used to change the software state for that device. For more information on the device states, see the state diagram on page 24. For a typical device management flow diagram, see page 27.

Adding a Device: tbsAdd

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

Prototype	`sTBS_HNDL tbsAdd(void *usrCtxt, void *baseAddr, INT4 **pperrDevice)`
Inputs	`usrCtxt` : user context for this device `baseAddr` : base address of the device `pperrDevice` : (pointer to) an area of memory

Outputs	`pperrDevice` : (pointer to) `errDevice` (inside the DDB)
	ERROR code written to the MDB on failure
	`TBS_ERR_INVALID_STATE`
	`TBS_ERR_DEVS_FULL`
	`TBS_ERR_DEV_ALREADY_ADDED`
	`TBS_ERR_INVALID_TYPE_ID`
	`TBS_FAILURE`

Returns	device handle
Valid States	`TBS_MOD_READY`
Side Effects	Changes the DEVICE state to `TBS_PRESENT`

Deleting a Device: tbsDelete

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

Prototype	`INT4 tbsDelete(sTBS_HNDL deviceHandle)`	
Inputs	`deviceHandle`	: device Handle (from `tbsAdd`)
Outputs	None	

Returns	Success = `TBS_SUCCESS`
	Failure = `TBS_ERR_INVALID_DEV`
	`TBS_FAILURE`	

Valid States	`TBS_PRESENT, TBS_ACTIVE, TBS_INACTIVE`	
Side Effects	Changes the device state to `TBS_PRESENT`	

Initializing a Device: tbsInit

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

Prototype	`INT4 tbsInit(sTBS_HNDL deviceHandle, sTBS_DIV *pdiv, UINT2 profileNum)`	
Inputs	`deviceHandle`	: device Handle (from `tbsAdd`)
	`pdiv`	: (pointer to) Device Initialization Vector
	`profileNum`	: profile number (not supported for this device)
Outputs	None	

Returns	Success = `TBS_SUCCESS`
	Failure = `TBS_ERR_INVALID_DEV`
	`TBS_ERR_INVALID_STATE`
	`TBS_ERR_INVALID_ARG`
	`TBS_ERR_CONNECT_EXIST`
	`TBS_ERR_POLL_TIMEOUT`	

Valid States	`TBS_PRESENT`	
Side Effects	Changes the DEVICE state to `TBS_INACTIVE`	

Updating the Configuration of a Device: tbsUpdate

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 `tbsUpdate` and `tbsInit` is that no soft reset will be applied to the device.

Prototype `INT4 tbsUpdate(sTBS_HNDL deviceHandle, sTBS_DIV *pdiv, UINT2 profileNum)`

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`pdiv`	: (pointer to) Device Initialization Vector
`profileNum`	: profile number (not supported for this device)

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Resetting a Device: tbsReset

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

Prototype `INT4 tbsReset(sTBS HNDL deviceHandle)`

Inputs

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

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`

Valid States `TBS_PRESENT, TBS_ACTIVE, TBS_INACTIVE`

Side Effects Changes the DEVICE state to `TBS_PRESENT`

Activating a Device: tbsActivate

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

Prototype `INT4 tbsActivate(sTBS_HNDL deviceHandle)`

Inputs	`deviceHandle`	: device Handle (from `tbsAdd`)
Outputs	None	

Returns	Success = `TBS_SUCCESS`
	Failure = `TBS_ERR_INVALID_DEV`
	`TBS_ERR_INVALID_STATE`	

Valid States	`TBS_INACTIVE`	
Side Effects	Changes the DEVICE state to `TBS_ACTIVE`	

De-Activating a Device: tbsDeActivate

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

Prototype	`INT4 tbsDeActivate(sTBS_HNDL deviceHandle)`	
Inputs	`deviceHandle`	: device Handle (from `tbsAdd`)
Outputs	None	

Returns	Success = `TBS_SUCCESS`
	Failure = `TBS_ERR_INVALID_DEV`
	`TBS_ERR_INVALID_STATE`	

Valid States	`TBS_ACTIVE`	
Side Effects	Changes the DEVICE state to `TBS_INACTIVE`	

5.3 Device Read and Write

Reading from Device Registers: tbsRead

This function can be used to read a register of a specific TBS device by providing the register number. The 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, `sysTbsRead`. Note that a failure to read returns a zero and that any error indication is written to the associated DDB.

Prototype	`UINT2 tbsRead( sTBS_HNDL deviceHandle, UINT2 regNum)`	
Inputs	`deviceHandle` `regNum`	: device Handle (from `tbsAdd`) : register number

Outputs	ERROR code written to the MDB: `TBS_ERR_INVALID_DEV` ERROR code written to the DDB: `TBS_ERR_INVALID_REG`

Returns	Success = value read
	Failure = 0

Valid States	`TBS_PRESENT, TBS_ACTIVE, TBS_INACTIVE`
Side Effects	May affect registers that change after a read operation

Writing to Device Registers: tbsWrite

This function can be used to write to a register of a specific TBS device by providing the register number. The function derives the actual address location based on the device handle and register number inputs. It then writes the contents of this address location using the system specific macro, `sysTbsWrite`. Note that a failure to write returns a zero and any error indication is written to the DDB.

Prototype

```
UINT2 tbsWrite( sTBS_HNDL deviceHandle, UINT2 regNum,
UINT2 value)
```

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`regNum`	: register number
`value`	: value to be written

Outputs	ERROR code written to the MDB: `TBS_ERR_INVALID_DEV` ERROR code written to the DDB: `TBS_ERR_INVALID_REG`

Returns	Success = value written
	Failure = 0

Valid States	`TBS_PRESENT, TBS_ACTIVE, TBS_INACTIVE`
Side Effects	May change the configuration of the Device

Reading from a block of Device Registers: tbsReadBlock

This function can be used to read a register block of a specific TBS device by giving it the starting register number and the size to read. The function derives the actual start address location based on the device handle and starting register number inputs. It then reads the contents of this data block using multiple calls to the system specific macro, `sysTbsRead`. Note that a failure to read returns a zero and any error indication is written to the DDB. It is the user's responsibility to allocate enough memory for the block read.

Prototype `UINT2 tbsReadBlock( sTBS_HNDL deviceHandle, UINT2 startRegNum, UINT2 size, UINT2 *pblock)`

Inputs

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

Outputs ERROR code written to the MDB
`TBS_ERR_INVALID_DEV`
ERROR code written to the DDB
`TBS_ERR_INVALID_ARG`
`TBS_ERR_INVALID_REG`
pblock : (pointer to) the block read

Returns Success = Last register value read
Failure = 0

Valid States `TBS_PRESENT, TBS_ACTIVE, TBS_INACTIVE`

Side Effects May affect registers that change after a read operation

Writing to a Block of Device Registers: tbsWriteBlock

This function can be used to write to a register block of a specific TBS device by giving it the starting register number and the block size. The function derives the actual starting address location based on the device handle and starting register number inputs. It then writes the contents of this data block using multiple calls to the system specific macro, `sysTbsWrite`. A bit from the passed block is only modified in the device's registers if the corresponding bit is set in the passed mask. Note that any error indication is written to the DDB

Prototype `UINT2 tbsWriteBlock( sTBS_HNDL deviceHandle, UINT2 startRegNum, UINT2 size, UINT2 *pblock, UINT2 *pmask)`

Inputs

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

Outputs ERROR code written to the MDB
`TBS_ERR_INVALID_DEV`
ERROR code written to the DDB
`TBS_ERR_INVALID_ARG`
`TBS_ERR_INVALID_REG`

Returns Success = Last register value written
Failure = 0

Valid States	`TBS_PRESENT, TBS_ACTIVE, TBS_INACTIVE`
Side Effects	May change the configuration of the Device

Indirect reading from Device Registers: tbsReadIndirect

This function can be used to perform an indirect read from am indirect register in the TBS device by giving it the register location and the indirect address to be read from. The 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, `sysTbsRead`. Note that a failure to read returns a zero and any error indication is written to the DDB.

Prototype

```
UINT2 tbsReadIndirect( sTBS_HNDL deviceHandle, UINT2
iaddrReg, UINT2 idataReg, UINT2 iaddr, UINT2 *pData)
```

Inputs

`deviceHandle`	: device Handle (from tbsAdd)
`iaddrReg`	: indirect address register number
`idataReg`	: indirect data register number
`indirAddr`	: indirect address to read
`pData`	: (pointer to) the data to read

Outputs

ERROR code written to the MDB
`TBS_ERR_INVALID_DEV`
ERROR code written to the DDB
`TBS_ERR_INVALID_REG`
`TBS_ERR_POLL_TIMEOUT`
`pData` : (pointer to) the block read

Returns

Success = value read
Failure = 0

Valid States	`TBS_PRESENT, TBS_ACTIVE, TBS_INACTIVE`
Side Effects	May affect registers that change after a read operation

Indirect writing to Device Registers: tbsWriteIndirect

This function can be used to perform an indirect write to an indirect access register in the TBS device by giving it the register location and the indirect address to be written to. The 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, `sysTbsWrite.` Note that a failure to write returns a zero and that any error indication is written to the DDB.

Prototype

```
UINT2 tbsWriteIndirect( sTBS_HNDL deviceHandle, UINT2
iaddrReg, UINT2 idataReg, UINT2 iaddr, UINT2 Data)
```

Inputs

`deviceHandle`	: device Handle (from tbsAdd)
`iaddrReg`	: indirect address register number

	`idataReg`	: indirect data register number
	`iaddr`	: indirect address to read
	`data`	: new data

Outputs	ERROR code written to the MDB
	`TBS_ERR_INVALID_DEV`
	ERROR code written to the DDB
	`TBS_ERR_INVALID_REG`
	`TBS_ERR_POLL_TIMEOUT`	

Returns	Success = value written
	Failure = 0	

Valid States	`TBS_PRESENT, TBS_ACTIVE, TBS_INACTIVE`	
Side Effects	May affect registers that change after a read operation	

5.4 Device Configuration

The following functions control the dynamic configuration of the device.

Setting Device Configuration Block: tbsDeviceSetConfig

This function sets the device configuration dynamically through the `sTBS_CFG_DEVICE` data structure.

Prototype	`INT4 tbsDeviceSetConfig( sTBS_HNDL deviceHandle, sTBS_CFG_DEVICE *pCfgParam)`	
Inputs	`deviceHandle`	: device Handle (from tbs`Add`)
	`pCfgParam`	: device configuration block
Outputs	None	

Returns	Success = `TBS_SUCCESS`
	Failure = `TBS_ERR_INVALID_DEV`
	`TBS_ERR_INVALID_STATE`
	`TBS_ERR_INVALID_ARG`	

Valid States	`TBS_ACTIVE, TBS_INACTIVE`	
Side Effects	May affect data flow	

Getting Device Configuration Block: tbsDeviceGetConfig

This function gets the current device configuration.

Prototype `INT4 tbsDeviceGetConfig( sTBS_HNDL deviceHandle, sTBS_CFG_DEVICE *pCfgParam)`

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`pCfgParam`	: device configuration block

Outputs

`pCfgParam`	: device configuration block

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_INVALID_ARG`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

5.5 Time Slot Interchange

The main objective of a time slot interchange is the remapping of time slots. Functions are provided to connect and disconnect calls. There are also functions for housekeeping, such as altering mapping mode, verifying unicast or multicast calls, checking mapping patterns, etc.

Setting global TSI mapping mode: tbsSetMapMode

Set the global mapping mode of all the TSIs in the device (user-defined or bypass). Bypass mode puts the chip in a through mode and user-defined mode activates all the TSIs inside the device.

Prototype `INT4 tbsSetMapMode( sTBS_HNDL deviceHandle, eTBS_TRAFFICDIR tdir, eTBS_TSIMODE mode)`

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`tdir`	: traffic direction
`mode`	: mapping mode

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_INVALID_ARG`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Getting global TSI mapping mode: tbsGetMapMode

Get the current global mapping mode of all the TSIs in the device (user-defined or bypass).

Prototype
```
INT4 tbsGetMapMode( sTBS_HNDL deviceHandle,
eTBS_TRAFFICDIR tdir, eTBS_TSIMODE *pMode)
```

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`tdir`	: traffic direction
`pMode`	: current mapping mode

Outputs pMode : current mapping mode

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_INVALID_ARG`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Setting active connection page in TSI: tbsSetPage

In conjunction with the external hardware pin xCMP (TCMP on the transmit side and OCMP on the receive side), this function selects the active connection memory page in the TSI.

Prototype
```
INT4 tbsSetPage(sTBS_HNDL deviceHandle,
eTBS_TRAFFICDIR tdir, eTBS_CHNLTYPE chnlType, UINT2
pgNum)
```

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`tdir`	: traffic direction
`chnlType`	: channel type
`pgNum`	: memory page number

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_INVALID_ARG`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Getting active connection page in TSI: tbsGetPage

Obtain the current active connection memory page of the TSI.

Prototype	`INT4 tbsGetPage(sTBS_HNDL deviceHandle, eTBS_TRAFFICDIR tdir, eTBS_CHNLTYPE chnlType, UINT2 *pPgNum)`	
Inputs	`deviceHandle`	: device Handle (from `tbsAdd`)
	`tdir`	: traffic direction
	`chnlType`	: channel type
Outputs	`pPgNum`	: pointer to active memory page number
Returns	Success = `TBS_SUCCESS`	
	Failure = `TBS_ERR_INVALID_DEV`	
	`TBS_ERR_INVALID_STATE`	
	`TBS_ERR_INVALID_ARG`	
Valid States	`TBS_ACTIVE, TBS_INACTIVE`	
Side Effects	None	

Mapping the source to destination slot(s) in TSI: tbsMapSlot

Map the source slot to destination slot(s). If the parameter `numSlots` is greater than one, the function maps the given `srcSlot` to a group of `destSlot`. An error code will be returned if there is an attempt to map into an occupied destination space-time slot.

Prototype	`INT4 tbsMapSlot( sTBS_HNDL deviceHandle, eTBS_TRAFFICDIR tdir, eTBS_CHNLTYPE chnlType, UINT2 pgNum, sTBS_SPTSLOT *psrcSlot, sTBS_SPTSLOT destSlot[], UINT4 numSlots)`	
Inputs	`deviceHandle`	: device Handle (from tbsAdd)
	`tdir`	: traffic direction
	`chnlType`	: channel type
	`pgNum`	: connection page number to be updated
	`psrcSlot`	: pointer to source space-time slot
	`destSlot[]`	: array of destination space-time slot(s)
	`numSlots`	: number of destination slot(s) to be mapped
Outputs	None	
Returns	Success = `TBS_SUCCESS`	
	Failure = `TBS_ERR_INVALID_DEV`	
	`TBS_ERR_INVALID_STATE`	
	`TBS_ERR_INVALID_ARG`	

`TBS_ERR_CONNECT_EXIST`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Retrieving source space-time Slot in TSI: tbsGetSrcSlot

Retrieves the source space-time slot for a given destination slot. A NULL slot will be returned if there is no match (A NULL slot has port and time slot numbers equal to zero).

Prototype

```
INT4 tbsGetSrcSlot(sTBS_HNDL deviceHandle,
eTBS_TRAFFICDIR tdir, eTBS_CHNLTYPE chnlType, UINT2
pgNum, sTBS_SPTSLOT *pdestSlot, sTBS_SPTSLOT
*psrcSlot)
```

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`tdir`	: traffic direction
`chnlType`	: channel type
`pgNum`	: connection page number to be updated
`pdestSlot`	: pointer to destination space-time slot(s)

Outputs

`psrcSlot`	: pointer to source space-time slot

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_INVALID_ARG`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Retrieving destination space-time Slot in TSI: tbsGetDestSlot

Retrieves the destination space-time slot for a given source space-time slot. If the number of destinations is less than or equal to one, the connection is unicast; otherwise, it is multicast. A NULL slot, along with *`pNumSlot` = 0, will be returned if there is no match. (A NULL slot has port and time slot numbers equal to zero)

Prototype

```
INT4 tbsGetDestSlot(sTBS_HNDL deviceHandle,
eTBS_TRAFFICDIR tdir, eTBS_CHNLTYPE chnlType, UINT2
pgNum, sTBS_SPTSLOT *psrcSlot, sTBS_SPTSLOT
destSlot[], UINT4 *pNumSlot)
```

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`tdir`	: traffic direction
`chnlType`	: channel type
`pgNum`	: connection page number to be updated

	`psrcSlot`	: pointer to source space-time slot(s)
Outputs	`destSlot[]`	: array of destination space-time slots
	`pNumSlot`	: pointer to number of destination time slots

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_INVALID_ARG`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Copying connection map from one page to another in TSI: tbsCopyPage

Copy the connection memory map from one page to another in the TSI. This also applies to page copying across different TSIs in the same traffic direction. For instance, page 0 of the working TSI can be copied to page 1 of the protection TSI using this function.

Prototype
```
INT4 tbsCopyPage(sTBS_HNDL deviceHandle,
eTBS_TRAFFICDIR tdir, eTBS_CHNLTYPE srcChnlType,
UINT2 srcPage, eTBS_CHNLTYPE destChnlType, UINT2
destPage)
```

Inputs	`deviceHandle`	: device Handle (from `tbsAdd`)
	`tdir`	: traffic direction
	`srcChnlType`	: source channel type
	`srcPage`	: source connection page number
	`destChnlType`	: destination channel type
	`destPage`	: destination connection page number

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_INVALID_ARG`
`TBS_FAILURE`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Inserting Idle Data in TSI: tbsInsIdleData

Insert an arbitrary idle data pattern into a given destination space-time slot. The pattern should be a valid 8b/10b character in most cases. This function is valid only for transmit the TSIs.

Prototype `INT4 tbsInsIdleData(sTBS_HNDL deviceHandle, eTBS_TRAFFICDIR tdir, eTBS_CHNLTYPE chnlType, UINT2 pgNum, sTBS_SPTSLOT *pdestSlot, BOOL insert, UINT2 idleDat)`

Inputs

`deviceHandle`	: device Handle (from tbsAdd)
`tdir`	: traffic direction
`chnlType`	: channel type
`pgNum`	: connection page to be updated
`pdestSlot`	: pointer to destination space-time slot
`insert`	: idle data insertion control. 0 = disable, 1 = enable
`idleDat`	: idle data pattern. valid range: (0-1023)

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_INVALID_ARG`
`TBS_ERR_POLL_TIMEOUT`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Removing established connection in TSI: tbsRmSlot

Disconnect the established connection between the given source and destination space-time slot(s).

Prototype `INT4 tbsRmSlot(sTBS_HNDL deviceHandle, eTBS_TRAFFICDIR tdir, eTBS_CHNLTYPE chnlType, UINT2 pgNum, sTBS_SPTSLOT *psrcSlot, sTBS_SPTSLOT destSlot[], UINT2 numSlot)`

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`tdir`	: traffic direction
`chnlType`	: channel type
`pgNum`	: connection page to be updated
`psrcSlot`	: pointer to source space-time slot
`destSlot[]`	: array of destination space-time slot(s)
`numSlot`	: number of destination space-time slot(s) to be removed

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`

```
TBS_ERR_INVALID_STATE
TBS_ERR_INVALID_ARG
TBS_ERR_CONNECT_NONEXISTENT
TBS_FAILURE
```

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Clearing all established connections in TSI: tbsClrSlot

Disconnect all established connection(s) for the given source space-time slot. This function is very useful in disconnecting multicast calls for a given source space-time slot.

Prototype

```
INT4 tbsClrSlot(sTBS_HNDL deviceHandle,
eTBS_TRAFFICDIR tdir, eTBS_CHNLTYPE chnlType, UINT2
pgNum, sTBS_SPTSLOT *psrcSlot)
```

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`tdir`	: traffic direction
`chnlType`	: channel type
`pgNum`	: connection page to be updated
`psrcSlot`	: pointer to source space-time slot

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`

```
TBS_ERR_INVALID_STATE
TBS_ERR_INVALID_ARG
TBS_ERR_CONNECT_NONEXISTENT
```

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Verifying a multicast connection in TSI: tbsIsMulticast

Query if a given source space-time slot is multi-casting. The returned value of this function can either be: (1) the total number of connections if it is a non-negative number or (2) an error code if it is a negative number.

Prototype

```
INT4 tbsIsMulticast(sTBS_HNDL deviceHandle,
eTBS_TRAFFICDIR tdir, eTBS_CHNLTYPE chnlType, UINT2
pgNum, sTBS_SPTSLOT *psrcSlot)
```

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`tdir`	: traffic direction
`chnlType`	: channel type

	`pgNum`	: connection page to be updated
	`psrcSlot`	: pointer to source space-time slot

Outputs None

Returns Success = Total number of connections. If it is greater than one, it is a multicast.
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_INVALID_ARG`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Verifying the connection map setting in TSI: tbsIsValidMap

This function acts as a sanity check for the connection map setting in the receive side TSIs; it verifies the current RWTSEN, RPTSEN, and RASTEN setting in the receive working, protection and auxiliary TSIs. When the `RWSEL_EN` bit (accessible via the API `tbsDeviceSetConfig` and `tbsDeviceGetConfig`) is low, the RxTSEN bit chooses which timeslot from the RxTI will be the data source to the outgoing TelecomBus. The purpose of this function is to detect multiple RxTSEN bit settings for a possible error in setting the map. (Note: If multiple RxTSEN bits are set, the RWTSEN will override the others; this means that the working-channel data will be selected.)

Prototype
```
INT4 tbsIsValidMap( sTBS_HNDL deviceHandle, BOOL
activePage)
```

Inputs	`deviceHandle`	: device Handle (from `tbsAdd`)
	`activePage`	: high to check active page, low to check inactive page

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_INVALID_MAP`
`TBS_FAILURE`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

5.6 8B/10B Decoder

8B/10B decoder (Rx8D) frames to the receive data stream to find 8B/10B character boundaries. Functions, mainly diagnostics in nature, are available for interacting with these decoders.

Forcing out of character alignment in 8B/10B decoder: tbsForceOutofChar

Force the character alignment block in the 8B/10B decoder (Rx8D) into the out-of-character alignment state. The block will search for and synchronize with the alignment character (K28.5) in the data stream.

Prototype

```
INT4 tbsForceOutofChar(sTBS_HNDL deviceHandle,
eTBS_CHNLTYPE chnlType, UINT2 portNum)
```

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`chnlType`	: channel type
`portNum`	: port number, valid range from 1 to 4

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_INVALID_ARG`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Forcing out of frame alignment in 8B/10B decoder: tbsForceOutofFrm

Force the frame alignment block in the 8B/10B decoder (Rx8D) into the out-of-frame alignment state. The block will search for and synchronize with the alignment character (K28.5) in the data stream.

Prototype

```
INT4 tbsForceOutofFrm(sTBS_HNDL deviceHandle,
eTBS_CHNLTYPE chnlType, UINT2 blkType, UINT2 portNum)
```

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`chnlType`	: channel type
`portNum`	: port number, valid range from 1 to 4
`blkType`	: obsolete, value is ignored

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_INVALID_ARG`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

5.7 Disparity Encoder

Disparity encoders ensure the disparity integrity of the 8B/10B character streams after a possible time slot remapping. Diagnostic functions are available for these encoders.

Forcing line code violation in disparity encoder: tbsTXDEForceLcv

Generate line code violation by inverting data to generate complementary running disparity in the disparity encoder blocks.

Prototype

```
INT4 tbsTXDEForceLcv(sTBS_HNDL deviceHandle,
eTBS_CHNLTYPE chnlType, UINT2 portNum, BOOL activate)
```

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`chnlType`	: channel type
`portNum`	: port number, valid range from 1 to 4
`activate`	: 0 = disable, 1 = enable

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_INVALID_ARG`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Centering FIFO in disparity encoder: tbsTXDECenterFIFO

Force the FIFO depth to be 4 8B/10B characters deep when the current FIFO depth is not in the range of 3, 4, or 5 characters. If the current FIFO depth is in the range of 3, 4 or 5, this function has no effect.

Prototype

```
INT4 tbsTXDECenterFIFO(sTBS_HNDL deviceHandle,
eTBS_CHNLTYPE chnlType, UINT2 portNum)
```

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`chnlType`	: channel type
`portNum`	: port number, valid range from 1 to 4

Outputs None

Returns	Success = `TBS_SUCCESS`
	Failure = `TBS_ERR_INVALID_DEV`
	`TBS_ERR_INVALID_STATE`
	`TBS_ERR_INVALID_ARG`

Valid States	`TBS_ACTIVE, TBS_INACTIVE`
Side Effects	momentary data corruption

Inserting test pattern in disparity encoder: tbsInsertTP

Insert user-specified test pattern into the disparity encoder block. The default pattern is 0x02aa.

Prototype

```
INT4 tbsInsertTP(sTBS_HNDL deviceHandle,
eTBS_CHNLTYPE chnlType, UINT2 portNum, UINT2 tp, BOOL
activate)
```

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`chnlType`	: channel type
`portNum`	: port number
`tp`	: test pattern
`activate`	: flag, 0 = disable, 1 = enable

Outputs	None

Returns	Success = `TBS_SUCCESS`
	Failure = `TBS_ERR_INVALID_DEV`
	`TBS_ERR_INVALID_STATE`
	`TBS_ERR_INVALID_ARG`

Valid States	`TBS_ACTIVE, TBS_INACTIVE`
Side Effects	None

5.8 PRBS Processors

The PRBS processors are present in the device for diagnostic use. Functions are designed to interact with them to facilitate the process.

Configuring and retrieving payload for PRBS processor: tbsPayloadCfg

This function configure the PRBS processors, either generators or monitors, for specific traffic patterns. Payload configuration on all ports is recommended. The function can also be invoked to retrieve the current payload configuration for a specified PRBS processor.

Prototype

```
INT4 tbsPayloadCfg( sTBS_HNDL deviceHandle,
eTBS_TRAFFICDIR tdir, eTBS_CHNLTYPE chnlType, UINT2
eTBS_TRAFFICDIR tdir, eTBS_CHNLTYPE chnlType, UINT2
portNum, UINT2 genmon, sTBS_CFG_PYLD  *pplParam, BOOL
rd)
```

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`tdir`	: traffic direction
`chnlType`	: channel type
`portNum`	: port number, valid range from 1 to 4
`genmon`	: 0 = generator, 1 = monitor configuration
`pplParam`	: pointer to payload configuration parameter block
`rd`	: FALSE = configure, TRUE = retrieve

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_INVALID_ARG`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Configuring PRBS generator: tbsPrbsGenCfg

This function configures the PRBS generator for a given space-time slot. The attributes of the generator can be specified in the pcfgParam data block. Options like putting the generator in TelecomBus or autonomous mode, PRBS byte-inversion, and B1/E1 byte-insertion can all be specified in that block. For STS-Nc signals, only the first STS-1 slot requires configuration. Subsequent configurations for other STS-1 slots overwrites previous ones.

Prototype

```
INT4 tbsPrbsGenCfg( sTBS_HNDL deviceHandle,
eTBS_TRAFFICDIR tdir, sTBS_SPTSLOT *psptSlot,
sTBS_CFG_PRBS  *pcfgParam,  UINT2 rd)
```

Inputs

`deviceHandle`	: device handle (from `tbsAdd`)
`tdir`	: traffic direction
`psptSlot`	: pointer to space time slot
`pcfgParam`	: pointer to parameter configuration block
`rd`	: 0 = write config to device,1 = read config from device

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_INVALID_ARG`

`TBS_ERR_POLL_TIMEOUT`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Forcing a bit error in PRBS generator: tbsPrbsForceBitErr

This function controls the insertion of single-bit errors into the generated PRBS data stream. A single-bit error is introduced by inverting the most significant byte (MSB) of a single PRBS byte.

Prototype
```
INT4 tbsPrbsForceBitErr( sTBS_HNDL deviceHandle,
eTBS_TRAFFICDIR tdir, sTBS_SPTSLOT *psptSlot)
```

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`tdir`	: traffic direction
`psptSlot`	: channel type

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_INVALID_ARG`
`TBS_ERR_POLL_TIMEOUT`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Configuring PRBS monitor: tbsPrbsMonCfg

This function configures the PRBS monitor for a given space-time slot. The attributes of the monitor can be specified in the pcfgParam data block. Options like putting the monitor in TelecomBus or autonomous mode, PRBS byte-inversion before comparing, PRBS byte-setting, and B1/E1 byte-monitoring can all be specified in that block. For STS-Nc signals, only the first STS-1 slot requires configuration. Subsequent configurations for other STS-1 slots overwrite the previous ones.

Prototype
```
INT4 tbsPrbsMonCfg( sTBS_HNDL deviceHandle,
eTBS_TRAFFICDIR tdir, eTBS_CHNLTYPE chnlType,
sTBS_SPTSLOT *psptSlot, sTBS_CFG_PRBS  *pcfgParam,
UINT2 rd)
```

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`tdir`	: traffic direction
`chnlType`	: channel type
`psptSlot`	: pointer to space time slot
`pcfgParam`	: pointer to monitor configuration block
`rd`	: 0 = write config to device,1 = read config from

device

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_INVALID_ARG`
`TBS_ERR_POLL_TIMEOUT`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Resynchronizing PRBS monitor: tbsPrbsResync

This function forces the PRBS monitor to enter the out-of-sync state on a per space-time slot basis. The monitor then attempts to regain synchronization. The argument chnlType is not used for the transmit-side PRBS monitor. Only the first STS-1 time slot requires resynchronization for a STS-Nc signal.

Prototype

```
INT4 tbsPrbsResync( sTBS_HNDL deviceHandle,
eTBS_TRAFFICDIR tdir, eTBS_CHNLTYPE chnlType,
sTBS_SPTSLOT *psptSlot)
```

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`tdir`	: traffic direction
`chnlType`	: channel type
`psptSlot`	: pointer to space time slot

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_INVALID_ARG`
`TBS_ERR_POLL_TIMEOUT`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

5.9 Interrupt Service Functions

This Section describes the 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 28 for an explanation of our interrupt servicing architecture.

Configuring ISR Processing: tbsISRConfig

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

Prototype	`INT4 tbsISRConfig( sTBS_HNDL deviceHandle, UINT2 mode)`
Inputs	`deviceHandle` : device Handle (from `tbsAdd`) `mode` : mode of operation
Outputs	None

Returns	Success = `TBS_SUCCESS`
	Failure = `TBS_ERR_INVALID_DEV`
	`TBS_ERR_INVALID_ARG`

Valid States	`TBS_PRESENT, TBS_ACTIVE, TBS_INACTIVE`
Side Effects	None

Getting the Interrupt Status Mask: tbsGetMask

Returns the contents of the interrupt mask registers of the TBS device.

Prototype	`INT4 tbsGetMask( sTBS_HNDL deviceHandle, sTBS_MASK *pmask)`
Inputs	`deviceHandle` : device Handle (from `tbsAdd`) `pmask` : (pointer to) mask structure
Outputs	`pmask` : (pointer to) updated mask structure

Returns	Success = `TBS_SUCCESS`
	Failure = `TBS_ERR_INVALID_DEV`
	`TBS_ERR_INVALID_STATE`

Valid States	`TBS_ACTIVE, TBS_INACTIVE`
Side Effects	None

Setting the Interrupt Enable Mask: tbsSetMask

Sets the contents of the interrupt mask registers of the TBS device. Any bits that are set in the passed structure are cleared in the associated TBS registers.

Prototype	`INT4 tbsSetMask( sTBS_HNDL deviceHandle, sTBS_MASK *pmask)`	
Inputs	`deviceHandle`	: device Handle (from `tbsAdd`)
	`pmask`	: (pointer to) mask structure
Outputs	None	

Returns	Success = `TBS_SUCCESS`
	Failure = `TBS_ERR_INVALID_DEV`
	`TBS_ERR_INVALID_STATE`	

Valid States	`TBS_ACTIVE, TBS_INACTIVE`	
Side Effects	May change the operation of the ISR / DPR	

Clearing the Interrupt Enable Mask: tbsClearMask

Clears individual interrupt bits and registers in the TBS device. Any bits that are set in the passed structure are cleared in the associated TBS registers.

Prototype	`INT4 tbsClearMask( sTBS_HNDL deviceHandle, sTBS_MASK *pmask)`	
Inputs	`deviceHandle`	: device Handle (from `tbsAdd`)
	`pmask`	: (pointer to) mask structure
Outputs	None	

Returns	Success = `TBS_SUCCESS`
	Failure = `TBS_ERR_INVALID_DEV`
	`TBS_ERR_INVALID_STATE`	

Valid States	`TBS_ACTIVE, TBS_INACTIVE`	
Side Effects	May change the operation of the ISR / DPR	

Polling the Interrupt Status Registers: tbsPoll

Commands the driver to poll the interrupt registers in the device. The call will fail unless the device was initialized (via `tbsInit`) or configured (via `tbsISRConfig)` into polling mode.

Prototype `INT4 tbsPoll( sTBS_HNDL deviceHandle)`

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

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_INVALID_MODE`

Valid States `TBS_ACTIVE`

Side Effects None

Interrupt Service Routine: tbsISR

Reads the state of the interrupt registers in the TBS and stores them in an ISV. Performs whatever functions are needed to clear the interrupt; these range from simply clearing the bits to performing complex functions. This routine is called by the application code from within `sysTbsISRHandler`. 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: care should be taken while designing these functions; keeping in mind all possible issues when multiple devices are present, and some are in polling mode and some are in ISR mode.

Prototype `void* tbsISR( sTBS_HNDL deviceHandle)`

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

Outputs None

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

Valid States `TBS_ACTIVE`

Side Effects None

Deferred Processing Routine: tbsDPR

Acts on data contained in the passed ISV, allocates one or more DPV buffers (via `sysTbsDPVBufferGet`) and invokes one or more callbacks (if defined and enabled). This routine is called by the application code within `sysTbsDPRTask`. Note that the callbacks are responsible for releasing the passed DPV. It is recommended that it be done as soon as possible to avoid running out of DPV buffers. Note: care should be taken while designing those functions, keeping in mind all possible issues when multiple devices are present, and some are in polling mode and some are in ISR mode.

Prototype `void tbsDPR( void *pisv)`

Inputs	`pisv`	: (pointer to) ISV buffer
Outputs	None	
Returns	None	
Valid States	`TBS_ACTIVE`	
Side Effects	None	

5.10 Alarm, Status, and Statistics Functions

Getting the Cumulative Device Statistics: tbsGetStats

This function retrieves all the cumulative statistical counts. It is the user's responsibility to ensure that the structure points to an area of memory large enough to hold the returned data.

Prototype `INT4 tbsGetStats( sTBS_HNDL deviceHandle, sTBS_CNTR *pCstats)`

Inputs	`deviceHandle`	: device Handle (from `tbsAdd`)
	`pCstats`	: (pointer to) cumulative statistics counter block
Outputs	`pCstats`	: updated statistics counter block

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_FAILURE`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Clearing the Device Statistics: tbsClrStats

This function clears the cumulative statistical counts that are kept in the DDB. Non-zero fields in the passed structure correspond to the counters that will be cleared. A NULL pointer will clear all the fields in the counter.

Prototype `INT4 tbsClrStats( sTBS_HNDL deviceHandle, sTBS_CNTR *pCstats)`

Inputs	`deviceHandle`	: device Handle (from `tbsAdd`)
	`pCstats`	: (pointer to) cumulative statistics counter block

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_FAILURE`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Getting Status of the Device : tbsGetStatus

This function returns the current status of the device.

Prototype

```
INT4 tbsGetStatus( sTBS_HNDL deviceHandle,
sTBS_STATUS  *pStatus)
```

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`pStatus`	: (pointer to) device status block

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_POLL_TIMEOUT`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Getting Delta Statistics Counter of the Device : tbsGetDelta

This function returns the delta statistics counter block. The delta statistics counter block clears after each read.

Prototype

```
INT4 tbsGetDelta(sTBS_HNDL deviceHandle, sTBS_CNTR
*pDstats)
```

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`pDstats`	: (pointer to) delta statistics counter block

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`

`TBS_FAILURE`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Getting Event Threshold of the Device : tbsGetThresh

This function retrieves the current threshold setting for all the device events.

Prototype

```
INT4 tbsGetThresh(sTBS_HNDL deviceHandle, sTBS_CNTR
*pThresh)
```

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`pThresh`	: (pointer to) event threshold block

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`

Valid States TBS_ACTIVE, TBS_INACTIVE

Side Effects None

Setting Event Threshold of the Device : tbsSetThresh

This function allows user to update the event threshold setting. The threshold value is used to control the frequency of callbacks; these will only occur if the event occurrence exceeds the threshold value

Prototype

```
INT4 tbsSetThresh(sTBS_HNDL deviceHandle, sTBS_CNTR
*pThresh)
```

Inputs

`deviceHandle`	: device Handle (from `tbsAdd`)
`pThresh`	: (pointer to) event threshold block

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

Controlling AIS Generation of the Device : tbsGenAIS

This function enables/disables AIS alarm generation in Rx8D blocks.

Prototype `INT4 tbsGenAIS( sTBS_HNDL deviceHandle, eTBS_CHNLTYPE chnlType, UINT2 portNum, UINT2 blkType, BOOL enable)`

Inputs

`deviceHandle`	: device Handle (from tbsAdd)
`chnlType`	: channel type
`portNum`	: port number
`blkType`	: obsolete, value is ignored
`enable`	: flag to control AIS insertion, 0 = disable, 1 = enable

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_INVALID_ARG`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects None

5.11 Device Diagnostics

Testing Register Accesses: tbsTestReg

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

Prototype `INT4 tbsTestReg( sTBS_HNDL deviceHandle)`

Inputs

`deviceHandle`	: device Handle (from `tbs`Add)

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`
`TBS_ERR_FAILRAMTEST`

Valid States `TBS_PRESENT`

Side Effects None

Testing RAM Accesses: tbsTestRAM

This 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 tbsTestRAM( sTBS_HNDL deviceHandle)`
Inputs	`deviceHandle` : device Handle (from `tbsAdd`)
Outputs	None

Returns	Success = `TBS_SUCCESS`
	Failure = `TBS_ERR_INVALID_DEV`
	`TBS_ERR_INVALID_STATE`
	`TBS_ERR_FAILRAMTEST`

Valid States	`TBS_PRESENT`
Side Effects	None

Enabling outgoing to incoming parallel TelecomBus Loopbacks: tbsLoopOut2InTCB

This clears / sets the outgoing to incoming parallel TelecomBus Loopback. It is up to the user to perform any tests on the looped data.

Prototype	`INT4 tbsLoopOut2InTCB( sTBS_HNDL deviceHandle, BOOL enable)`
Inputs	`deviceHandle` : device Handle (from `tbsAdd`) `enable` : sets loop if non-zero, else clears loop
Outputs	None

Returns	Success = `TBS_SUCCESS`
	Failure = `TBS_ERR_INVALID_DEV`
	`TBS_ERR_INVALID_STATE`

Valid States	`TBS_ACTIVE, TBS_INACTIVE`
Side Effects	Will inhibit the flow of active data

Enabling incoming to outgoing parallel TelecomBus Loopbacks: tbsLoopIn2OutTCB

This clears / sets the incoming to outgoing parallel TelecomBus Loopback. It is up to the user to perform any tests on the looped data.

Prototype	`INT4 tbsLoopIn2OutTCB( sTBS_HNDL deviceHandle, BOOL enable)`
Inputs	`deviceHandle` : device Handle (from `tbsAdd`) `enable` : sets loop if non-zero, else clears loop
Outputs	None

Returns	Success = `TBS_SUCCESS`
	Failure = `TBS_ERR_INVALID_DEV`
	`TBS_ERR_INVALID_STATE`

Valid States	`TBS_ACTIVE, TBS_INACTIVE`
Side Effects	Will inhibit the flow of active data

Enabling receive to transmit serial TelecomBus Loopbacks: tbsLoopRx2TxLVDS

This clears / sets the receive to transmit serial TelecomBus Loopback. It is up to the user to perform any tests on the looped data.

Prototype	`INT4 tbsLoopRx2TxLVDS( sTBS_HNDL deviceHandle, sTBS_CHNLTYPE chnlType, BOOL enable)`
Inputs	`deviceHandle` : device Handle (from tbsAdd) `chnlType` : channel type `enable` : sets loop if non-zero, else clears loop
Outputs	None

Returns	Success = `TBS_SUCCESS`
	Failure = `TBS_ERR_INVALID_DEV`
	`TBS_ERR_INVALID_STATE`
	`TBS_ERR_INVALID_ARG`

Valid States	`TBS_ACTIVE, TBS_INACTIVE`
Side Effects	Will inhibit the flow of active data

Enabling transmit to receive serial TelecomBus Loopbacks: tbsLoopTx2RxLVDS

This clears / sets the transmit to receive serial TelecomBus Loopback. It is up to the user to perform any tests on the looped data.

Prototype	`INT4 tbsLoopTx2RxLVDS( sTBS_HNDL deviceHandle, BOOL enable)`
Inputs	`deviceHandle` : device Handle (from `tbsAdd`)

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

Outputs None

Returns Success = `TBS_SUCCESS`
Failure = `TBS_ERR_INVALID_DEV`
`TBS_ERR_INVALID_STATE`

Valid States `TBS_ACTIVE, TBS_INACTIVE`

Side Effects Will inhibit the flow of active data

5.12 Callback Functions

The TBS driver has the capability to callback to functions within the user code when certain events occur. These events and their associated callback routine declarations are detailed below. There is no user code action that is required by the driver for these callbacks – the user is free to either: (1) implement these callbacks in any manner or (2) to delete them from the driver.

The names given to the callback functions are given as examples only. The addresses of the callback functions invoked by the `tbsDPR` function are passed during the `tbsInit` call (inside a DIV). However, the user should use the exact same prototype. The application is responsible for releasing the passed DPV as soon as possible (to avoid running out of DPV buffers) by calling `sysTbsDPVBufferRtn` either within the callback function or later inside the application code.

Calling Back to the Application due to IO events: cbackIO

This callback function is provided by the user and is used by the DPR to report significant I/O events back to the application. The 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 `tbsInit` call. If the address of the callback function was passed as a NULL at initialization, no callback will be made.

Prototype `void cbackIO( sTBS_USR_CTXT usrCtxt, sTBS_DPV *pdpv)`

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

Outputs None

Returns None

Valid States `TBS_ACTIVE`

Side Effects None

Calling Back to the Application due to TSI events: cbackTSI

This callback function is provided by the user and is used by the DPR to report significant TSI events back to the application. The 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 `tbsInit` call. If the address of the callback function was passed as a NULL at initialization, no callback will be made.

Prototype `void cbackTSI( sTBS_USR_CTXT usrCtxt, sTBS_DPV *pdpv)`

Inputs

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

Outputs None

Returns None

Valid States `TBS_ACTIVE`

Side Effects None

Calling Back to the Application due to PRBS events: cbackPRBS

This callback function is provided by the user and is used by the DPR to report significant PRBS events back to the application. The 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 `tbsInit` call. If the address of the callback function was passed as a NULL at initialization, no callback will be made.

Prototype
```
void cbackPRBS( sTBS_USR_CTXT usrCtxt, sTBS_DPV
*pdpv)
```

Inputs

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

Outputs None

Returns None

Valid States `TBS_ACTIVE`

Side Effects None

Calling Back to the Application due to TXDE events: cbackTXDE

This callback function is provided by the user and is used by the DPR to report significant transmit disparity encoder events back to the application. The 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 `tbsInit` call. If the address of the callback function was passed as a NULL at initialization, no callback will be made.

Prototype `void cbackTXDE( sTBS_USR_CTXT usrCtxt, sTBS_DPV *pdpv)`

Inputs

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

Outputs None

Returns None

Valid States `TBS_ACTIVE`

Side Effects None

Calling Back to the Application due to RX8D events: cbackRX8D

This callback function is provided by the user and is used by the DPR to report significant 8B/10B decoder events back to the application. The 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 `tbsInit` call. If the address of the callback function was passed as a NULL at initialization, no callback will be made.

Prototype `void cbackRX8D( sTBS_USR_CTXT usrCtxt, sTBS_DPV *pdpv)`

Inputs

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

Outputs None

Returns None

Valid States `TBS_ACTIVE`

Side Effects None

6 HARDWARE INTERFACE

The TBS 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: sysTbsRead

The most basic hardware connection reads the contents of a specific register location. This Macro should be UINT2 oriented and should be defined by the user to reflect the target system's addressing logic. There is no need for error recovery in this function.

Format `#define sysTbsRead(ba, offset)`

Prototype `UINT2 sysTbsRead(UINT2 ba, UINT2 offset)`

Inputs

`ba`	: base address of the device
`offset`	: offset of the register from base address to be read

Outputs None

Returns Value read from the addressed register location

Writing to a Device Register: sysTbsWrite

The most basic hardware connection writes the supplied value to the specific register location. This macro should be UINT2 oriented and should be defined by the user to reflect the target system's addressing logic. There is no need for error recovery in this function.

Format `#define sysTbsWrite(ba, offset, value)`

Prototype `UINT2 sysTbsWriteReg(UINT2 ba, UINT2 offset, UINT2 value)`

Inputs

`ba`	: register location to be written
`offset`	: offset of the register from base address to be written
`value`	: data to be written

Outputs None

Returns Value written to the addressed register location

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 (`sysTbsISRHandler`) in the interrupt vector table of the system processor. This handler should call `deviceISR` for each device that has interrupt servicing enabled to perform the ISR related housekeeping required by each device.

During execution of the API function `tbsModuleStart` / `tbsModuleStop`, the driver informs the application that it is time to install/uninstall this shell via `sysTbsISRHandlerInstall` / `sysTbsISRHandlerRemove` (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 the provided 'polling' routine (`tbsPoll`) that in turn calls `tbsISR`.

Installing the ISR Handler : sysTbsISRHandlerInstall

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

Format `#define sysTbsISRHandlerInstall( func)`

Prototype `INT4 sysTbsISRHandlerInstall( void *func)`

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

Outputs None

Returns Success = `0`
Failure = `<any other value>`

Pseudocode
```
Begin
        install sysTbsISRHandler   in processor's interrupt vector
table
End
```

ISR Handler: sysTbsISRHandler

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

Format `#define sysTbsISRHandler()`

Prototype `void sysTbsISRHandler( void)`

Inputs None

Outputs None

Returns None

Pseudocode

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

Removing the ISR Handler : sysTbsISRHandlerRemove

This performs Disable Interrupt Processing for this device; it also removes the user-supplied Interrupt Service routine (ISR), `sysTbsISRHandler,` from the processor's interrupt vector table.

Format `#define sysTbsISRHandlerRemove()`

Prototype `void sysTbsISRHandlerRemove( void)`

Inputs None

Outputs None

Returns None

Pseudocode

```
Begin
remove sysTbsISRHandler from the processor's interrupt vector
table
End
```

7 RTOS INTERFACE

The TBS 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: sysTbsMemAlloc

This allocates a specified number of bytes of memory.

Format	`#define sysTbsMemAlloc( numBytes)`
Prototype	`UINT1 *sysTbsMemAlloc( 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: sysTbsMemFree

This frees any memory allocated using `sysTbsMemAlloc`.

Format	`#define sysTbsMemFree( pfirstByte)`
Prototype	`void sysTbsMemFree( 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 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 this size during the `sysTbsBufferStart` call.

Starting Buffer Management : sysTbsBufferStart

This alerts the RTOS that the time has come to make sure that the ISV buffers and DPV buffers are available and sized correctly. This may involve either: (1) creating new buffer pools or (2) doing nothing − it depends upon the RTOS.

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

Returns	Success = `0`
	Failure = `<any other value>`

Getting an ISV Buffer: sysTbsISVBufferGet

This 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 sysTbsISVBufferGet()`
Prototype	`sTBS_ISV *sysTbsISVBufferGet( void)`
Inputs	None
Outputs	None

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

Returning an ISV Buffer: sysTbsISVBufferRtn

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

Format `#define sysTbsISVBufferRtn( pISV)`

Prototype `void sysTbsISVBufferRtn( sTBS_ISV *pisv)`

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

Outputs None

Returns None

Getting a DPV Buffer: sysTbsDPVBufferGet

This 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 sysTbsDPVBufferGet()`

Prototype `sTBS_DPV *sysTbsDPVBufferGet( void)`

Inputs None

Outputs None

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

Returning a DPV Buffer: sysTbsDPVBufferRtn

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

Format `#define sysTbsDPVBufferRtn( pDPV)`

Prototype `void sysTbsDPVBufferRtn( sTBS_DPV *pdpv)`

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

Outputs None

Returns None

Stopping Buffer Management : sysTbsBufferStop

This alerts the RTOS: (1) that the driver no longer needs any of either the ISV buffers or the DPV buffers and (2) that if any special resources were created to handle these buffers, they can be deleted now.

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

7.3 Timers

Sleeping a Task: sysTbsTimerSleep

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

Format	`#define sysTbsTimerSleep( time)`
Prototype	`void sysTbsTimerSleep( UINT4 time)`
Inputs	`time` : sleep time in milliseconds
Outputs	None

Returns	Success = `0`
	Failure = `<any other value>`

7.4 Semaphores

Creating a Semaphore: sysTbsSemCreate

This creates a binary semaphore object.

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

Returns	Success = (pointer to) a semaphore object
	Failure = NULL (pointer)

Taking a Semaphore: sysTbsSemTake

This takes a binary semaphore.

Format	`#define sysTbsSemTake( psem)`
Prototype	`INT4 sysTbsSemTake( void *psem)`
Inputs	`psem` : (pointer to) a semaphore object
Outputs	None

Returns	Success = `0`
	Failure = `<any other value>`

Giving a Semaphore: sysTbsSemGive

This gives a binary semaphore.

Format	`#define sysTbsSemGive( psem)`
Prototype	`INT4 sysTbsSemGive( void *psem)`
Inputs	`psem` : (pointer to) a semaphore object
Outputs	None

Returns	Success = `0`
	Failure = `<any other value>`

Deleting a Semaphore: sysTbsSemDelete

This deletes a binary semaphore object.

Format	`#define sysTbsSemDelete( psem)`
Prototype	`void sysTbsSemDelete( void *psem)`
Inputs	`psem` : (pointer to) a semaphore object
Outputs	None
Returns	None

7.5 Preemption

Disabling Preemption : sysTbsPreemptDisable

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

Format	`#define sysTbsPreemptDisable()`
Prototype	`INT4 sysTbsPreemptDisable( void)`
Inputs	None
Outputs	None
Returns	Preemption key (passed back as an argument in `sysTbsPreemptEnable`)

Re-Enabling Preemption : sysTbsPreemptEnable

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

Format	`#define sysTbsPreemptEnable( key)`	
Prototype	`void sysTbsPreemptEnable( INT4 key)`	
Inputs	`key`	: preemption key (returned by `sysTbsPreemptDisable`)
Outputs	None	
Returns	None	

7.6 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’ (`sysTbsDPRTask`) that in turn calls `tbsDPR` with an ISV to perform the ISR related processing that is required by each interrupting device.

During execution of the API function `tbsModuleStart` / `tbsModuleStop`, the driver informs the application that it is time to install/uninstall this shell via `sysTbsDPRTaskInstall` / `sysTbsDPRTaskRemove` (which needs to be supplied by the user).

Installing the DPR Task: sysTbsDPRTaskInstall

This informs the application that it is time to install the user-supplied function `sysTbsDPRTask` into the RTOS as a task.

Format `#define sysTbsDPRTaskInstall( func)`

Prototype `INT4 sysTbsDPRTaskInstall( void *func)`

Inputs func : (pointer to) the function tbsDPR

Outputs None

Returns Success = `0`
Failure = `<any other value>`

Pseudocode

```
Begin
install sysTbsDPRTask in the RTOS as a task
End
```

DPR Task: sysTbsDPRTask

This routine is installed as a separate task within the RTOS. It runs periodically to retrieve the interrupt status information sent to it by `tbsISR` ; it then invokes `tbsDPR` for the appropriate device.

Format `#define sysTbsDPRTask()`

Prototype `void sysTbsDPRTask( void)`

Inputs None

Outputs None

Returns None

Pseudocode

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

Removing the DPR Task: sysTbsDPRTaskRemove

This informs the application that it is time to remove (suspend) the user-supplied task, `sysTbsDPRTask`.

Format `#define sysTbsDPRTaskRemove()`

Prototype `void sysTbsDPRTaskRemove( void)`

Inputs None

Outputs None

Returns None

Pseudocode

```
Begin
remove/suspend sysTbsDPRTask
End
```

8 PORTING THE TBS DRIVER

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

8.1 Driver Source Files

The C source files listed in Table 26 contain the code for the TBS driver. You may need to either modify the code or develop additional code. The code is in the form of constants, macros, and functions. For ease of porting, the code is grouped into source files (`src`) and include files (`inc`). The `src` files contain the functions and the `inc` files contain the constants and macros.

8.2 Driver Porting Procedures

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

To port the TBS driver to your platform:

Procedure 1: Port the driver's OS extensions (page 105):

Procedure 2: Port the driver to your hardware platform (page 106):

Procedure 3: Port the driver's application-specific elements (page 106):

Procedure 4: Build the driver (page 108).

Porting Assumptions

The following porting assumptions have been made:

It is assumed that RAM assigned to the driver's static variables is initialized to ZERO before any driver function is called.

It is assumed that a RAM stack of 4K is available to all of the driver's non-ISR functions and that a RAM stack of 1K is available to the driver's ISR functions.

- It is assumed that there is no memory management or MMU in the system and that all accesses by the driver, to either memory or hardware, can be direct.

Procedure 1: Porting Driver OS Extensions

The RTOS extensions encapsulate all RTOS specific services and data types used by the driver. The `tbs_rtos.h` file contains data types and compiler-specific data-type definitions. It also contains macros for RTOS specific services used by the OS extensions. These RTOS 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 `tbs_rtos.c` file contains the utility and interrupt-event handler functions that use RTOS specific services.

To port the driver's OS extensions:

1. Modify the data types in tbs_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 RTOS specific services in tbs_rtos.h. Redefine the following macros to the corresponding system calls that your target system supports:

Service Type	Macro Name	Description
Memory	`sysTbsMemAlloc`	Allocates the memory block
	`sysTbsMemFree`	Frees the memory block
	`sysTbsMemCpy`	Copies the memory block from src to dest

3. Modify the utilities and interrupt services that use RTOS specific services in the `tbs_rtos.c.` The `tbs_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	`sysTbsMemSet`	Sets each character in the memory buffer

Timer	`sysTbsTimerSleep`	Sets the task execution delay in milliseconds
Interrupt	`sysTbsIntInstallHandler`	Installs the interrupt handler for the RTOS
	`sysTbsIntRemoveHandler`	Removes the interrupt handler from the RTOS
	`sysTbsISRHandler`	Interrupt handler for the TBS device
	`sysTbsDPRTask`	Deferred interrupt-processing routine (DPR)

Procedure 2: Porting Drivers to Hardware Platforms

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

To port the driver to your hardware platform:

1. Modify the low-level device read/write macros in the tbs_hw.h file. You may need to modify the raw read/write access macros (sysTbsReadReg and sysTbsWriteReg) to reflect your system's addressing logic.

2. Define the hardware system-configuration constants in the tbs_hw.h file. Modify the following constants to reflect your system's hardware configuration:

Device Constant	Description	Default
`TBS_MEM_ADDR_RANGE`	The assigned address memory range for each TBS device. Your system's memory map determines it.	0x1000
`TBS_MAX_DEVS`	The maximum number of TBS devices on each card	16

Procedure 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 TBS driver.

To port the driver's application-specific elements:

1. Define the following driver task-related constants for your OS-specific services in file `tbs_rtos.h and tbs_hw.h:`

Task Constant	Description	Default
`TBS_DPR_TASK_PRIORITY`	Deferred Task (DPR) task priority	85
`TBS_DPR_TASK_STACK_SZ`	DPR task stack size, in bytes	16384
`TBS_STATTASK_PRIORITY`	Statistics task priority	95
`TBS_STATTASK_STACK_SZ`	Statistics task stack size, in bytes	8192
`TBS_POLL_DELAY`	Constant used in polling task mode, this constant defines the interval time in millisecond between each polling action	1000
`TBS_TASK_SHUTDOWN_` `DELAY`	Delay time in millisecond. 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
`TBS_MAX_MSGS`	The queue message depth of the queue used for pass interrupt context between the ISR task and DPR task	500
`TBS_STATTASK_POLLPERIOD`	Statistics collection task polling period in milliseconds	100
`TBS_MAX_DEVS`	The maximum number of TBS devices in the system (from 1 to 128)	16

2. Code the callback functions according to your application. The driver will call callback functions when an event occurs on the device. The application is responsible for releasing the DPV buffer using `sysTbsDPVBufferRtn` after necessary processing is completed. These functions must conform to the following prototypes:

- `void cbackTbsXX(sTBS_CTXT usrCtxt, sTBS_DPV *pdpv)`
- …

Procedure 4: Building the Driver

This section describes how to build the TBS 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 TBS driver using your make utility.
3. Link the TBS 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 24: Variable Type Definitions

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

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 25: Naming Conventions

Type	Case	Naming convention	Examples
Macros	Uppercase	prefix with "m" and device abbreviation	mTBS_WRITE
Constants	Uppercase	prefix with device abbreviation	TBS_REG

Type	Case	Naming convention	Examples
Structures	Hungarian Notation	prefix with "`s`" and device abbreviation	`sTBS_DDB`
API Functions	Hungarian Notation	prefix with device name	`tbsAdd()`
Porting Functions	Hungarian Notation	prefix with "`sys`" and device name	`sysTbsReadReg()`
Other Functions	Hungarian Notation		`utilTbsSlotIsValid()`
Variables	Hungarian Notation		`maxDevs`
Pointers to variables	Hungarian Notation	prefix variable name with "`p`"	`pmaxDevs`
Global variables	Hungarian Notation	prefix with device name	`tbsMdb`

Macros

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

Constants

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

Structures

- Structure names must be all uppercase.
- Words shall be 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: `sTBS_DDB` is a valid name for a structure.

Functions

API Functions

- Naming of the API functions must follow the Hungarian notation.
- The device's full name in all lowercase shall be used as a prefix.
- Example: `tbsAdd()` 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 shall be used to indicate a porting function.
- The device's name starting with an uppercase must follow the prefix.
- Example: `sysTbsReadReg ()` is hardware / RTOS specific.

Other Functions

- Other Functions are all the remaining functions that are part of the driver and have no special naming convention. However, they must follow the Hungarian notation.
- Example: `utilTbsSlotIsValid()`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 `tbsBaseAddress` 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 26 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 number of characters. If a file size is getting too big, you can separate it into two or more files, providing that a number is added at the end of the file name (e.g. `tbs_api1.c` or `tbs_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 26: File Naming Conventions

File Type	**File Name**
API	`tbs_api.c, tbs_api.h`
Hardware Dependent	`tbs_hw.c, tbs_hw.h`
RTOS Dependent	`tbs_rtos.c,  tbs_rtos.h`
Other	`tbs_util.c, tbs_fns.h`

API Files

- The name of the API files must start with the device abbreviation followed by an underscore and `'api'`. Eventually a number might be added at the end of the name.
- Examples: `tbs_api1.c` is the only valid name for the file that contains the first part of the API functions; `tbs_api.h` is the only valid name for the file that contains all of the API functions headers.

Hardware Dependent Files

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

RTOS Dependent Files

- The name of the RTOS dependent files must start with the device abbreviation followed by an underscore and `'rtos'`. Eventually a number might be added at the end of the file name.
- Examples: `tbs_rtos.c` is the only valid name for the file that contains all of the RTOS dependent functions, `tbs_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; this should convey the purpose of the functions within that file with a minimum amount of characters.
- Examples: `tbs_util.c` is a valid name for a file that would have utility functions of the device, `tbs_fns.h` is a valid name for the header file.

APPENDIX B: ERROR CODES

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

Table 27: TBS Error Codes

Error Code	Description
TBS_SUCCESS	Success
TBS_FAILURE	Failure
TBS_ERR_MEM_ALLOC	Not enough memory for allocation
TBS_ERR_INVALID_ARG	Invalid parameter
TBS_ERR_MODULE_NOT_OPEN	Module not open
TBS_ERR_MODULE_ALREADY_OPEN	Module already open
TBS_ERR_INVALID_MIV	Invalid module initialization vector
TBS_ERR_INVALID_STATE	Invalid device state
TBS_ERR_INVALID_MAP	Invalid connection map
TBS_ERR_DEVS_FULL	Device table is full
TBS_ERR_DEV_NOT_DETECTED	Device not found
TBS_ERR_DEV_ALREADY_ADDED	Device is already in table
TBS_ERR_INVALID_TYPE_ID	Manufacturer and/or device ID mismatch
TBS_ERR_INVALID_DEV	Invalid device handle
TBS_ERR_INVALID_DIV	Invalid device initialization vector
TBS_ERR_INT_INSTALL	Unable to install interrupt handler
TBS_ERR_INVALID_MODE	Invalid mode
TBS_ERR_INVALID_REG	Invalid register number
TBS_ERR_POLL_TIMEOUT	Indirect read/write busy bit timeout

`TBS_ERR_FAILRAMTEST`	RAM test failed
`TBS_ERR_CONNECT_EXIST`	Connection already exists
`TBS_ERR_CONNECT_NONEXISTENT`	Connection does not exist
`TBS_NODEBUG`	Debug not installed, use `TBS_DEBUG` compile switch

APPENDIX C: EVENT CODES

Table 28 below describes the interrupt event codes used in the TBS device driver. Note that specific callback is defined by the "event" and "cause" fields of the `sTBS_DPV` structure (for the structure's definition, please refer to Table 23). Information encoded in these two fields explicitly defines the cause of the callback. The "event" field encodes the nature of the callback (for example, `TBS_EVENT_TXDE_FIFOERR` represents a FIFO overrun/underrun error in either of the TWDE, the TPDE, or the TADE blocks); the "cause" field is a 16-bit parameter that can be interpreted as a bit vector, and is used to further indicate the absolute cause(s) of the callback event. Each cause event bit is encoded for a unique event, such that a "1" in a specific bit position indicates the occurrence of the event for that channel/port/time-slot. For example, the event `TBS_EVENT_TXDE_FIFOERR` in Table 28 is interpretted as follows: if the cause field is set to 0x85A (0000 1000 0101 1010), there have been FIFO errors detected in TWDE block port #2 (bit 1) and #4 (bit 3), TPDE port#1 (bit 4) and #3 (bit 6), and TADE port#4 (bit 11).

Table 28: TBS Event Codes

Event Code	Description	Cause
`TBS_EVENT_IO_IPE`	Incoming data parity error	bit 0..3: port# 1..4
`TBS_EVENT_IO_DLLERR`	DLL error	n/a, always 0
`TBS_EVENT_IO_CSULOCKCHG`	CSU lock status change	n/a, always 0
`TBS_EVENT_TXDE_FIFOERR`	FIFO error in TxDE block	bit 0..3: working port# 1..4 bit 4..7: protection port# 1..4 bit 8..11: auxiliary port# 1..4
`TBS_EVENT_ITPP1_BYTEERR`	Byte error in Tx PRBS monitor ITPP#1	bit 0..11: timeslot# 1..12
`TBS_EVENT_ITPP2_BYTEERR`	Byte error in Tx PRBS monitor ITPP#2	bit 0..11: timeslot# 1..12
`TBS_EVENT_ITPP3_BYTEERR`	Byte error in Tx PRBS monitor ITPP#3	bit 0..11: timeslot# 1..12
`TBS_EVENT_ITPP4_BYTEERR`	Byte error in Tx PRBS monitor ITPP#4	bit 0..11: timeslot# 1..12
`TBS_EVENT_ITPP1_B1E1MSH`	B1/E1 mismatch in Tx PRBS monitor ITPP#1	bit 0..11: timeslot# 1..12
`TBS_EVENT_ITPP2_B1E1MSH`	B1/E1 mismatch in Tx PRBS monitor ITPP#2	bit 0..11: timeslot# 1..12

`TBS_EVENT_ITPP3_B1E1MSH`	B1/E1 mismatch in Tx PRBS monitor ITPP#3	bit 0..11: timeslot# 1..12
`TBS_EVENT_ITPP4_B1E1MSH`	B1/E1 mismatch in Tx PRBS monitor ITPP#4	bit 0..11: timeslot# 1..12
`TBS_EVENT_ITPP1_SYNCCHG`	Sync state change in Tx PRBS monitor ITPP#1	bit 0..11: timeslot# 1..12
`TBS_EVENT_ITPP2_SYNCCHG`	Sync state change in Tx PRBS monitor ITPP#2	bit 0..11: timeslot# 1..12
`TBS_EVENT_ITPP3_SYNCCHG`	Sync state change in Tx PRBS monitor ITPP#3	bit 0..11: timeslot# 1..12
`TBS_EVENT_ITPP4_SYNCCHG`	Sync state change in Tx PRBS monitor ITPP#4	bit 0..11: timeslot# 1..12
`TBS_EVENT_RWPM1_BYTEERR`	Byte error in Rx working PRBS monitor RWPM#1	bit 0..11: timeslot# 1..12
`TBS_EVENT_RWPM2_BYTEERR`	Byte error in Rx working PRBS monitor RWPM#2	bit 0..11: timeslot# 1..12
`TBS_EVENT_RWPM3_BYTEERR`	Byte error in Rx working PRBS monitor RWPM#3	bit 0..11: timeslot# 1..12
`TBS_EVENT_RWPM4_BYTEERR`	Byte error in Rx working PRBS monitor RWPM#4	bit 0..11: timeslot# 1..12
`TBS_EVENT_RPPM1_BYTEERR`	Byte error in Rx protection PRBS monitor RPPM#1	bit 0..11: timeslot# 1..12
`TBS_EVENT_RPPM2_BYTEERR`	Byte error in Rx protection PRBS monitor RPPM#2	bit 0..11: timeslot# 1..12
`TBS_EVENT_RPPM3_BYTEERR`	Byte error in Rx protection PRBS monitor RPPM#3	bit 0..11: timeslot# 1..12
`TBS_EVENT_RPPM4_BYTEERR`	Byte error in Rx protection PRBS	bit 0..11: timeslot# 1..12

	monitor RPPM#4	
`TBS_EVENT_RAPM1_BYTEERR`	Byte error in Rx auxiliary PRBS monitor RAPM#1	bit 0..11: timeslot# 1..12
`TBS_EVENT_RAPM2_BYTEERR`	Byte error in Rx auxiliary PRBS monitor RAPM#2	bit 0..11: timeslot# 1..12
`TBS_EVENT_RAPM3_BYTEERR`	Byte error in Rx auxiliary PRBS monitor RAPM#3	bit 0..11: timeslot# 1..12
`TBS_EVENT_RAPM4_BYTEERR`	Byte error in Rx auxiliary PRBS monitor RAPM#4	bit 0..11: timeslot# 1..12
`TBS_EVENT_RWPM1_B1E1MSH`	B1/E1 mismatch in Rx working PRBS monitor RWPM#1	bit 0..11: timeslot# 1..12
`TBS_EVENT_RWPM2_B1E1MSH`	B1/E1 mismatch in Rx working PRBS monitor RWPM#2	bit 0..11: timeslot# 1..12
`TBS_EVENT_RWPM3_B1E1MSH`	B1/E1 mismatch in Rx working PRBS monitor RWPM#3	bit 0..11: timeslot# 1..12
`TBS_EVENT_RWPM4_B1E1MSH`	B1/E1 mismatch in Rx working PRBS monitor RWPM#4	bit 0..11: timeslot# 1..12
`TBS_EVENT_RPPM1_B1E1MSH`	B1/E1 mismatch in Rx protection PRBS monitor RPPM#1	bit 0..11: timeslot# 1..12
`TBS_EVENT_RPPM2_B1E1MSH`	B1/E1 mismatch in Rx protection PRBS monitor RPPM#2	bit 0..11: timeslot# 1..12
`TBS_EVENT_RPPM3_B1E1MSH`	B1/E1 mismatch in Rx protection PRBS monitor RPPM#3	bit 0..11: timeslot# 1..12
`TBS_EVENT_RPPM4_B1E1MSH`	B1/E1 mismatch in Rx protection PRBS	bit 0..11: timeslot# 1..12

	monitor RPPM#4	
`TBS_EVENT_RAPM1_B1E1MSH`	B1/E1 mismatch in Rx auxiliary PRBS monitor RAPM#1	bit 0..11: timeslot# 1..12
`TBS_EVENT_RAPM2_B1E1MSH`	B1/E1 mismatch in Rx auxiliary PRBS monitor RAPM#2	bit 0..11: timeslot# 1..12
`TBS_EVENT_RAPM3_B1E1MSH`	B1/E1 mismatch in Rx auxiliary PRBS monitor RAPM#3	bit 0..11: timeslot# 1..12
`TBS_EVENT_RAPM4_B1E1MSH`	B1/E1 mismatch in Rx auxiliary PRBS monitor RAPM#4	bit 0..11: timeslot# 1..12
`TBS_EVENT_RWPM1_SYNCCHG`	Sync state change in Rx working PRBS monitor RWPM#1	bit 0..11: timeslot# 1..12
`TBS_EVENT_RWPM2_SYNCCHG`	Sync state change in Rx working PRBS monitor RWPM#2	bit 0..11: timeslot# 1..12
`TBS_EVENT_RWPM3_SYNCCHG`	Sync state change in Rx working PRBS monitor RWPM#3	bit 0..11: timeslot# 1..12
`TBS_EVENT_RWPM4_SYNCCHG`	Sync state change in Rx working PRBS monitor RWPM#4	bit 0..11: timeslot# 1..12
`TBS_EVENT_RPPM1_SYNCCHG`	Sync state change in Rx protection PRBS monitor RPPM#1	bit 0..11: timeslot# 1..12
`TBS_EVENT_RPPM2_SYNCCHG`	Sync state change in Rx protection PRBS monitor RPPM#2	bit 0..11: timeslot# 1..12
`TBS_EVENT_RPPM3_SYNCCHG`	Sync state change in Rx protection PRBS monitor RPPM#3	bit 0..11: timeslot# 1..12
`TBS_EVENT_RPPM4_SYNCCHG`	Sync state change in Rx protection PRBS	bit 0..11: timeslot# 1..12

	monitor RPPM#4	
`TBS_EVENT_RAPM1_SYNCCHG`	Sync state change in Rx auxiliary PRBS monitor RPPM#1	bit 0..11: timeslot# 1..12
`TBS_EVENT_RAPM2_SYNCCHG`	Sync state change in Rx auxiliary PRBS monitor RPPM#2	bit 0..11: timeslot# 1..12
`TBS_EVENT_RAPM3_SYNCCHG`	Sync state change in Rx auxiliary PRBS monitor RPPM#3	bit 0..11: timeslot# 1..12
`TBS_EVENT_RAPM4_SYNCCHG`	Sync state change in Rx auxiliary PRBS monitor RPPM#4	bit 0..11: timeslot# 1..12
`TBS_EVENT_TSI_MEMPGCHG`	TSI block connection page change	bit 0: RATI, bit 1: RPTI, bit 2: RWTI, bit 8: TATI, bit 9: TPTI, bit 10: TWTI
`TBS_EVENT_RX8D_OFA`	Out of frame alignment in Rx8D blocks	bit 0..3: working port# 1..4 bit 4..7: protection port# 1..4 bit 8..11: auxiliary port# 1..4
`TBS_EVENT_RX8D_OCA`	Out of character alignment in Rx8D blocks	bit 0..3: working port# 1..4 bit 4..7: protection port# 1..4 bit 8..11: auxiliary port# 1..4
`TBS_EVENT_RX8D_FUO`	FIFO underrun/overrun error in Rx8D blocks	bit 0..3: working port# 1..4 bit 4..7: protection port# 1..4 bit 8..11: auxiliary port# 1..4
`TBS_EVENT_RX8D_LCV`	Line code violation in Rx8D blocks	bit 0..3: working port# 1..4 bit 4..7: protection port# 1..4 bit 8..11: auxiliary port# 1..4

LIST OF TERMS

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

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

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

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

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

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

MODULE: All of the code that is part of this driver; there is only ONE instance of this MODULE connected to ONE OR MORE TBS chips.

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

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

ACRONYMS

API:	Application programming interface
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:	Initialization service (routine) vector
RTOS:	Real-time operating system
TSI:	Time Slot Interchange
PRBS:	Pseudo random bit sequence
TBS:	TelecomBus
LVDS:	Low voltage differential signal

INDEX

A

AIS
- functions
  - tbsGenAIS, 35, 87

AIS generation
- controlling AIS generation of device, 87

API
- API Files, 112

B

buffers
- buffer management, 97, 98
- functions
  - sysTbsBufferStart, 55, 97
  - sysTbsBufferStop, 98, 99
  - sysTbsDPVBufferGet, 83, 98
  - sysTbsDPVBufferRtn, 90, 98, 107
  - sysTbsISVBufferGet, 55, 97
  - sysTbsISVBufferRtn, 55, 97, 98
- getting
  - DPV buffer, 98
  - ISV buffer, 97
- returning
  - an ISV buffer, 97
  - DPV buffer, 98
- starting
  - buffer management, 97
- stopping
  - buffer management, 98

C

callbacks
- callback functions, 18, 90
- cbackIO, 40, 49, 90
- cbackPRBS, 40, 49, 91
- cbackRX8D, 40, 41, 49, 92
- cbackTSI, 40, 49, 91
- cbackTXDE, 40, 41, 49, 92
- IO events, 90
- PRBS events, 91
- RX8D events, 92
- TSI events, 91
- TXDE events, 92

clearing
- device statistics, 84
- interrupt enable mask, 82

configuration
- functions
  - pcfgParam, 78, 79
  - rxtiConMap, 40, 41
  - tbsDeviceGetConfig, 66, 67, 74
  - tbsDeviceSetConfig, 66, 74
  - tbsPayloadCfg, 36, 77
- input and output, 22

D

decoders
- 8B/10B decoder, 17, 23, 46, 53, 75
- receive 8B/10B decoder, 17, 23

deferred processing
- routine, 16, 55, 83, 121
- vector, 55, 98

device registers
- reading
  - device registers, 62

devices
- activating, 61
- adding, 59
- configuration, 42, 66
- data block, 25, 47, 48, 49, 56, 60, 61
- de-activating, 62
- deleting, 60
- diagnostics, 17, 87
- getting
  - event threshold, 86

status device, 85
initialization vector, 39, 40, 60, 61
initializing a device, 60
management, 26, 27, 59
read and write, 62
resetting, 61
setting
event threshold, 86
states, 25
status block, 52
updating configuration, 61

disparity encoder, 17, 21, 76

DPR

functions
sysTbsDPRTask, 28, 29, 83, 101, 102, 103, 106
sysTbsDPRTaskInstall, 102
sysTbsDPRTaskRemove, 102, 103
tbsDPR, 22, 28, 29, 30, 83, 90, 101, 102
removing
DPR Task, 103
system-specific DPR routine, 101
task, 102

driver

porting procedures, 104
source files, 104

drivers

closing
driver module, 57
porting drivers, 106
porting TBS driver, 104
starting
driver module, 58
stopping
driver module, 58

E

encoders

incoming 8B/10B encoder, 17, 23

errors

errDevice, 48, 49, 56, 59
errModule, 39, 48, 56

events

event codes, 116

F

FIFO

centering, 35

G

global variable, 56

H

hardware

hardware dependent files, 112
hardware interface, 20, 93

I

initialization

initPayloadCfg, 40, 41
initTSImap, 40, 41

input and output status block, 52

interrupts

functions
sysTbsIntInstallHandler, 106
sysTbsIntRemoveHandler, 106
installing
DPR task, 102
ISR handler, 94
removing ISR handler, 95
service routine, 83, 94, 121
service vector, 28, 30, 55, 97
servicing, 16, 27, 94
calling
tbsDPR, 29
tbsISR, 28
tbsPoll, 29
system-specific interrupt servicing, 94

ISR
functions
sysTbsISRHandler, 28, 29, 83, 94, 95, 106
sysTbsISRHandlerInstall, 29, 94
sysTbsISRHandlerRemove, 94, 95
TBS_ISR_MODE, 39, 81
tbsISR, 22, 28, 29, 30, 83, 94, 95, 102
tbsISRConfig, 81, 82
handler, 94, 95

ITPP, 31, 32, 35, 54

L

looping
functions
tbsLoopIn2OutTCB, 36, 88, 89
tbsLoopOut2InTCB, 36, 88
tbsLoopRx2TxLVDS, 36, 89
tbsLoopTx2RxLVDS, 36, 89

M

map
functions
tbsGetMapMode, 34, 68
tbsIsValidMap, 34, 74
tbsMapSlot, 33, 69
tbsSetMapMode, 34, 67

masks
functions
tbsClearMask, 45, 82
tbsGetMask, 45, 81
tbsSetMask, 45, 82
interrupts
getting interrupt status mask, 81
intrmask, 40, 41, 50
setting interrupt enable mask, 82
ISR
enable and disable mask, 45

memory
allocation, 96
deallocation, 96
freeing memory, 96
functions
sysTbsMemAlloc, 96, 105
sysTbsMemFree, 96, 105
sysTbsMemSet, 105

modules
data block, 25, 47, 48, 56
functions
tbsModuleClose, 57
tbsModuleOpen, 39, 57
tbsModuleStart, 58, 94, 102
tbsModuleStop, 58, 94, 102
initialization vector, 25, 39, 57
management, 26, 57
opening driver module, 57
states, 24

P

payload configuration block, 43

polling
functions
TBS_POLL_MODE, 39, 81
tbsPoll, 29, 30, 82, 94
interrupt status registers, 82
pollISR, 39, 40, 49

PRBS
configuring
CFG_PRBS, 53, 54, 78, 79
PRBS generator, 78
PRBS generator and monitor STS-1 configuration parameters, 54
PRBS monitor, 79
configuring and retrieving payload for PRBS processor, 77
forcing bit error in PRBS generator, 79
functions
cfgEgPrbsGen, 40, 41
cfgEgPrbsMon, 40, 41
cfgIgPrbsGen, 40, 41
cfgIgPrbsMon, 40, 41
tbsPrbsForceBitErr, 36, 79
tbsPrbsGenCfg, 36, 78
tbsPrbsMonCfg, 37, 79
tbsPrbsResync, 37, 80
monitor status block, 53

PRBS generator and monitor configuration block, 53, 54
processor events, 47
processors, 17, 23, 77
resynchronizing PRBS monitor, 80

preemption, 101
disabling, 101
functions
sysTbsPreemptDisable, 101
sysTbsPreemptEnable, 101
re-enabling, 101

PYLD, 41, 43, 77

R

read
functions
tbsRead, 62
tbsReadBlock, 63, 64
tbsReadIndirect, 65

registers
functions
sysTbsReadReg, 106, 110, 111

rtos
files
tbs_rtos.c, 105, 112, 113
tbs_rtos.h, 105, 107, 112, 113

RTOS, 20
interface, 96
structures passed through RTOS buffers, 55

RWPM, 31

S

semaphores, 99
creating a semaphore, 99
deleting a semaphore, 100
functions
sysTbsSemCreate, 99
sysTbsSemDelete, 100
sysTbsSemGive, 100
sysTbsSemTake, 100
giving a semaphore, 100
taking a semaphore, 100

sleeping a task, 99

stateDevice, 38, 49, 56

stateModule, 38, 48, 56

statistics
block, 50
event and statistics counter structure, 50
functions
tbsClrStats, 35, 84
tbsGetStats, 35, 84
getting
cumulative device statistics, 84
delta statistics counter of device, 85

status
functions
tbsGetStatus, 35, 85
STATUS_IO, 52
STATUS_PRBS, 53
STATUS_RX8D, 53
sTBS_STATUS, 52, 53, 85

T

telecombus
enabling
incoming to outgoing parallel telecombus loopbacks, 88
outgoing to incoming parallel telecombus loopbacks, 88
receive to transmit serial telecombus loopbacks, 89
transmit to receive serial telecombus loopbacks, 89

testing
functions
tbsTestRAM, 88
tbsTestReg, 87
RAM accesses, 88
register accesses, 87

time slot, 16, 21, 23, 32, 54, 67, 122

timers, 99

  functions
    sysTbsTimerSleep, 99, 106

transmit disparity encoder, 17

TSI

  connection map, 44
  connection page, 44
  space-time slot, 45

W

write

  functions
    sysTbsWrite, 63, 64, 65
    sysTbsWriteReg, 106
    tbsWrite, 63
    tbsWriteBlock, 64
    tbsWriteIndirect, 65