PM8315, PM5365, PM4328

TEMUX/TEMAP/TECT3

T1/E1 FRAMER, VT/TU MAPPER, M12/M13 MUX

DRIVER MANUAL

PROPRIETARY AND CONFIDENTIAL

RELEASE

ISSUE 2: AUGUST, 2001

ABOUT THIS MANUAL AND TEMUX/TEMAP/TECT3

This manual describes the device driver for the TEMUX/TEMAP/TECT3 devices. Since the TEMAP and TECT3 devices contain a subset of the features provided by the TEMUX device, the same driver is used for all. Briefly the differences between the devices is as follows, please refer to the devices respective data sheets for more information.

Table 1: Device Differences

Device	28 T1, 21 E1 Framers	VT1.5/VT2/TU-11/TU-12 Sonet/SDH Mapping	M13 MUX with DS3 Framer	Sonet/SDH DS3 Mapper
TEMUX	Yes	Yes	Yes	Yes
TEMAP	No	Yes	Yes	Yes
TECT3	Yes	No	Yes	No

This manual describes the driver's functions, data structures, and architecture. It focuses on the driver's interfaces to your application, Real-Time Operating System, and to the TEMUX/TEMAP/TECT3 device. It also describes in general terms how to modify and port the driver to your software and hardware platform.

Audience

This manual is written for people who need to:

- Evaluate and test the TEMUX/TEMAP/TECT3 devices
- Modify and add to the TEMUX/TEMAP/TECT3 driver's functions
- Port the TEMUX/TEMAP/TECT3 driver to a particular platform.

References

For more information about the TEMUX/TEMAP/TECT3 driver, see the driver's release notes. For more information about the TEMUX/TEMAP/TECT3 device, see the documents listed in the table below and any related errata documents.

Table 2: Related Documents

Document Number	Document Name
PMC-1981411	TEMAP High Density VT/TU Mapper and M13 Multiplexer Short Form Data Sheet
PMC-1981125	High Density T1/E1 Framer with Integrated VT/TU Mapper and M13 Multiplexer Telecom Standard Product Data Sheet
PMC-2011596	High Density T1/E1 Framer with Integrated M13 Multiplexer Telecom Standard Product Datasheet

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	February 2000	Document created
Issue 2	July 2001	Added documentation to support TEMAP and TECT3 devices. Added more detail to porting section. Added init profile information and some missing API.

Legal Issues

PMC-1991611 (P1), ref PMC-990551 (P1)

The technology discussed is protected by one or more of the following Patents:

U.S. Patent No. 5,835,545 5,973,977 5,343,482

Relevant patent applications and other patents may also exist.

Contacting PMC-Sierra

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

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

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

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

The following sections of the TEMUX/TEMAP/TECT3 driver manual describe the TEMUX/TEMAP/TECT3 device driver. The code provided throughout this document is written in the ANSI C language. This has been done to promote greater driver portability to other embedded hardware and Real-Time Operating System environments.

This driver can be used for the TEMUX/TEMAP/TECT3 devices. See Table 1 for a brief description of the differences between these devices. To properly support the TEMAP device, use the compile switch DEV_IS_TEMAP during compilation. To properly support the TECT3 device, use the compile switch DEV_IS_TECT3 during compilation.

Section 2 of this document, Software Architecture, defines the software architecture of the TEMUX/TEMAP/TECT3 device driver by including a discussion of the driver's external interfaces and its main components. The Data Structure information in Section 4 describes the elements of the driver that either configure or control its behavior. Included here are the constants, variables, and structures that the TEMUX/TEMAP/TECT3 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 TEMUX/TEMAP/TECT3 driver Application Programming Interface (API). This section outlines: (1) function calls that hide device-specific details and (2) application callbacks that notify the user of significant device events.

For your convenience, this manual provides a brief guide to porting the TEMUX/TEMAP/TECT3 device driver to your hardware and RTOS platform (page 87). In addition, an Appendix (beginning on page 92) and Index (page 101), provide you with useful reference information.

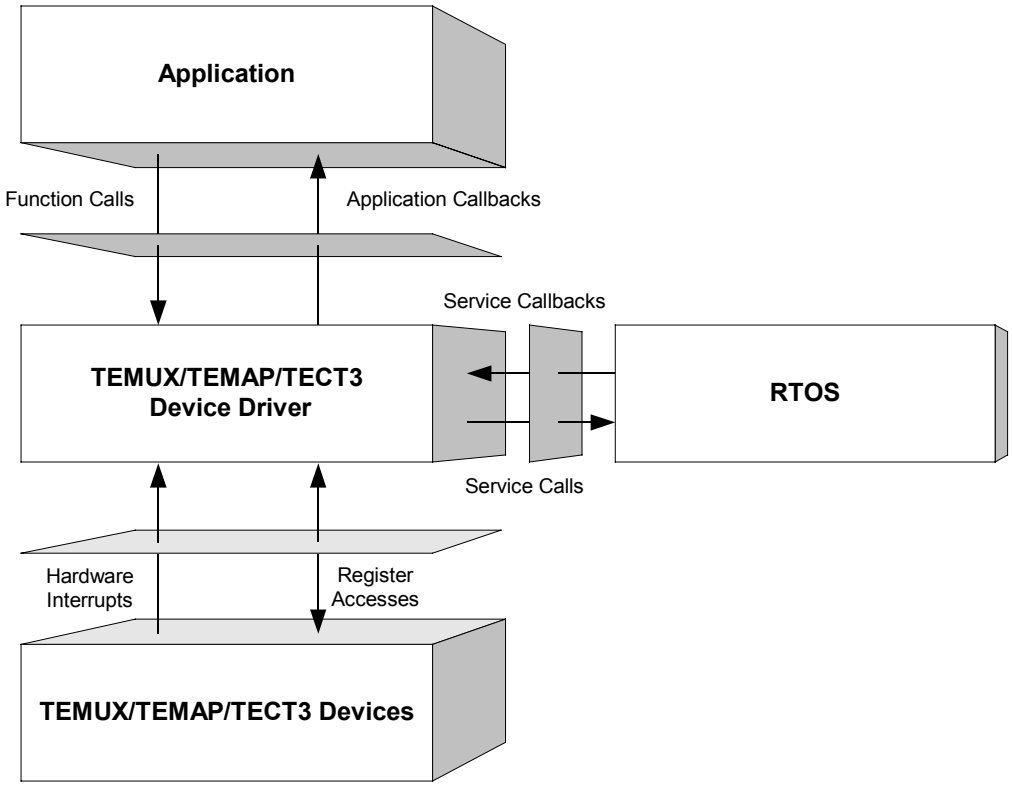
2 SOFTWARE ARCHITECTURE

This section describes the software architecture of the TEMUX/TEMAP/TECT3 device driver. This includes a discussion of the driver's external interfaces and its main components.

2.1 Driver External Interfaces

Figure 1 illustrates the external interfaces defined for the TEMUX/TEMAP/TECT3 device driver.

Figure 1: Driver Interfaces



Application Programming Interface

The driver's API is a collection of high level functions that can be called by application programmers to configure, control, and monitor the TEMUX/TEMAP/TECT3 device, such as:

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

The driver API functions use the driver library functions as building blocks to provide this system level functionality to the application programmer. The driver API also consists of callback functions that notify the application of significant events that take place within the device.

Real-Time RTOS Interface

The driver's RTOS interface module provides functions that let the driver use RTOS services. The RTOS interface functions perform the following tasks for the TEMUX/TEMAP/TECT3 driver:

- Allocate and de-allocate memory
- Manage buffers for the ISR and DPR
- Timer management
- Synchronization management
- Task management

You must modify the RTOS interface code to suit your RTOS environment.

Driver Hardware Interface

The TEMUX/TEMAP/TECT3 hardware interface provides functions that read from and write to device-registers. The hardware interface also provides a template for an ISR that the driver calls when the device raises a hardware interrupt. You must modify this function based on the interrupt configuration of your system.

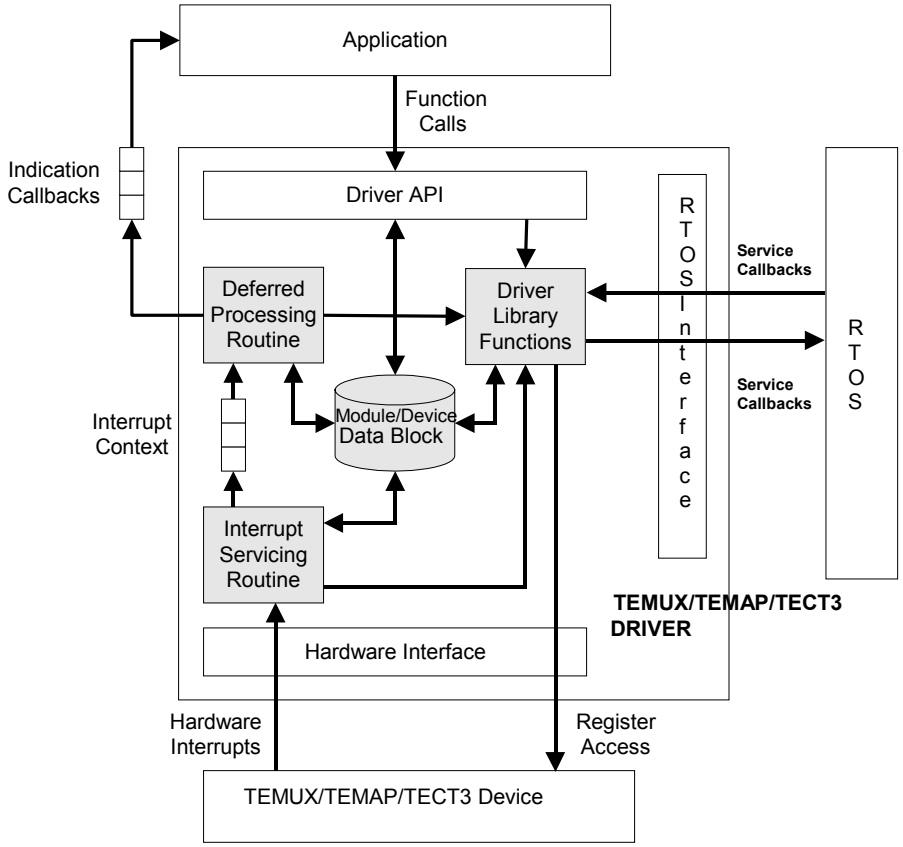
2.2 Main Components

Figure 2 illustrates the top level architectural components of the TEMUX/TEMAP/TECT3 device driver. This applies to 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 five main modules:

- Module and Device management
- Driver API
- Driver library
- Interrupt processing

Figure 2: Driver Architecture



Module and Device Management

Module Data Block (MDB)

The Module Data Block (MDB) and Module Status Block (MSB) are the top layer data structures, created by the TEMUX/TEMAP/TECT3 device driver to keep track of its initialization and operating parameters, modes and dynamic data. The MDB is allocated via an RTOS call, when the driver is first initialized and contains the MSB and all the Device Structures.

Device Data Block (DDB)

The Device Data Block (DDB) is contained in the MDB and is allocated when the module is opened. The DDB contains context and status information for each TEMUX/TEMAP/TECT3 device that the driver manages. It is initialized when a device is added to the module.

The DDB stores context information about the TEMUX/TEMAP/TECT3 device, such as:

- Device state
- Control information
- Initialization vector

- Callback function pointers
- Statistical counts

Driver API

The driver API consists of functions used to configure and monitor the various subsystems in the TEMUX/TEMAP/TECT3 device. The API functions are divided into following sections:

Alarms, Status, and Statistics Section

This section is responsible for monitoring alarms, tracking devices status information and retrieving performance and error statistics for each device registered with (added to) the driver.

Diagnostics Section

This section is responsible for providing access to the diagnostic capabilities of the TEMUX/TEMAP/TECT3 devices. Functions are provided to various loopback modes and to test register accesses.

Device Read/Write Section

This section provides read/write access functions to the various sub-blocks of the TEMUX/TEMAP/TECT3 devices. Functions are provided to write to the T1/E1 framer block, SONET/SDH Mapper block, and the DS3 Mux/Demux block. Functions are also provided to read a block of registers and access the indirect registers.

Driver Library

The driver library module is a collection of low-level utility functions that manipulate the device registers and the contents of the driver's DDB. The driver library functions serve as building blocks for higher level functions that constitute the driver API module. Application software does not normally call the driver library functions.

Interrupt Processing

The TEMUX/TEMAP/TECT3 driver provides an ISR called `temuxISR` that checks if there are any valid interrupt conditions present for the device. This function can be used by a system-specific interrupt-handler function to service interrupts raised by the device. Its main purpose is to collect information about the current interrupt condition of the device and pass this information along to the Deferred-Processing Routine for actual processing.

The low-level interrupt-handler function that traps the hardware interrupt and calls `temuxISR` is system and RTOS dependent. Therefore, it is outside the scope of the driver. Example implementations of an interrupt handler and functions that install and remove it are provided as a reference on page 69. You can customize these example implementations to suit your specific needs.

Deferred-Processing Routine Module

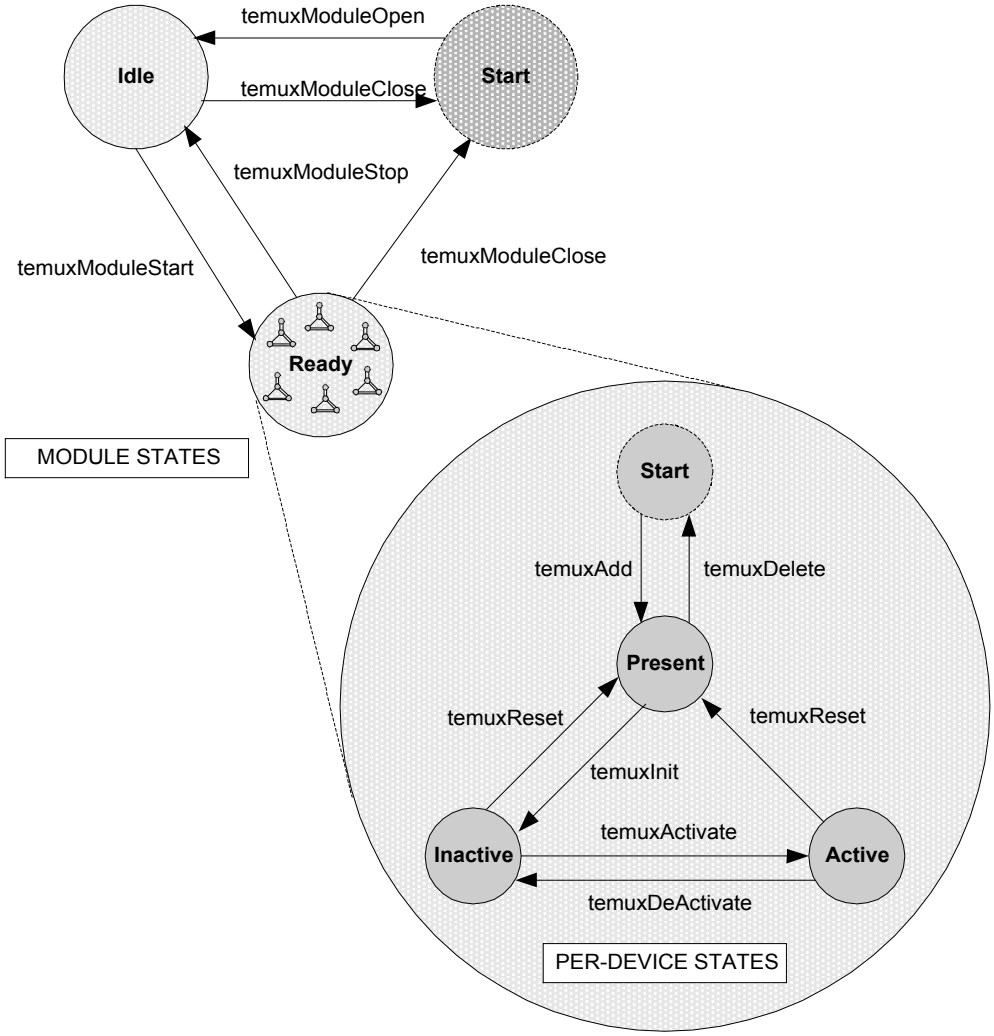
The DPR provided by the driver (`temuxDPR`) clears and processes interrupt conditions for the device. Typically, a system specific function, which runs as a separate task within the RTOS, executes the DPR.

See page 22 for a detailed explanation of the DPR and interrupt-servicing model.

3 SOFTWARE STATE DESCRIPTION

Figure 3 shows the software state diagrams for the TEMUX/TEMAP/TECT3 module and device(s) as maintained by the driver

Figure 3: State Diagram



State transitions are made on the successful execution of the corresponding transition routines shown. State information helps maintain the integrity of the MDB and the DDB(s) by controlling the set of operations that are allowed in each state.

3.1 Module States

The following is a description of the TEMUX/TEMAP/TECT3 driver module states.

Start: TMX_MOD_START

The TEMUX/TEMAP/TECT3 driver module has not been initialized. The only API function that will be accepted in this state is `temuxModuleOpen`. In this state the driver does not hold any RTOS resources (memory, timers, etc), has no running tasks, and performs no actions.

Idle: TMX_MOD_IDLE

The TEMUX/TEMAP/TECT3 driver module has been initialized successfully via the API function `temuxModuleOpen`. 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. The only API functions that will be accepted in this state are `temuxModuleStart` and `temuxModuleClose`.

Ready: TMX_MOD_READY

This is the normal operating state for the driver module. This means that the RTOS resources have been allocated and the driver is ready for devices to be added. In order to get to this state, the API function `temuxModuleStart` is called (this function is responsible for creating and/or allocating all of the RTOS resources necessary for the proper operation of the module and the devices). The API functions accepted here for module control are `temuxModuleStop` and `temuxModuleClose`. The driver module remains in this state while devices are in operation. Devices can be added via `temuxAdd`.

3.2 Device States

Once the driver module has progressed into the READY state, the user can begin to add individual devices into operation. The driver module remains in the READY state while devices are in operation. Devices can be added via `temuxAdd`. The module functions `temuxModuleStop` and `temuxModuleClose` are always available in this state (and therefore not mentioned below) and if used, will cause each and every device (that is not in the START state) to be deleted, before that module function is fully executed.

The following is a description of the TEMUX/TEMAP/TECT3 driver device states:

Start: TMX_START

The TEMUX/TEMAP/TECT3 driver device has not been initialized. The only API function that will be accepted in this state is `temuxAdd`. In this state, the device is unknown by the driver and performs no actions. There is a separate flow for each device that can be added and they all start here.

Present: TMX_PRESENT

This device state is a quiet state for the device. In order to get to this state, the API needs to be called by one of two functions:

- `temuxAdd`: Responsible for verifying the presence of the device and for initializing the data structures associated with this device.
- `temuxReset`: De-activates the device and restores the device's data structures to the initialized condition.

In this state, devices can be initialized via `temuxInit` or deleted via `temuxDelete`.

Active: TMX_ACTIVE

This is the normal operating state for the device(s). State changes can be initiated from the active state via `temuxDelete`, `temuxDeActivate` and `temuxReset`.

Inactive: TMX_INACTIVE

This state is entered via the `temuxDeActivate` or `temuxInit` function calls. In this state the device remains configured but all data functions are de-activated including interrupts and status, alarms, and counter functions. `temuxActivate` will return the device to the active state, while `temuxReset` or `temuxDelete` will de-configure the device.

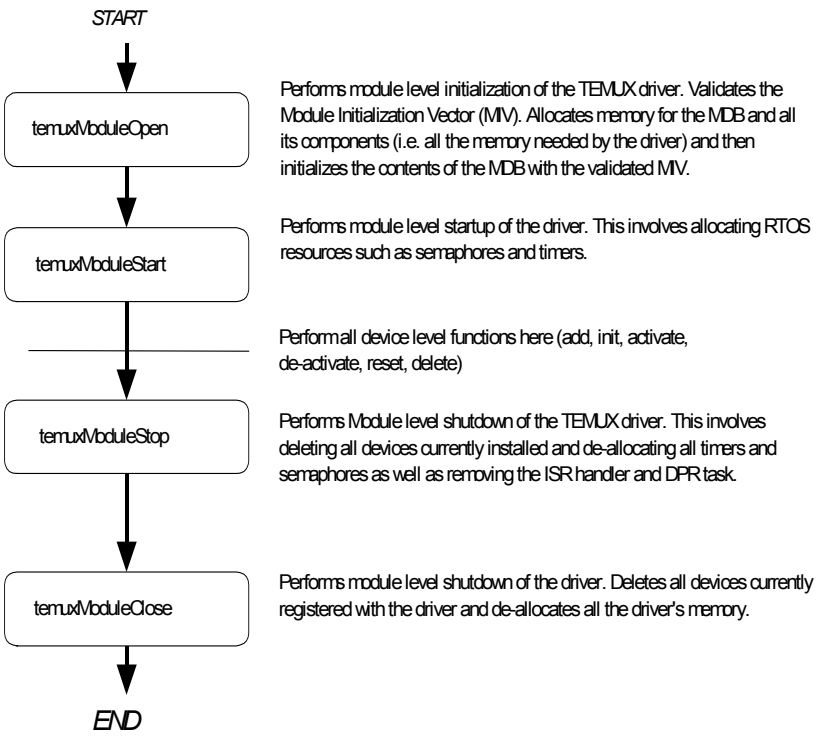
3.3 Processing Flows

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 flow diagram illustrates the typical function call sequences that occur when initializing or shutting down the TEMUX/TEMAP/TECT3 driver module.

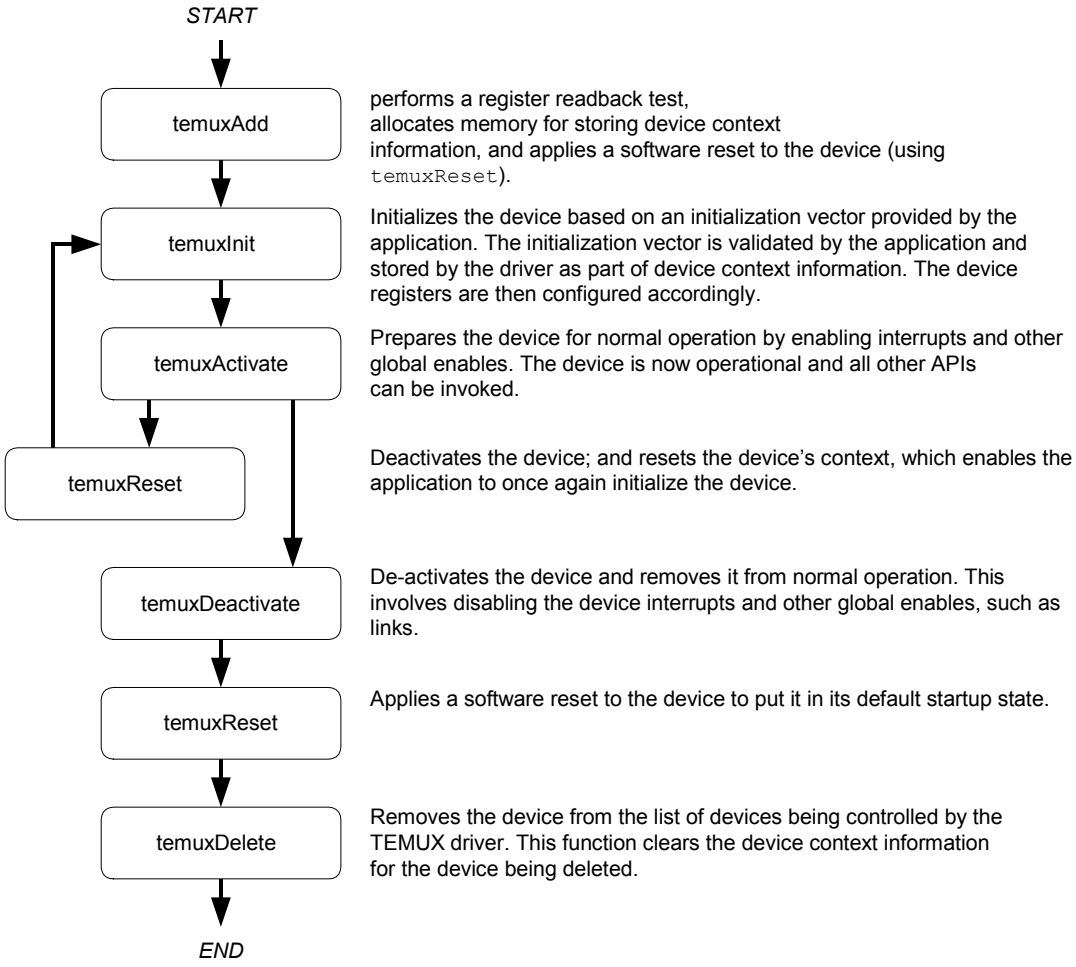
Figure 4: Module Management Flow Diagram



Device Management

The following figure shows the functions and process that the driver uses to add, initialize, re-initialize, or delete devices.

Figure 5: Device Management Flow Diagram



3.4 Interrupt Servicing

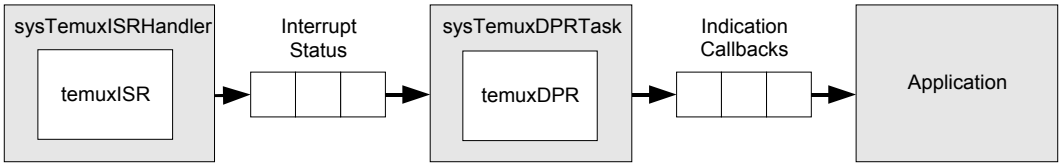
The TEMUX/TEMAP/TECT3 driver services device interrupts using an Interrupt-Service Routine (ISR) that traps interrupts and a deferred interrupt-processing routine (DPR) that actually processes the interrupt conditions. This lets the ISR execute quickly and exit. Most of the time-consuming processing 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 TEMUX/TEMAP/TECT3 driver.

The driver provides system-independent functions, `temuxISR` and `temuxDPR`. You must fill in the corresponding system-specific functions, `sysTemuxISRHandler` and `sysTemuxDPRTask`. The system-specific functions isolate the system-specific communication mechanism (between the ISR and DPR) from the system-independent functions, `temuxISR` and `temuxDPR`.

Figure 6 illustrates the interrupt service model used in the TEMUX/TEMAP/TECT3 driver design.

Figure 6: Interrupt Service Model



Note: Instead of using an interrupt service model, you can use a polling service model in the TEMUX/TEMAP/TECT3 driver to process the device's event-indication registers.

Calling temuxISR

An interrupt handler function, which is system dependent, must call `temuxISR`. But first, the low-level interrupt-handler function must trap the device interrupts. You must implement this function for your system.

The interrupt handler that you implement (`sysTemuxISRHandler`) is installed in the interrupt vector table of the system processor. Then it is called when one or more TEMUX/TEMAP/TECT3 devices interrupt the processor. The interrupt handler then calls `temuxISR` for each device in the active state that requires service.

The `temuxISR` function reads from the master interrupt-status registers and the miscellaneous interrupt-status registers of the TEMUX/TEMAP/TECT3. Then `temuxISR` returns with this status information if valid status bits are set. The `temuxISR` also clears those status bits, which in turn clears the initial cause of the interrupt. The `sysTemuxISRHandler` function then sends a message to the DPR task (for each device that requested service) which contains the valid interrupt status bits and the device's context handle.

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

Calling temuxDPR

The `sysTemuxDPRTask` function is a system specific function that runs as a separate task within the RTOS. You should set the DPR task's priority higher than the application task(s) interacting with the TEMUX/TEMAP/TECT3 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, `sysTemuxDPRTask` calls the DPR (`temuxDPR`).

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

Typically, you should implement these callback functions as simple message posting functions that post messages to an application task. However, you can implement the indication callback to perform processing within the DPR task context and return without sending any messages. In this case, ensure that the indication function does not call any API functions that change the driver's state, such as `temuxDelete`. Also, ensure that the indication function is non-blocking. You can customize these callbacks to suit your system.

Note: Since the `temuxISR` and `temuxDPR` 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, `sysTemuxISRHandler` and `sysTemuxDPRTask`. When the driver calls `sysTemuxISRHandlerInstall` for the first time, the driver installs `sysTemuxISRHandler` in the interrupt vector table of the processor. The `sysTemuxDPRTask` function is spawned as a task when `sysTemuxDPRTaskInstall` is called.

The `sysTemuxISRHandlerInstall` function also creates the communication channel between `sysTemuxISRHandler` and `sysTemuxDPRTask`. This communication channel is most commonly a message queue associated with the `sysTemuxDPRTask`.

Similarly, during removal of interrupts, the driver removes `sysTemuxISRHandler` from the microprocessor's interrupt vector table when `sysTemuxDPRTaskRemove` is called.

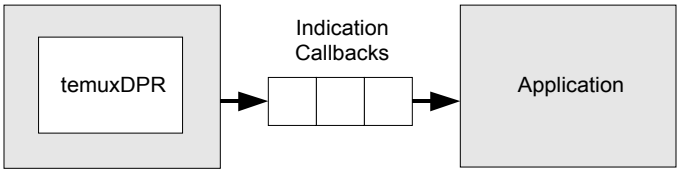
As a reference, this manual provides example implementations of the interrupt installation and removal functions on page 81. You can customize these prototypes to suit your specific needs.

3.5 Polling Servicing

Instead of using an interrupt service model, you can use a polling service model in the TEMUX/TEMAP/TECT3 driver to process the device's event-indication registers.

Figure 7 illustrates the polling service model used in the TEMUX/TEMAP/TECT3 driver design.

Figure 7: Polling Service Model



The polling service code includes some system specific code (prefixed by "`sysTemux`"), which typically you must implement for your application. The polling service code also includes some system independent code (prefixed by "`temux`") provided by the driver that does not change from system to system.

In polling mode, `sysTemuxISRHandler` and `temuxISR` are not used. When `temuxPoll` is called, the message normally sent to the DPR is now passed internally.

The nature of this processing can differ from system to system. Therefore, the DPR calls different indication callbacks for different interrupt conditions. You can customize these callbacks to fit your application's specific requirements. See page 75 for a description of these callback functions.

4 DATA STRUCTURES

4.1 Constants

The file `temux.h` defines the lowest level (compile time) items, generally those that are needed by any application code in order to interface with the driver. All source and header files include this file. The compile constants are defined in the file `temux.h` as `UINT1, UINT2, UINT4`, etc.

The file `tmx_api.h` defines the set of (run time) items needed to interface directly with the core API functions. The following constants are defined in the file `tmx_api.h`.

- `TMX_MAX_DEVICES`: 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
- `<TEMUX ERROR CODES>`: error codes used throughout the driver code, returned by the API functions and (when `TMX_FAIL` is returned to a function's caller) used in the global error number field of the MDB or DDB
- `sTMX_MIV`: structure passed by the application into the `temuxModuleOpen` function call.
- `sTMX_DIV`: structure passed by the application into the `temuxAdd` function call.

The remaining files should only be needed for extended interfaces that make use of the internal structures or functions of the driver.

4.2 Data Structures

The following are the main data structures employed by the TEMUX/TEMAP/TECT3 driver.

4.3 Structures Passed by the Application

These structures are defined for use by the application and are passed by reference to functions within the driver.

Module Initialization Vector

Passed via the `temuxModuleOpen` call, this structure contains all the information needed by the driver to initialize the module. Special or unusual fields are described first:

- `pMDB`: can be used by the application to pass the address of a pre-allocated MDB. If `pMDB` is NULL, the driver will allocate sufficient memory to hold the MDB and return its address in the `pMDB` field.

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

Table 3: Module Initialization Vector: sTMX_MIV

Field Name	Field Type	Field Description
autoStart	UINT2	indicates to driver to start the module when opened
pMDB	void *	(pointer to) pre-allocated or (if NULL) returned MDB
maxDevs	UINT2	number of devices that must be supported for this session

Device Initialization Vector

Passed via the temuxAdd call, this structure contains all the information needed by the driver to initialize a TEMUX/TEMAP/TECT3 device. Special or unusual fields are described first:

- pDDB can be used by the application to pass the address of a pre-allocated DDB. If pDDB is NULL, the driver will allocate sufficient memory to hold the DDB and return its address in the pDDB field.
- baseAddress: must contain the hardware base address of the device.
- usrCtxt: this field is strictly a user field. The value passed into the function via this element will be stored in the DDB and passed back to the application during DPR processing. The user might use it to identify 'this' device or point to some data related to this device.
- autoInit: is a flag that tells the driver to automatically initialize the device being added (calling temuxInit internally does this). If the flag is zero, the DDB will be initialized and the device left uninitialized, and the application will have to call temuxInit at a later time.
- profileNum: is used only when autoInit is set and indicates which mode the device should be initialized into. The function of this element is the same as the profileNum argument to the function temuxInit. A value of zero indicates that during initialization, after the device is reset, all registers should remain unchanged (in their initial state).
- cbackIO, cbackDS3, cbackFramer, cbackMapper: Passes the addresses of application functions used by the DPR to inform the application code of pending events. If the user sets the element to NULL, then any events that might cause the DPR to 'call back' to the application will be processed during ISR processing but ignored by the DPR.

Table 4: Device Initialization Vector: sTMX_DIV

Field Name	Field Type	Field Description
pDDB	void *	(pointer to) pre-allocated or (if NULL) returned DDB
baseAddress	UINT1 *	device base address

Field Name	Field Type	Field Description
usrCtxt	void *	a user-supplied value that is returned in callback functions
autoInit	UINT2	if non-zero, temuxInit is called internally
profileNum	UINT2	profile number to be used for initialization. A profile number of zero indicates that the driver should leave all the device registers unchanged after reset
modeISR	TMX_ISR_MODE	indicates the type of ISR/polling to do
cbackIO	void *	address of the callback function for IO Events
cbackDS3	void *	address of the callback function for DS3 Events
cbackFramer	void *	address of the callback function for Framer Events
cbackMapper	void *	address of the callback function for Mapper Events (not valid in TECT3)

Device Initialization Profile

The device initialization profile is used to initialize the TEMUX/TEMAP/TECT3 device to a specific operating mode. It is used by the profile manipulation functions (temuxGetInitProfile, etc.) and temuxInit. Important fields are given below.

Table 5: Device Initialization Profile: sTMX_INIT_PROF

Field Name	Field Type	Field Description
lineopt	UINT2	Line side configuration options
sysopt	UINT2	System side configuration options
opmode	UINT2	Operation mode
e1Mode	BOOLEAN	Set device to E1 mode
autoActivate	BOOLEAN	Activates the device for operation

lineopt can be one of the following:

- TMX_LINEOPT_LIU_DS3 : DS3 Mux with serial LIU interface
- TMX_LINEOPT_SDH_DS3: DS3 Mux with DS3 SONET/SDH Mapper (Not valid in TECT3)
- TMX_LINEOPT_SDH_E1T1: E1/T1 Mapper

sysopt can be one of the following:

- TMX_SYSOPT_CDATA: Serial Clock/Data Interface
- TMX_SYSOPT_MVIP: H-MVIP interface (not valid in TEMAP)
- TMX_SYSOPT_SBI: SBI Interface
- TMX_SYSOPT_SBI_CCS: SBI with CAS or CCS H-MVIP (not valid in TEMAP)
- TMX_SYSOPT_CDATA_CCS: Serial Clock/Data with CCS H-MVIP (not valid in TEMAP)

opmode can be one of the following:

- TMX_OPMODE_FRAMER: High Density Framer Mode (not valid in TEMAP)
- TMX_OPMODE_MAPPER: Mapper/Multiplexor Mode (not valid in TECT3)
- TMX_OPMODE_TRANSMUX: TransMux mode (not valid in TECT3)
- TMX_OPMODE_DS3_ONLY: DS3 Framer Only

Preset Profiles

There are 8 preset profiles (indexed 0-7) that can be used to initialize the device during the call to temuxInit. They are set to the values shown in Table 6.

Table 6: Preset Profiles: temuxInit

Profile Number	lineopt	sysopt	opmode	e1Mode
0	TMX_LINEOPT_LIU_DS3	TMX_SYSOPT_SBI	TMX_OPMODE_FRAMER	FALSE
1	TMX_LINEOPT_SDH_E1T1	TMX_SYSOPT_CDATA	TMX_OPMODE_FRAMER	TRUE
2	TMX_LINEOPT_SDH_E1T1	TMX_SYSOPT_MVIP	TMX_OPMODE_FRAMER	FALSE
3	TMX_LINEOPT_SDH_E1T1	TMX_SYSOPT_SBI	TMX_OPMODE_FRAMER	FALSE
4	TMX_LINEOPT_LIU_DS3	TMX_SYSOPT_CDATA	TMX_OPMODE_FRAMER	FALSE
5	TMX_LINEOPT_LIU_DS3	TMX_SYSOPT_CDATA	TMX_OPMODE_MAPPER	FALSE
6	TMX_LINEOPT_LIU_DS3	TMX_SYSOPT_CDATA	TMX_OPMODE_DS3_ONLY	FALSE
7	TMX_LINEOPT_LIU_DS3	TMX_SYSOPT_CDATA	TMX_OPMODE_TRANSMUX	FALSE

Profiles 0-4 are not valid with the TEMAP device. Profiles 1-3, 5, 7 are not valid in the TECT3 device.

Interrupt-Service Routine Mask Vector

Passed via the `temuxClearMask`, `temuxGetMask` and the `temuxSetMask` calls, this structure contains all the information needed by the driver to enable and disable any of the interrupts in the TEMUX/TEMAP/TECT3.

Table 7: ISR Mask Vector: sTMX_MASK

Field Name	Field Type	Sets / Clears Interrupt Condition
`sdet0e`	`UINT1`	SBI collision detect (register 0)
`sdet1e`	`UINT1`	SBI collision detect (register 1)
`io`	`struct tmx_mask_io`	IO Section interrupts
`ds3`	`struct tmx_mask_ds3`	DS3 Section interrupts
`mux[7]`	`struct tmx_mask_mux`	MUX/MX12/DS2 Section interrupts
`framer[28]`	`struct tmx_mask_framer`	FRAMER Section interrupts
`mapper`	`struct tmx_mask_mapper`	MAPPER Section interrupts (not valid in TECT3)

Mask Sub-structures

These structures also appear in sTMX_MASK (above).

Table 8: IO Section Masks: struct tmx_mask_io

Field Name	Field Type	Field Description
`exsbi.ovre`	`UINT1`	SBI bus egress overrun
`exsbi.unde`	`UINT1`	SBI bus egress underrun
`insbi.ovre`	`UINT1`	SBI bus ingress overrun
`insbi.unde`	`UINT1`	SBI bus ingress underrun
`exsbi.pare`	`UINT1`	SBI bus parity

Table 9: DS3 Section Masks: struct tmx_mask_ds3

Field Name	Field Type	Field Description
pmon.inte	UINT1	DS3 PMON Accumulation Transfer
rdlc.inte	UINT1	DS3 Receive DLC State Transition
rboc.idle	UINT1	DS3 Receive BOC FEAC Removed (not valid in TEMAP)
rboc.feace	UINT1	DS3 Receive BOC FEAC Detected
frmr.cofae	UINT1	DS3 Framer Change of Frame Alignment
frmr.rede	UINT1	DS3 Framer RED Alarm Transition
frmr.cbite	UINT1	DS3 Framer C-Bit Transition
frmr.ferfe	UINT1	DS3 Framer Far End Receive Failure Transition
frmr.idle	UINT1	DS3 Framer Idle Signal Transition
frmr.aise	UINT1	DS3 Framer AIS Signal Transition
frmr.oofe	UINT1	DS3 Framer Out of Frame Transition
frmr.lose	UINT1	DS3 Framer Loss of Signal Transition
tdpr.fulle	UINT1	DS3 Transmit DLC Fifo Full
tdpr.ovre	UINT1	DS3 Transmit DLC Fifo Overrun
tdpr.unde	UINT1	DS3 Transmit DLC Fifo Underrun
tdpr.lfille	UINT1	DS3 Transmit DLC Empty (or Low Water Mark)
mx23.inte	UINT1	MX23 Loopback Request Detected
prgd.synce	UINT1	DS3 PRGD Synchronization Transition
prgd.bee	UINT1	DS3 PRGD Bit Error Detected
prgd.xfere	UINT1	DS3 PRGD Accumulation Transfer

Table 10: MUX Section Masks: struct tmx_mask_mux

Field Name	Field Type	Field Description
ds2.inte.cofae	UINT1	DS2 Change of Frame Alignment Detected

Field Name	Field Type	Field Description
ds2.inte.rede	UINT1	DS2 Red Alarm Transition
ds2.inte.ferfe	UINT1	DS2 Far End Receive Failure Transition
ds2.inte.aise	UINT1	DS2 AIS Signal Transition
ds2.inte.oofe	UINT1	DS2 Out of Frame Transition
ds2.intr.inte	UINT1	DS2 Error Counter Transfer
mx12.inte	UINT1	MX12 Loopback Request Detected

Table 11: Framer Section Masks: struct tmx_mask_framer

Field Name	Field Type	Field Description
rjat.ovre	UINT1	T1/E1 Framer RJAT Overrun
rjat.unde	UINT1	T1/E1 Framer RJAT Underrun
tjat.ovre	UINT1	T1/E1 Framer TJAT Overrun
tjat.unde	UINT1	T1/E1 Framer TJAT Underrun
tx_elst.slipe	UINT1	T1/E1 Framer TX_ELST Slip (not valid in TEMAP)
sig_elst.slipe	UINT1	T1/E1 Framer Signaling ELST Slip (not valid in TEMAP)
rx_elst.slipe	UINT1	T1/E1 Framer RX_ELST Slip (not valid in TEMAP)
sigx.sige	UINT1	Change of signaling state (COS) (not valid in TEMAP)
t1_frmr.cofae	UINT1	T1 Framer Change of Frame Alignment Detected
t1_frmr.fere	UINT1	T1 Framer Framing Bit Error Detected
t1_frmr.beee	UINT1	T1 Framer Bit Error Detected
t1_frmr.sfee	UINT1	T1 Framer Severely Errored Framing Event Detected
t1_frmr.mfpe	UINT1	T1 Framer Framing Bit Mimic Detected
t1_frmr.infre	UINT1	T1 Framer is Inframe
e1_frmr.c2nciwe	UINT1	E1 Framer CRC to Non-CRC Interworking Mode Transition

Field Name	Field Type	Field Description
e1_frmr.oofe	UINT1	E1 Framer Out of Frame Transition
e1_frmr.oosmfe	UINT1	E1 Framer Out of Signaling Multiframe Transition
e1_frmr.oocmfe	UINT1	E1 Framer Out of CRC Multiframe Transition
e1_frmr.cofae	UINT1	E1 Framer Change of Frame Alignment Detected
e1_frmr.fere	UINT1	E1 Framer Framing Bit Error Detected
e1_frmr.smfere	UINT1	E1 Framer Signaling Multiframe Framing Bit Error Detected
e1_frmr.cmfere	UINT1	E1 Framer CRC Multiframe Framing Bit Error Detected
e1_frmr.raie	UINT1	E1 Framer Remote Alarm Indication Transition
e1_frmr.rmaie	UINT1	E1 Framer Remote Multiframe Alarm Indication Transition
e1_frmr.aisde	UINT1	E1 Framer Alarm Indication
e1_frmr.rede	UINT1	E1 Framer Red Alarm Transition
e1_frmr.aise	UINT1	E1 Framer AIS
e1_frmr.febee	UINT1	E1 Framer Far End Bit Error Detected
e1_frmr.crcee	UINT1	E1 Framer CRC Error Detected
e1_frmr.sa4e	UINT1	E1 Framer National Use Bit Sa4 Transition
e1_frmr.sa5e	UINT1	E1 Framer National Use Bit Sa5 Transition
e1_frmr.sa6e	UINT1	E1 Framer National Use Bit Sa6 Transition
e1_frmr.sa7e	UINT1	E1 Framer National Use Bit Sa7 Transition
e1_frmr.sa8e	UINT1	E1 Framer National Use Bit Sa8 Transition
e1_frmr.ooofe	UINT1	E1 Framer Out of Offline Frame
e1_frmr.raiccrce	UINT1	E1 Framer RAI & CRC Detected
e1_frmr.cfebee	UINT1	E1 Framer Continuous FEBE Detected
e1_frmr.v52linke	UINT1	E1 Framer V52 Link Detected

Field Name	Field Type	Field Description
e1_frmr.ifpe	UINT1	E1 Framer Input Frame Pulse Asserted
e1_frmr.icsmfpe	UINT1	E1 Framer Input CRC Sub-Multiframe Pulse Asserted
e1_frmr.icmfpe	UINT1	E1 Framer Input CRC Multiframe Pulse Asserted
e1_frmr.ismfpe	UINT1	E1 Framer (Rx) Input Signaling Multiframe Pulse Asserted
e1_tran.sigmfe	UINT1	E1 Framer (Tx) Signaling Multiframe Boundry Detected (not valid in TEMAP)
e1_tran.nfase	UINT1	E1 Framer (Tx) NFAS Frame Boundary Detected (not valid in TEMAP)
e1_tran.mfe	UINT1	E1 Framer (Tx) CRC-4 Multiframe Boundary Detected (not valid in TEMAP)
e1_tran.smfe	UINT1	E1 Framer (Tx) CRC-4 Sub-Multiframe Boundary Detected (not valid in TEMAP)
e1_tran.frme	UINT1	E1 Framer (Tx) Frame Boundary Detected (not valid in TEMAP)
pmon.inte	UINT1	E1/T1 Framer PMON Counter Transfer
aprm.inte	UINT1	T1 Framer APRM 1 Second Data Available (not valid in TEMAP)
prbs.synce	UINT1	T1 Framer PRBS Synchronization Transition
prbs.bee	UINT1	T1 Framer PRBS Bit Error Detected
prbs.xfere	UINT1	T1 Framer PRMS Accumulation Transfer (not valid in TEMAP)
tdpr.printe	UINT1	E1/T1 Framer Performance Report Ready (not valid in TEMAP)
tdpr.fulle	UINT1	E1/T1 Framer Fifo Full (not valid in TEMAP)
tdpr.ovre	UINT1	E1/T1 Framer Fifo Overrun (not valid in TEMAP)
tdpr.unde	UINT1	E1/T1 Framer Fifo Underrun (not valid in TEMAP)
tdpr.lfille	UINT1	E1/T1 Framer Fifo Empty (or Low Water Mark) (not valid in TEMAP)

Field Name	Field Type	Field Description
rdlc.inte	UINT1	E1/T1 Framer Receive DLC (not valid in TEMAP)
almi.yele	UINT1	E1/T1 Framer ALMI Yellow Alarm Detected
almi.rede	UINT1	E1/T1 Framer ALMI RED Alarm Detected
almi.aise	UINT1	E1/T1 Framer ALMI AIS Detected
rboc.idlee	UINT1	T1 Framer Receive BOC FEAC Removed (not valid in TEMAP)
rboc.boce	UINT1	T1 Framer Receive BOC FEAC Detected (not valid in TEMAP)

Table 12: Mapper Section Masks: struct tmx_mask_mapper (not valid in TECT3 device)

Field Name	Field Type	Field Description
icfg.ldpe	UINT1	Line Side Drop Parity Error
d3ma.oflie	UINT1	D3MA elastic store overflow
d3ma.uflie	UINT1	D3MA elastic store underflow
d3md.oflie	UINT1	D3MD elastic store overflow
d3md.uflie	UINT1	D3MD elastic store underflow
etpp[TUG2].pee[TU][1]	UINT1	Egress Tributary Payload Processor pointer event
etpp[TUG2].alarme[TU][1]	UINT1	Egress Tributary Payload Processor loss of pointer and path AIS
itpp[TUG2].pee[TU][1]	UINT1	Ingress Tributary Payload Processor pointer event
itpp[TUG2].alarme[TU][1]	UINT1	Ingress Tributary Payload Processor loss of pointer and path AIS
rtop[TUG2].pslue[TU][1]	UINT1	Receive Tributary Path Overhead Processor path signal label unstable
rtop[TUG2].pslme[TU][1]	UINT1	Receive Tributary Path Overhead Processor path signal label mismatch

Field Name	Field Type	Field Description
rtop[TUG2].copsle[TU][1]	UINT1	Receive Tributary Path Overhead Processor change of path signal label
rtop[TUG2].rfie[TU][1]	UINT1	Receive Tributary Path Overhead Processor remote failure indication
rtop[TUG2].rdie[TU][1]	UINT1	Receive Tributary Path Overhead Processor remote defect indication

Note

1. TUG2 refers to a range from 1 to 7 corresponding to TUG2 #1 to TUG2 #7 and TU refers to a range from 1 to 4 corresponding TU #1 to TU #4

Device Diagnostic Structures

Table 13: PRGD configuration: sTMX_PRGD

Field Name	Field Type	Field Description
enable	BOOLEAN	enable DS3 pseudo random number generator
control	UINT1	PRGD control register
ienable	UINT1	PRGD interrupt enable register
length	UINT1	PRGD length register
tap	UINT1	PRGD tap register
error	UINT1	PRGD error insertion register
pattern	UINT4	PRGD pattern insertion registers #1-4

Table 14: PRBS configuration: sTMX_PRBS

Field Name	Field Type	Field Description
enable	UINT1	Enable Framer PRBS generation
control	UINT1	PRBS generator/checker control register
ienable	UINT1	PRBS checker interrupt enable register
pattern	UINT1	PRBS pattern select register

4.4 Structures in the Driver's Allocated Memory

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

Module Data Block

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. Special or unusual elements in the MDB are described first, followed by the complete list of elements in table form.

- `errModule`: most of the module functions return a specific error code directly. When the returned code is `TMX_FAIL`, that indicates that the top level function was not able to carry the specific 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 to a `INT4` pointer and retrieve the local error code (this eliminates the need to include the MDB template into the application code).
- `modValid`: indicates that this structure has been properly initialized and may be read by the user.
- `modState`: contains the current state of the module and could be set to: `TMX_MOD_START`, `TMX_MOD_IDLE`, or `TMX_MOD_READY`.
- `user[]`: space is set aside for the user scratch area. The size of the space is controlled by the constant `TMX_USR_SIZE` and cannot be less than one `UINT4` element. This element can be used by the user for any type of storage, but only when the MDB field `modValid` is set.

Table 15: Module Data Block: sTMX_MDB

Field Name	Field Type	Field Description
errModule	INT4	Global error indicator for module calls
maxDevs	UINT2	Maximum number of devices supported
autoStart	BOOLEAN	Indication that `temuxModuleStart` will be called internally
ModState	UINT2	Module state (`TMX_MOD_START`, `TMX_MOD_IDLE`, `TMX_MOD_READY`)
ModValid	UINT2	Indication that this structure has been initialized (contains `0xCAFE`)
NumDevs	UINT2	Number of devices currently registered
numProfiles	UINT2	Number of profiles currently registered
semModule	void *	Semaphore object

Field Name	Field Type	Field Description
bufOK	BOOLEAN	Indicates that the call sysTemuxBufferStart succeeded
isrOK	BOOLEAN	Indicates that the ISR is installed
appMDB	BOOLEAN	Indication that the MDB was pre-allocated by the application
user[]	UINT4	Extra space for use by the application. The array is sized by the constant value: TMX_USR_SIZE
timerModule	void *	Timer object
modMSB	sTMX_MSB	Module Status Block
pDDB[]	sTMX_DDB	Array of (pointers to) DDBs – maxDevs determines how many in the array

Module Status Block

The MSB contains dynamic information about the health of the module.

Table 16: Module Status Block: sTMX_MSB

Field Name	Field Type	Field Description
statModule	INT4	General health of the module
valid	BOOLEAN	Indication that this structure is valid

Device Data Block

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. Special or unusual elements in the DDB are described first, followed by the complete list of elements in table form.

- errDevice: most of the device functions return a specific error code directly. When the returned code is TMX_FAIL, that 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 code (this eliminates the need to include the DDB template into the application code).
- usrCtxt: this value can be used by the user to identify this device during the execution of callback functions. It is passed to the driver in the DIV when temuxAdd is called and returned to the user in the DPV when a callback function is called. The element is unused by the driver itself and may contain any value.

- DevState: contains the current state of the device and could be set to: TMX_START, TMX_PRESENT, TMX_ACTIVE, or TMX_INACTIVE.
- devValid: indicates that this structure has been properly initialized and may be read by the user.
- user[]: space is set aside for the user scratch area. The size of the space is controlled by the constant TMX_USR_SIZE and cannot be less than one UINT4 element. This element can be used by the user for any type of storage while the MDB is 'valid', even if the DDB is not 'valid'.

Table 17: Device Data Block: sTMX_DDB

Field Name	Field Type	Field Description
errDevice	INT4	Global error indicator for device calls
baseAddress	UINT1 *	Device base address
usrCtxt	void *	Information provided by the user and returned in callback functions
autoInit	BOOLEAN	Indication that the driver will invoke temuxInit internally
profileNum	UINT2	Current profile (mode) that is in use
modeISR	TMX_ISR_MODE	Indication of the ISR mode
cbackIO	void *	Address of the callback function for IO Events
cbackDS3	void *	Address of the callback function for DS3 Events
cbackFramer	void *	Address of the callback function for Framer Events
cbackMapper	void *	Address of the callback function for Mapper Events. (not valid in TECT3)
DevState	UINT2	Device state (TMX_START, TMX_PRESENT, TMX_ACTIVE, TMX_INACTIVE)
devValid	UINT2	Indication that this structure has been initialized (contains 0xBEEF)
isTemap	BOOLEAN	Indicates if device is TEMAP
isTect3	BOOLEAN	Indicates if device is TECT3
index	UINT2	Index of this DDB in the table of DDBs
revision	UINT2	Device type and version (from device registers)

Field Name	Field Type	Field Description
hwFail	BOOLEAN	Indicates if device cannot be found
maskSaved	BOOLEAN	Indicates if the current isr mask has been saved into the member savedMask
user[]	UINT4	Extra space for use by the application. The array is sized by the constant value: TMX_USR_SIZE
devIPV	sTMX_IPV	Initialization profile
devMASK	sTMX_MASK	Interrupt mask
savedMask	sTMX_MASK	Saved copy of the interrupt mask
devDSB	sTMX_DSB	Device Status Block

Device Status Block

DSB Top-level Structure

Table 18: Device Status Block: sTMX_DSB

Field Name	Field Type	Field Description
statDevice	INT4	A flag derived from periodic checks that verify that the device is OK
sbi_monitor	UINT1	SBI clock monitor register
clock_1_monitor	UINT1	Master clock monitor #1
clock_2_monitor	UINT1	Master clock monitor #2
clock_3_monitor	UINT1	Master clock monitor #3
clock_4_monitor	UINT1	Master clock monitor #4
clock_5_monitor	UINT1	Master clock monitor #5
io	struct tmx_dsb_io	Alarms, status and counters from the IO section(s)
ds3	struct tmx_dsb_ds3	Alarms, status and counters from the DS3 section(s)
mux[7]	struct tmx_dsb_mux	Alarms, status and counters from the MUX/MX12/DS2 section(s)

Field Name	Field Type	Field Description
framer[28]	struct tmx_dsb_framer	Alarms, status and counters from the FRAMER section(s)
mapper	struct tmx_dsb_mapper	Alarms, status and counters from the MAPPER section(s) (not valid in TECT3)
valid	BOOLEAN	Indication that this structure is valid

DSB Sub-structures

These structures also appear in the DSB (above).

Table 19: IO Section Status Block: struct tmx_dsb_io

Field Name	Field Type	Field Description
sbidet0	UINT2	SBI bus collision detect count
sbidet1	UINT2	SBI bus collision detect count

Table 20: DS3 Section Status Block: struct tmx_dsb_ds3

Field Name	Field Type	Field Description
frmr.stat	UINT1	DS3 framer status (and alarms)
pmon.stat	UINT1	DS3 framer PMON status
pmon.lcv	UINT2	Line code violation event count
pmon.ferr	UINT2	F-bit / M-bit error event count
pmon.exzs	UINT2	Excess zeros error event count
pmon.perr	UINT2	Parity error event count
pmom.cper	UINT2	Path parity error event count
pmon.febe	UINT2	FEBE event count
rdlc.stat	UINT1	DS3 framer PMON status

Field Name	Field Type	Field Description
prgd.pdr	UINT4	Pattern detector error count

Table 21: MUX Section Status Block: struct tmx_dsb_mux

Field Name	Field Type	Field Description
frmr.stat	UINT1	DS2 framer status
frmr.ferr	UINT1	DS2 framing bit-error event count

Table 22: Framer Section Status Block: struct tmx_dsb_framer

Field Name	Field Type	Field Description
pmon.fer	UINT1	FAS / Fe-bit / framing bit error event count
pmon.oof	UINT2	OOF / COFA / far end block error event count
pmon.bee	UINT2	CRC / bit error count
prbs.ecnt	UINT4	PRBS error count
e1.stat	UINT1	E1 framer status
e1.alarm	UINT1	E1 framer alarm bits
e1.crc	UINT2	E1 framer crc error count

Table 23: Mapper Section Status Block: struct tmx_dsb_mapper (Not valid in TECT3)

Field Name	Field Type	Field Description
telecom	UINT1	SONET/SDH Telecom Bus Signal Monitor Accumulation Trigger register contents

4.5 Global Variables

Although most of the variables and structure elements within the driver are supposed to be hidden from the application code, there are several that provide information that might come in handy for debugging. They are to be considered read-only by the application.

- `temuxMDB`: A global pointer to the Module Data Block (MDB). The address of the MDB is also returned to the application via the MIV (passed with the `temuxModuleOpen` function call), in order make available the error code field. The user is cautioned that the MDB is only valid if the '`modValid`' flag is set.
  - `errModule`: this structure element is used to store an error code that specifies the reason for a module API function's failure. The field is only valid when the function in question returns a `TMX_FAIL` value.
  - `modValid`: a flag that indicates when the MDB has been properly initialized and can be read by the user.
  - `modState`: this element stores the state of the module (see Figure 3: State Diagram).
- `temuxDDB[]`: An array of pointers to the individual Device Data Blocks. The address of each DDB is also returned to the application via the DIV (passed with the `temuxAdd` function call), in order to make available the error code field. The user is cautioned that a DDB is only valid if the '`devValid`' flag is set and 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 a device API function's failure. The field is only valid when the function in question returns a `TMX_FAIL` value.
  - `devValid`: a flag that indicates when the DDB has been properly initialized and can be read by the user.
  - `devState`: this element stores the state of the device (see Figure 3: State Diagram).

4.6 Structures Passed through RTOS Buffers

Interrupt Status Vector

This block captures the state of the device (during a POLL or during 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 `sysTemuxBufferStart` function call. An individual ISV buffer is then obtained by the driver via the `sysTemuxISVBufferGet` macro and returned to the 'pool' via the `sysTemuxISVBufferRtn` macro.

Table 24: ISR Status Vector: sTMX_ISV

Field Name	Field Type	Field Description
`devId`	`sTMX_HNDL`	Device handle

Field Name	Field Type	Field Description
usrCtxt	void *	The user context
master	UINT1	Copy of the master ISR register
intsSDH	UINT1	SDH master interrupt source (not valid in TECT3)
intsSBI	UINT1	SBI master interrupt source
intsDS3	UINT1	DS3 master interrupt source
intsDS2	UINT1	DS2 master interrupt source
intsMX12	UINT1	MX12 master interrupt source
channels	UINT4	Bitmask indicating which T1/E1 framer (of the 28) slice has generated an interrupt
sbiDet0	UINT2	Master SBIDET0 collision detect interrupt source
sbiDet1	UINT2	Master SBIDET1 collision detect interrupt source
io	struct tmx_isv_io	IO Section interrupts
ds3	struct tmx_isv_ds3	DS3 Section interrupts
mux[7]	struct tmx_isv_mux	MUX/MX12/DS2 Section interrupts
framer[28]	struct tmx_isv_framer	FRAMER Section interrupts
mapper	struct tmx_isv_mapper	MAPPER Section interrupts (not valid in TECT3)

ISV Sub-structures

These structures also appear in the ISV (above).

Table 25: IO Section ISR Status Vector: struct tmx_isv_io

Field Name	Field Type	Field Description
exsbi.und	UINT1	SBI bus egress underrun interrupt bits

Field Name	Field Type	Field Description
exsbi.ovr	UINT1	SBI bus egress overrun interrupt bits
exsbi.ints	UINT1	SBI bus parity error interrupt bits
insbi.und	UINT1	SBI bus ingress underrun interrupt bits
insbi.ovr	UINT1	SBI bus ingress overrun interrupt bits

Table 26: DS3 Section ISR Status Vector: struct tmx_isv_ds3

Field Name	Field Type	Field Description
pmon.ints	UINT1	DS3 PMON interrupt bits
rdlc.ints	UINT1	DS3 RDLC interrupt bits
rboc.ints	UINT1	DS3 RBOC interrupt bits
frmr.ints	UINT1	DS3 FRMR interrupt bits
tdpr.ints	UINT1	DS3 TDPR interrupt bits
mx23.ints	UINT1	DS3 MX23 interrupt bits
prgd.intr	UINT1	DS3 PRGD interrupt bits

Table 27: MUX Section ISR Status Vector: struct tmx_isv_mux

Field Name	Field Type	Field Description
ds2.ints	UINT1	DS2 per-channel interrupt bits
ds2.intr	UINT1	DS2 per-channel interrupt bits (additional)
mx12.ints	UINT1	MX12 per-channel interrupt bits

Table 28: Framer Section ISR Status Vector: struct tmx_isv_framer

Field Name	Field Type	Field Description
frInts1	UINT1	E1/T1 Interrupt Source #1 interrupt bits

Field Name	Field Type	Field Description
frInts2	UINT1	E1/T1 Interrupt Source #2 interrupt bits
rjat.ints	UINT1	E1/T1 receive jitter attenuator interrupt bits
tjat.ints	UINT1	E1/T1 transmit jitter attenuator interrupt bits
tx_elst.intr	UINT1	E1/T1 transmit elastic store interrupt bits (not valid in TEMAP)
sig_elst.intr	UINT1	E1/T1 receive signaling elastic store interrupt bits (not valid in TEMAP)
rx_elst.intr	UINT1	E1/T1 receive elastic store interrupt bits (not valid in TEMAP)
sigx.coss	UINT1	E1/T1 Signaling Extractor interrupt bits (not valid in TEMAP)
t1_frmr.ints	UINT1	T1 framer interrupt bits
e1_frmr.sints	UINT1	E1 framer status interrupt bits
e1_frmr.mints	UINT1	E1 framer maintenance interrupt bits
e1_frmr.nints	UINT1	E1 framer National Codeword interrupt bits
e1_frmr.lints	UINT1	E1 framer V52 Link interrupt bits
e1_tran.ints	UINT1	E1 transmit interrupts (not valid in TEMAP)
pmon.intr	UINT1	T1 performance monitor interrupt bits
aprm.ints	UINT1	T1 APRM interrupt bits (not valid in TEMAP)
rdlc.stat	UINT1	E1/T1 receive HDLC interrupt bits (not valid in TEMAP)
prbs.intr	UINT1	E1/T1 pattern generator interrupt bits
tdpr.ints	UINT1	E1/T1 transmit HDLC interrupt bits (not valid in TEMAP)
almi.ints	UINT1	E1/T1 alarm integrator interrupt bits
rboc.ints	UINT1	T1 receive bit oriented code interrupts (not valid in TEMAP)

Table 29: Mapper Section ISR Status Vector: struct tmx_isv_mapper

Field Name	Field Type	Field Description
etpp[TUG2].ais	UINT1	Egress Tributary Payload Processor AIS interrupt bits
etpp[TUG2].lop	UINT1	Egress Tributary Payload Processor LOP interrupt bits
etpp[TUG2].stat[TU][1]	UINT1	Egress Tributary Payload Processor alarm status interrupt bits
itpp[TUG2].ais	UINT1	Ingress Tributary Payload Processor AIS interrupt bits
itpp[TUG2].lop	UINT1	Ingress Tributary Payload Processor LOP interrupt bits
itpp[TUG2].stat[TU][1]	UINT1	Ingress Tributary Payload Processor alarm status interrupt bits
rtop[TUG2].copsl	UINT1	Receive Tributary Path Overhead Processor Change of Path Signal Label interrupt bits
rtop[TUG2].pslm	UINT1	Receive Tributary Path Overhead Processor Path Signal Label Mismatch interrupt bits
rtop[TUG2].pslu	UINT1	Receive Tributary Path Overhead Processor Path Signal Label Unstable interrupt bits
rtop[TUG2].rdi	UINT1	Receive Tributary Path Overhead Processor Remote Defect Indication interrupt bits
rtop[TUG2].rfi	UINT1	Receive Tributary Path Overhead Processor Remote Failure Indication interrupt bits
d3md.ints	UINT1	DS3 Drop Side Mapper interrupt bits
d3ma.ints	UINT1	DS3 Add Side Mapper interrupt bits

Note

1. TUG2 refers to a range from 1 to 7 corresponding to TUG2 #1 to TUG2 #7 and TU refers to a range from 1 to 4 corresponding TU #1 to TU #4

Deferred-Processing Routine Vector

The DPV block is the template for data that is assembled by the DPR and sent to the application code as a parameter in a callback routine. The callback routine itself identifies the Section of the device that caused the DPR processing. Arguments passed via the callback routine identify the source device (via the `usrContext`) and the event that triggered the processing. For some events, that is all the information that the user needs. For others, additional information is needed. The size of the DPV is kept under sixteen (16) bytes to accommodate the simpler message passing schemes used by some Operating Systems. The DPV structure definition shown below defines the format for sTMX_DPV_IO, sTMX_DPV_DS3, sTMX_DPV_FRAMER, and sTMX_DPV_MAPPER.

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

Table 30: Deferred-Processing Vector: sTMX_DPV

Field Name	Field Type	Field Description
`channels`	`UINT4`	Framer# or mx# of channel that triggered the event (if needed)
`data`	`UINT4`	Pointer to HDLC receive data or transmit buffers

5 APPLICATION PROGRAMMING INTERFACE

This section provides a detailed description of each function that is a member of the TEMUX/TEMAP/TECT3 driver Application Programming Interface (API).

5.1 Module Initialization

Opening Modules: temuxModuleOpen

This function performs module level initialization of the driver. This involves allocating all of the memory needed by the driver and initializing the internal structures. It is also possible for the user to pre-allocate the memory needed by the driver. The user can also set a flag in the MIV that will cause this function to invoke `temuxModuleStart` before returning.

Prototype `INT4 temuxModuleOpen (sTMX_MIV *pMIV)`

Inputs `pMIV` : (pointer to) the MIV

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

Returns Success = `TMX_OK`
Failure = `TMX_ERR_ARG`
`TMX_ERR_ISOPEN`
`TMX_ERR_ALLOC`

Valid States `MOD_START`

Side Effects changes the STATE of the MODULE to IDLE

Closing Modules: temuxModuleClose

This function performs module level shutdown of the driver. If the driver is in the READY state, then `temuxModuleStop` will be called. All RTOS resources will be returned to the RTOS and the MDB de-allocated.

Prototype `INT4 temuxModuleClose (void)`

Inputs None

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_STOP`
`TMX_ERR_NOTIDLE`

Valid States ALL STATES

Side Effects Changes the STATE of the MODULE to START

5.2 Module Activation

Starting Modules: temuxModuleStart

This function connects the RTOS resources to the driver. This involves allocating semaphores, initializing buffers and installing the ISRs and the Deferred-Processing Routine (DPR) Task. Upon successful return from this function, the driver is ready to add active devices.

Prototype `INT4 temuxModuleStart (void)`

Inputs None

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_ISREADY`
`TMX_ERR_NOTIDLE`

Valid States IDLE

Side Effects changes the STATE of the MODULE to READY

Stopping Modules: temuxModuleStop

This function disconnects the RTOS resources from the driver. This involves deallocating semaphores, freeing-up buffers and uninstalling the ISRs and the Deferred-Processing Routine (DPR) Task. If there are any registered devices, `temuxDelete` is called for each.

Prototype `INT4 temuxModuleStop (void)`

Inputs None

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_ISIDLE`
`TMX_ERR_NOTREADY`

Valid States READY

Side Effects Changes the STATE of the MODULE to IDLE

5.3 Device Initialization

Adding Devices: temuxAdd

This function verifies the presence of a new device in the hardware, configures a Device Data Block (DDB), stores the contents of the passed Device Initialization Vector (DIV) and passes a handle back to the application. The handle is used as a parameter to most of the device API functions. Caution: It is the user's responsibility to ensure enough space has been allocated for the MDB and DDBs if the user decides to handle this task (indicated by passing NULL for the `pDIV` parameter).

Prototype `sTMX_HNDL temuxAdd (sTMX_DIV *pDIV)`

Inputs `pDIV` : (pointer to) the DIV

Outputs Places the address of the DDB into the DIV passed by the application. Places any error codes into the MDB.

Returns Success = Handle that must be used with most other device calls
Failure = NULL with `temuxMdb->errModule` set to either
`TMX_ERR_ARG,`
`TMX_ERR_CLOSED,`
`TMX_ERR_INVALID,`
`TMX_ERR_NOTREADY,`
`TMX_ERR_MAXDEVICE,`
`TMX_ERR_ADD,`
`TMX_ERR_HNDL,`
`TMX_ERR_HWFAIL`

Valid States START

Side Effects changes the STATE of the DEVICE to PRESENT

Deleting Devices: temuxDelete

This function removes the specified device from the group of devices being controlled by the TEMUX/TEMAP/TECT3 driver. Deleting a device involves invalidating the DDB for that device.

Prototype `INT4 temuxDelete (sTMX_HNDL devId)`

Inputs `devId` : device handle (from temuxAdd)

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_NOTREADY`
`TMX_ERR_DEACTIVATE`
`TMX_ERR_NOTINACTIVE`
`TMX_ERR_RESET`
`TMX_ERR_NOTPRESENT`
`TMX_ERR_ISSTART`

Valid States ACTIVE, INACTIVE, PRESENT

Side Effects Changes the STATE of the device to START

Initializing Devices: temuxInit

This function initializes the device from the information stored in both the DDB and in profiles that are hard coded into the driver. The device is reset before the initialization is carried out.

Prototype `INT4 temuxInit (sTMX_HNDL devId, sTMX_DIV *pDIV, UINT2 profileNum)`

Inputs `devId` : device handle (from `temuxAdd`)

`pDIV` : pointer to the Device Initialization Vector

`profileNum` : identifies the MODE profile to use

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_HNDL`
`TMX_ERR_ARG`
`TMX_ERR_NOTREADY`
`TMX_ERR_INVALID`
`TMX_ERR_NOTPRESENT`
`TMX_ERR_CLOSED`

Valid States PRESENT

Side Effects Changes the STATE of the DEVICE to INACTIVE

Resetting Devices: temuxReset

This function applies a software reset to the TEMUX/TEMAP/TECT3 device and in doing so reinitializes the DDB for this device. This function is typically called before reinitializing the device (via `temuxInit`).

Prototype `INT4 temuxReset (sTMX_HNDL devId)`

Inputs `devId` : device handle (from `temuxAdd`)

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_HNDL`
`TMX_ERR_CLOSED`
`TMX_ERR_NOTREADY`
`TMX_ERR_DEACTIVATE`
`TMX_ERR_INVALID`
`TMX_ERR_NOTINACTIVE`

Valid States ACTIVE, INACTIVE

Side Effects Changes the STATE of the DEVICE to PRESENT

Deactivating Devices: temuxDeActivate

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

Prototype `INT4 temuxDeActivate (sTMX_HNDL devId)`

Inputs `devId` : device handle (from `temuxAdd`)

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_HNDL`
`TMX_ERR_INVALID`
`TMX_ERR_NOTREADY`
`TMX_ERR_CLOSED`
`TMX_ERR_NOTACTIVE`

Valid States ACTIVE

Side Effects Changes the STATE of the DEVICE to INACTIVE

Activating Devices: temuxActivate

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

Prototype
```
INT4 temuxActivate (sTMX_HNDL devId)
```

Inputs `devId` : device handle (from `temuxAdd`)

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_HNDL`
`TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_NOTREADY`
`TMX_ERR_NOTINACTIVE`

Valid States INACTIVE

Side Effects Changes the STATE of the DEVICE to ACTIVE

Add Initialization Profile: temuxAddInitProfile

This function is used to add a profile to a vector of profiles. Note that the first 8 profiles (0-7) are preset and can not be altered or deleted. Profiles between 8 and `TMX_MAX_IPROFILES` can be added.

Prototype
```
INT4 temuxAddInitProfile (sTMX_INIT_PROF
*pProf, UINT2 *pprofileNum)
```

Inputs `pProf` : pointer to a vector of profiles
`pprofileNum` : pointer to the profile number, assigned by `temuxAddInitProfile`

Outputs profile number assigned by the driver

Returns Success = `TMX_OK`
Failure = `TMX_ERR_ARG`
`TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_MAXPROF`

Valid States ALL STATES except `MOD_START`

Side Effects None

Get Initialization Profile: temuxGetInitProfile

This function is used to get a profile from a vector of profiles. Note that the first 8 profiles (0-7) are preset and profiles between 8 and `TMX_MAX_IPROFILES` are user defined.

Prototype
```
INT4 temuxGetInitProfile (UINT2 profileNum,
sTMX_INIT_PROF *pProf)
```

Inputs `profileNum` : the profile to return
`pProf` : pointer to a vector of profiles

Outputs contents of the referenced Profile

Returns Success = `TMX_OK`
Failure = `TMX_ERR_ARG`
`TMX_ERR_CLOSED`
`TMX_ERR_INVALID`

Valid States ALL STATES except `MOD_START`

Side Effects None

Delete Initialization Profile: temuxDeleteInitProfile

This function is used to delete an added profile from a vector of profiles. Note that the first 8 profiles are preset and can not be deleted and profiles between 8 and `TMX_MAX_IPROFILES` are user defined and may be removed.

Prototype
```
INT4 temuxDeleteInitProfile (UINT2 profileNum)
```

Inputs `profileNum` : the profile to delete

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_ARG`
`TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_RANGE`

Valid States ALL STATES except `MOD_START`

Side Effects None

Updating a device: temuxUpdate

This function is used to update a device that has been already been added and initialized with temuxAdd. A new device initialization vector and profile number can be passed in to configure the device with a different configuration.

Prototype
```
INT4 temuxUpdate (sTMX_HNDL devId, sTMX_DIV
*pDIV, UINT2 profileNum)
```

Inputs
`devId` : handle from `temuxAdd()`
`pDIV` : (pointer) to Device Initialization Vector
`profileNum` : profile number to use when initializing device

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_HNDL`
`TMX_ERR_ARG`
`TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_NOTREADY`
`TMX_ERR_ISPRESENT`

Valid States ACTIVE, INACTIVE

Side Effects None

5.4 Device Reading and Writing

Reading Registers: temuxRead

This function reads the registers of specific TEMUX/TEMAP/TECT3 devices by providing the register number.

Prototype `UINT1 temuxRead (sTMX_HNDL devId, UINT2 regNum)`

Inputs
`devId` : device handle (from `temuxAdd`)
`regNum` : register number

Outputs
Error Code written to MDB:
`TMX_ERR_HNDL`
Error Code written to DDB:
Success = `TMX_OK`
Failure = `TMX_ERR_RANGE`
`TMX_ERR_ADDR`
`TMX_ERR_HWFAIL`

Returns Data read from the register

Valid States PRESENT, ACTIVE, INACTIVE

Side Effects MAY affect registers that change after a read operation

Writing Registers: temuxWrite

This function writes to the registers of specific TEMUX/TEMAP/TECT3 devices by providing the register number.

Prototype
```
UINT1 temuxWrite (sTMX_HNDL devId, UINT2
regNum, UINT1 wdata)
```

Inputs
`devId` : device handle (from `temuxAdd`)
`regNum` : register number
`wdata` : data to be written

Outputs
Error Code written to MDB:
`TMX_ERR_HNDL`
Error Code written to DDB:
Success = `TMX_OK`
Failure = `TMX_ERR_RANGE`
`TMX_ERR_ADDR`
`TMX_ERR_HWFAIL`

Returns Previous register value

Valid States PRESENT, ACTIVE, INACTIVE

Side Effects May change the configuration of the device

Reading Framer Registers: temuxReadFR

This function reads the E1/T1 framer registers of specific TEMUX/TEMAP/TECT3 devices by providing the framer number and register number.

Prototype
```
UINT1 temuxReadFR (sTMX_HNDL devId, UINT2
frNum, UINT2 regNum)
```

Inputs

`devId`	: device handle (from `temuxAdd`)
`frNum`	: framer number (1-28 T1, 1-21 E1)
`regNum`	: register number

Outputs Error Code written to MDB:
`TMX_ERR_HNDL`
Error Code written to DDB:
Success = `TMX_OK`
Failure = `TMX_ERR_RANGE`
`TMX_ERR_ADDR`
`TMX_ERR_HWFAIL`

Returns data read from framer register

Valid States PRESENT, ACTIVE, INACTIVE

Side Effects MAY affect registers that change after a read operation

Writing Framer Registers: temuxWriteFR

This function writes to the framer registers of specific TEMUX/TEMAP/TECT3 devices by providing the framer number and register number. If the framer number passed is zero, all the framers will be written to with the same value.

Note: A failure to write forces a return of zero and any specific error indication is written to the associated DDB

Prototype
```
UINT1 temuxWriteFR (sTMX_HNDL devId, UINT2
frNum, UINT2 regNum, UINT1 wdata)
```

Inputs

`devId`	: device handle (from `temuxAdd`)
`frNum`	: framer number (1-28 T1, 1-21 E1)
`regNum`	: register number
`wdata`	: data to be written

Outputs

Error Code written to MDB:
`TMX_ERR_HNDL`
Error Code written to DDB:
Success = `TMX_OK`
Failure = `TMX_ERR_RANGE`
`TMX_ERR_ADDR`
`TMX_ERR_HWFAIL`

Returns previous register value

Valid States PRESENT, ACTIVE, INACTIVE

Side Effects May change the configuration of the device

Reading DS2/MX12 Multiplexer Registers: temuxReadMX

This function reads the DS2/MX12 multiplexer registers of specific TEMUX/TEMAP/TECT3 devices by providing the slice number and register number.

Prototype

```
UINT1 temuxReadMX (sTMX_HNDL devId, UINT2
mxNum, UINT2 regNum)
```

Inputs

`devId`	: device handle (from `temuxAdd`)
`mxNum`	: framer number (1-7)
`regNum`	: register number

Outputs

Error Code written to MDB:
`TMX_ERR_HNDL`
Error Code written to DDB:
Success = `TMX_OK`
Failure = `TMX_ERR_RANGE`
`TMX_ERR_ADDR`
`TMX_ERR_HWFAIL`

Returns data read from framer register

Valid States PRESENT, ACTIVE, INACTIVE

Side Effects MAY affect registers that change after a read operation

Writing DS2/MX12 Multiplexer Registers: temuxWriteMX

This function writes to a DS2/MX12 multiplexer register of a specific TEMUX/TEMAP/TECT3 device by providing the framer number and register number. If the multiplexer number passed is zero, all multiplexers will be written with the same value.

Note: A failure to write forces a return of zero and any specific error indication is written to the associated DDB

Prototype
```
UINT1 temuxWriteMX (sTMX_HNDL devId, UINT2
mxNum, UINT2 regNum, UINT1 wdata)
```

Inputs

`devId`	: device handle (from `temuxAdd`)
`mxNum`	: framer number (1-7)
`regNum`	: register number
`wdata`	: data to be written

Outputs

Error Code written to MDB:
`TMX_ERR_HNDL`
Error Code written to DDB:
Success = `TMX_OK`
Failure = `TMX_ERR_RANGE`
`TMX_ERR_ADDR`
`TMX_ERR_HWFAIL`

Returns Previous register value

Valid States PRESENT, ACTIVE, INACTIVE

Side Effects May change the configuration of the device

Reading Indirect Registers: temuxReadInd

This function reads an indirect register of a specific TEMUX/TEMAP/TECT3 device by providing the section number and other arguments.

Prototype
```
UINT1 temuxReadInd (sTMX_HNDL devId,
TMX_SECTION section, UINT2 arg1, UINT2 arg2,
UINT2 arg3, UINT2 arg4)
```

Inputs

`devId`	: device handle (from `temuxAdd`)
`section`	: section (`RPSC`,`TPSC`,`SIGX`,`INSBI` `EXSBI`,`RTDM`,`TRAP`,`TTOP`,`TTMP`)
`arg1`	: see parameter table below
`arg2`	: see parameter table below
`arg3`	: see parameter table below
`arg4`	: see parameter table below

Outputs Error Code written to MDB:
`TMX_ERR_HNDL`
Error Code written to DDB:
Success = `TMX_OK`
Failure = `TMX_ERR_RANGE`
`TMX_ERR_ADDR`
`TMX_ERR_HWFAIL`

Returns Success = data read from indirect register

Valid States PRESENT, ACTIVE, INACTIVE

Side Effects MAY affect registers that change after a read operation

Table 31: Table of Parameters: temuxReadInd

section	arg1	arg2	arg3	arg4
RPSC	Framer Number	Indirect Register	n/u	n/u
TPSC	Framer Number	Indirect Register	n/u	n/u
SIGX	Framer Number	Indirect Register	n/u	n/u
INSBI	SPE Number	Tributary Number	n/u	n/u
EXSBI	SPE Number	Tributary Number	n/u	n/u
RTDM	Page Number	SPE Number	Stream Number	n/u
TRAP	Page Number	TUG3 Number	TUG2 Number	TU Number
TTOP	TUG3 Number	TUG2 Number	TU Number	Register Number
TTMP	Page Number	TUG3 Number	TUG2 Number	TU Number

`RPSC, TPSC, SIGX` are not valid for the TEMAP. `RTDM, TRAP, TTOP, TTMP` are not valid in the TECT3.

Writing Indirect Registers: temuxWriteInd

This function writes to an indirect register of a specific TEMUX/TEMAP/TECT3 device by providing the section number and other arguments. This function derives the actual address location based on the device handle, section number, and other argument inputs. It then writes the contents of the data parameter to this address location using the system specific macro, `sysTemuxWriteReg`.

Prototype

```
UINT1 temuxWriteInd (sTMX_HNDL devId,
TMX_SECTION section, UINT2 arg1, UINT2 arg2,
UINT2 arg3, UINT2 arg4, UINT1 wdata)
```

Inputs

`devId`	: device handle (from `temuxAdd`)
`section`	: section (`RPSC,TPSC,SIGX,INSBI EXSBI,RTDM,TRAP,TTOP,TTMP`)
`arg1`	: see parameter table below
`arg2`	: see parameter table below
`arg3`	: see parameter table below
`arg4`	: see parameter table below
`wdata`	: data to be written

Outputs

Error Code written to MDB:
`TMX_ERR_HNDL`
Error Code written to DDB:
Success = `TMX_OK`
Failure = `TMX_ERR_RANGE`
`TMX_ERR_ADDR`
`TMX_ERR_HWFAIL`

Returns Success = last previous value found

Valid States PRESENT, ACTIVE, INACTIVE

Side Effects May change the configuration of the device

Table 32: Table of Parameters: temuxWriteInd

section	arg1	arg2	arg3	arg4
RPSC	Framer Number	Indirect Register	n/u	n/u
TPSC	Framer Number	Indirect Register	n/u	n/u
SIGX	Framer Number	Indirect Register	n/u	n/u
INSBI	SPE Number	Tributary Number	n/u	n/u
EXSBI	SPE Number	Tributary Number	n/u	n/u
RTDM	Page Number	SPE Number	Stream Number	n/u
TRAP	Page Number	TUG3 Number	TUG2 Number	TU Number
TTOP	TUG3 Number	TUG2 Number	TU Number	Register Number
TTMP	Page Number	TUG3 Number	TUG2 Number	TU Number

RPSC, TPSC, SIGX are not valid for the TEMAP. RTDM, TRAP, TTOP, TTMP are not valid in the TECT3.

Reading from Register Blocks: temuxReadBlock

This function reads the block registers of a specific TEMUX/TEMAP/TECT3 device by providing the starting register number and the length.

Prototype
```
UINT1 temuxReadBlock (sTMX_HNDL devId, UINT2
regNum, UINT2 length, UINT1 *pBlock)
```

Inputs

devId	: device handle (from temuxAdd)
regNum	: register number
length	: number of registers to read
pBlock	: (pointer to) block read area

Outputs

Error Code written to MDB:
Success = TMX_OK
Failure = TMX_ERR_HNDL
Error Code written to DDB:
Success = TMX_OK
Failure = TMX_ERR_RANGE
TMX_ERR_ADDR
TMX_ERR_HWFAIL

Returns

Last register value read

Valid States PRESENT, ACTIVE, INACTIVE

Side Effects MAY affect registers that change after a read operation

Writing to Register Blocks: temuxWriteBlock

This function writes to the block registers of a specific TEMUX/TEMAP/TECT3 device by providing the starting register number and length.

Note: A failure to write forces a return of zero and any specific error indication is written to the associated DDB

Prototype
```
UINT1 temuxWriteBlock (sTMX_HNDL devId, UINT2
regNum, UINT2 length, UINT1 *pBlock)
```

Inputs

`devId`	: device handle (from `temuxAdd`)
`regNum`	: register number
`length`	: number of register to write
`pBlock`	: (pointer to) the block of data to write

Outputs
Error Code written to MDB:
`TMX_ERR_HNDL`
Error Code written to DDB:
Success = `TMX_OK`
Failure = `TMX_ERR_RANGE`
`TMX_ERR_ADDR`
`TMX_ERR_HWFAIL`

Returns last previous value found

Valid States PRESENT, ACTIVE, INACTIVE

Side Effects May change the configuration of the device

Reading Mapper Registers: temuxReadMapper (TEMUX/TEMAP only)

This function reads the mapper registers of a specific TEMUX/TEMAP device by providing the base register, TUG3, TUG2 and TU number.

Prototype
```
UINT1 temuxReadMapper (sTMX_HNDL devId, UINT2
regNum, UINT2 tug3, UINT2 tug2, UINT2 tu)
```

Inputs

`devId`	: device handle (from `temuxAdd`)
`regNum`	: register number
`tug3`	: tributary unit 3 group number (1-3)
`tug2`	: tributary unit 2 group number (1-7)
`tu`	: tributary unit number (1-4)

Outputs Error Code written to MDB:
`TMX_ERR_HNDL`
Error Code written to DDB:
Success = `TMX_OK`
Failure = `TMX_ERR_RANGE`
`TMX_ERR_ADDR`
`TMX_ERR_HWFAIL`

Returns data read from mapper register

Valid States PRESENT, ACTIVE, INACTIVE

Side Effects MAY affect registers that change after a read operation

Writing Mapper Registers: temuxWriteMapper (TEMUX/TEMAP only)

This function writes to the mapper register of a specific TEMUX/TEMAP device by providing the register, TUG3, TUG2 and TU number. This function derives the actual address location based on the device handle, register, TUG3, TUG2, and TU number inputs. It then writes the contents of this address location using the system specific macro, `sysTemuxWriteReg`.

Note: A failure to write forces a return of zero and any specific error indication is written to the associated DDB

Prototype
```
UINT1 temuxWriteMapper (sTMX_HNDL devId, UINT2
regNum, UINT2 tug3, UINT2 tug2, UINT2 tu, UINT1
wdata)
```

Inputs

`devId`	: device handle (from `temuxAdd`)
`regNum`	: register number
`tug3`	: tributary unit 3 group number (1-3)
`tug2`	: tributary unit 2 group number (1-7)
`tu`	: tributary unit number (1-4)
`wdata`	: data to be written

Outputs Error Code written to MDB:
`TMX_ERR_HNDL`
Error Code written to DDB:
Success = `TMX_OK`
Failure = `TMX_ERR_RANGE`
`TMX_ERR_ADDR`
`TMX_ERR_HWFAIL`

Returns Previous register value

Valid States PRESENT, ACTIVE, INACTIVE

Side Effects May change the configuration of the device

DS3-HDLC Service: temuxLinkDataDS3

This function is normally called after a callback by indicating that the DS3-TxHDLC register fifo is nearing empty or that the DS3-RxHDLC register fifo is nearing full.

Prototype `INT4 temuxLinkDataDS3 (sTMX_HNDL devId, UINT1 *data, UINT2 length, BOOLEAN read)`

Inputs

`devId`	: device handle (from `temuxAdd`)
`data`	: (pointer to) HDLC data
`length`	: size of HDLC data
`read`	: if set, read link data

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_HNDL`
`TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_NOTREADY`
`TMX_ERR_ISSTART`
`TMX_ERR_ISPRESENT`
`TMX_ERR_ADDR`

Valid States ACTIVE

Side Effects None

T1-HDLC Service: temuxLinkDataT1 (TEMUX/TECT3 Only)

This function is normally called after a callback when it indicates that the T1-TxHDLC register fifo is nearing empty or that the T1-RxHDLC register fifo is nearing full.

Prototype `INT4 temuxLinkDataT1 (sTMX_HNDL devId, UINT2 frNum, UINT1 *data, UINT2 length, BOOLEAN read)`

Inputs

`devId`	: device handle (from `temuxAdd`)
`frNum`	: framer number (1-28)
`data`	: (pointer to) HDLC data
`length`	: size of HDLC data
`read`	: if set, read link data

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_HNDL`
`TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_NOTREADY`
`TMX_ERR_ISSTART`
`TMX_ERR_ISPRESENT`
`TMX_ERR_ADDR`
`TMX_ERR_ARG`

Valid States ACTIVE

Side Effects None

5.5 Interrupt Service Functions

Getting Mask Registers: temuxGetMask

This function returns the contents of the interrupt mask registers of the TEMUX/TEMAP/TECT3 device.

Prototype `INT4 temuxGetMask(sTMX_HNDL devId, void *pMASK)`

Inputs `devId` : device handle (from `temuxAdd`)
`pMASK` : (pointer to) mask structure (`sTMX_MASK *`) cast to void *

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_HNDL`
`TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_NOTREADY`
`TMX_ERR_ISSTART`
`TMX_ERR_ARG`

Valid States PRESENT, ACTIVE, INACTIVE

Side Effects None

Setting Mask Registers: temuxSetMask

This function sets individual interrupt bits and registers in the TEMUX/TEMAP/TECT3 device. Any bits that are set in the passed structure are set in the associated TEMUX/TEMAP/TECT3 registers.

Prototype `INT4 temuxSetMask (sTMX_HNDL devId, void *pMASK)`

Inputs

`devId`	: device handle (from `temuxAdd`)
`pMASK`	: (pointer to) mask structure (`sTMX_MASK *`) cast to void *

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_HNDL`
`TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_NOTREADY`
`TMX_ERR_ISSTART`
`TMX_ERR_ARG`

Valid States PRESENT, ACTIVE, INACTIVE

Side Effects May change the operation of the ISR/DPR

Clearing Mask Registers: temuxClearMask

This function clears individual interrupt bits and registers in the TEMUX/TEMAP/TECT3 device. Any bits that are set in the passed structure are cleared in the associated TEMUX/TEMAP/TECT3 registers.

Prototype `INT4 temuxClearMask (sTMX_HNDL devId, void *pMASK)`

Inputs

`devId`	: device handle (from `temuxAdd`)
`pMASK`	: (pointer to) mask structure (`sTMX_MASK *`) cast to void *

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_HNDL`
`TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_NOTREADY`
`TMX_ERR_ISSTART`
`TMX_ERR_ARG`

Valid States PRESENT, ACTIVE, INACTIVE

Side Effects May change the operation of the ISR/DPR

Polling Interrupt Registers: temuxPoll

This function commands the driver to poll the interrupt registers in the device. The call will fail unless the device was initialized into polling mode. The output of the poll is the same as it would be if interrupts were enabled: the data gathered is passed to the DPR for disposition.

Prototype `INT4 temuxPoll (sTMX_HNDL devId, void *pBuf)`

Inputs

`devId`	: device handle (from `temuxAdd`)
`pBuf`	: (pointer to) an ISV ((`sTMX_ISV` *) cast to `void` *)

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_HNDL`
`TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_NOTREADY`
`TMX_ERR_ISSTART`
`TMX_ERR_MODE`

Valid States ACTIVE, INACTIVE

Side Effects None

Interrupt Service: temuxISR

This function reads the state of the interrupt registers in the TEMUX/TEMAP/TECT3 and stores them into an ISV. Performs whatever functions are needed to clear the interrupt, from simply clearing bits to complex functions. This routine is called by the application code, from within `sysTemuxISRHandler`.

Prototype `void *temuxISR (sTMX_HNDL devId)`

Inputs

`devId`	: device Handle (from `temuxAdd`)

Outputs None

Returns (pointer to) an ISV or NULL on error

Valid States ACTIVE, INACTIVE, PRESENT

Side Effects None

Interrupt Processing: temuxDPR

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

Prototype `void *temuxDPR (void *pBuf)`

Inputs `pBuf` : (pointer to) an ISV ((`sTMX_ISV` *) cast to `void` *)

Outputs None

Returns (pointer to) an ISV or NULL on error

Valid States ACTIVE, INACTIVE, PRESENT

Side Effects None

Configure ISR: temuxISRConfig

This function sets the mode of operations for the ISR/DPR functions.

Prototype
```
INT4 temuxISRConfig (sTMX_HNDL devId,
TMX_ISR_MODE mode)
```

Inputs
`devId` : device handle (from `temuxAdd`)
`mode` : ISR/polling mode:
`TMX_ISR_HDWR` - hardware ISR mode
`TMX_ISR_MANUAL` - polling mode

Outputs None

Returns
Success = `TMX_OK`
Failure = `TMX_ERR_HNDL`
`TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_NOTREADY`
`TMX_ERR_ISSTART`
`TMX_ERR_ARG`

Valid States PRESENT, INACTIVE, ACTIVE

Side Effects None

5.6 Alarms, Status and Statistics Functions

The TEMUX/TEMAP/TECT3 device provides alarm, simple status and statistical counts of errors via the following functions.

Retrieving Statistical Counts: temuxGetStats

This function retrieves the statistical counts that are kept by the TEMUX/TEMAP/TECT3 device.

Prototype `INT4 temuxGetStats (sTMX_HNDL devId, void *pBlock)`

Inputs

`devId`	: device handle (from `temuxAdd`)
`pBlock`	: (pointer to) device status block ((`sTMX_DSB` *) cast to `void` *)

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_HNDL`
`TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_NOTREADY`
`TMX_ERR_ISSTART`
`TMX_ERR_ADDR`

Valid States ACTIVE, INACTIVE

Side Effects None

Clearing Statistical Counts: temuxClearStats

This function clears the statistical counts that are kept by the TEMUX/TEMAP/TECT3 device.

Prototype `INT4 temuxClearStats (sTMX_HNDL devId)`

Inputs

`devId`	: device handle (from `temuxAdd`)

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_HNDL`
`TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_NOTREADY`
`TMX_ERR_ISSTART`
`TMX_ERR_ADDR`

Valid States ACTIVE, INACTIVE

Side Effects None

5.7 Device Diagnostics

Clearing/Setting Mapper Loopbacks: temuxLoopMapper (TEMUX/TEMAP only)

This function clears or sets loopbacks within the Mapper section of the device. It is up to the user to perform any tests on the looped data.

Prototype
```
INT4 temuxLoopMapper (sTMX_HNDL devId,
TMX_LOOP_TYPE loop, BOOLEAN enable)
```

Inputs

`devId`	: device handle (from `temuxAdd`)	
`enable`	: clears loop if clear, else sets loop	

`loop`	: loop type:	`TMX_NOLOOP`
		`TMX_DLOOP`
		`TMX_LLOOP`
		`TMX_PLOOP`

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_HNDL`
`TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_NOTREADY`
`TMX_ERR_ISPRESENT`
`TMX_ERR_ADDR`
`TMX_ERR_ARG`

Valid States ACTIVE, INACTIVE

Side Effects May inhibit the flow of active data

Clearing/Setting DS3 Devices Loopbacks: temuxLoopDS3

This function clears or sets loopbacks within the DS3 section of the device. It is up to the user to perform any tests on the looped data.

Prototype `INT4 temuxLoopDS3 (sTMX_HNDL devId, TMX_LOOP_TYPE loop, BOOLEAN enable)`

Inputs

`devId`	: device handle (from `temuxAdd`)	
`enable`	: clears loop if clear, else sets loop	

`loop`	: loop type:	`TMX_NOLOOP`
		`TMX_DLOOP`
		`TMX_LLOOP`
		`TMX_PLOOP`

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_HNDL`
`TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_NOTREADY`
`TMX_ERR_ISPRESENT`
`TMX_ERR_ADDR`
`TMX_ERR_ARG`

Valid States ACTIVE, INACTIVE

Side Effects May inhibit the flow of active data

Clearing/Setting DS3 Bert Tests: temuxBertDS3

This function clears or sets the Pseudo Random Pattern generation and detection hardware within the DS3 section of the device. The results can be returned to the application code via normal ISR processing or by calling `temuxGetStats`. It is up to the user to interpret any results from the test.

Prototype `INT4 temuxBertDS3 (sTMX_HNDL devId, sTMX_PRGD *pPRGD)`

Inputs

`devId`	: device handle (from `temuxAdd`)
`pPRGD`	: (pointer to) BERT test block

Outputs Error Code written to the MDB on failure

Returns Success = `TMX_OK`
Failure = `TMX_ERR_HNDL`
`TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_NOTREADY`
`TMX_ERR_ISPRESENT`
`TMX_ERR_ADDR`

Valid States ACTIVE, INACTIVE

Side Effects May inhibit the flow of active data

Clearing or Setting Bert Framer: temuxBertFramer (TEMUX/TECT3 only)

This function clears or sets the Pseudo Random Pattern generation and detection hardware within the E1/T1 section of the device. The results can be returned to the application code via normal ISR processing or by calling `temuxGetStats`. It is up to the user to interpret any results from the test.

Prototype
```
INT4 temuxBertFramer (sTMX_HNDL devId, UINT2
frNum, sTMX_PRBS *pPRBS)
```

Inputs

`devId`	: device handle (from `temuxAdd`)
`frNum`	: framer number (1-28 T1), (1-21 E1)
`pPRBS`	: (pointer to) PRBS structure

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_HNDL`
`TMX_ERR_RANGE`
`TMX_ERR_ARG`
`TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_NOTREADY`
`TMX_ERR_ISPRESENT`
`TMX_ERR_ISTEMAP`
`TMX_ERR_ADDR`

Valid States INACTIVE, ACTIVE

Side Effects Will inhibit the flow of active data

Clearing/Setting E1/T1 Framer Loopbacks: temuxLoopFramer (TEMUX/TECT3 only)

This function clears or sets loopbacks within the E1/T1 framer section of the device. It is up to the user to perform any tests on the looped data.

Prototype `INT4 temuxLoopFramer (sTMX_HNDL devId, UINT2 frNum, TMX_LOOP_TYPE loop, BOOLEAN enable)`

Inputs

`devId`	: device handle (from `temuxAdd`)	
`enable`	: clears loop if clear, else sets loop	
`frNum`	: framer number (1-28 T1), (1-21 E1)	

`loop`	: loop type:	`TMX_NOLOOP`
		`TMX_DLOOP`
		`TMX_LLOOP`
		`TMX_PLOOP`

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_HNDL`
`TMX_ERR_RANGE`
`TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_NOTREADY`
`TMX_ERR_ISPRESENT`
`TMX_ERR_ISTEMAP`
`TMX_ERR_ADDR`

Valid States ACTIVE, INACTIVE

Side Effects May inhibit the flow of active data

Clearing/Setting MX12 Devices Loopbacks: temuxLoopMX12

This function clears or sets loopbacks within the MX12 section of the device. It is up to the user to perform any tests on the looped data.

Prototype `INT4 temuxLoopMX12 (sTMX_HNDL devId, UINT2 mxNum, UINT2 ds0Num, UINT2 up, BOOLEAN ais)`

Inputs

`devId`	: device handle (from `temuxAdd`)
`mxNum`	: framer number (1-7)
`ds0Num`	: channel number (1-4)
`up`	: up/down flag
`ais`	: add AIS when looping up flag

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_HNDL`
`TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_NOTREADY`
`TMX_ERR_ISPRESENT`
`TMX_ERR_ISSTART`
`TMX_ERR_ADDR`
`TMX_ERR_ARG`

Valid States ACTIVE, INACTIVE

Side Effects May inhibit the flow of active data

Clearing/Setting MX23 Devices Loopbacks: temuxLoopMX23

This function clears or sets loopbacks within the MX23 section of the device. It is up to the user to perform any tests on the looped data.

Prototype
```
INT4 temuxLoopMX23 (sTMX_HNDL devId, UINT2
mxNum, UINT2 up, BOOLEAN ais)
```

Inputs

`devId`	: device handle (from `temuxAdd`)
`mxNum`	: framer number (1-7)
`up`	: up/down flag
`ais`	: add AIS when looping up flag

Outputs None

Returns Success = `TMX_OK`
Failure = `TMX_ERR_HNDL`
`TMX_ERR_CLOSED`
`TMX_ERR_INVALID`
`TMX_ERR_NOTREADY`
`TMX_ERR_ISPRESENT`
`TMX_ERR_ISSTART`
`TMX_ERR_ADDR`
`TMX_ERR_ARG`

Valid States ACTIVE, INACTIVE

Side Effects May inhibit the flow of active data

5.8 Callback Functions

The TEMUX/TEMAP/TECT3 driver has the capability to callback functions within the user code when certain events occur. These events and their associated callback routine declarations are detailed below. There is no user code action that is required by the driver for these callbacks - the user is free to implement these callbacks in any manner or else they can be deleted from the driver.

Reporting IO Events: sysTemuxCBackIO

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

Prototype
```
void sysTemuxCBackIO (void *usrCtxt
TMX_DPR_EVENT event, sTMX_DPV_IO *pDPV)
```

Inputs

`usrCtxt`	: user context (passed in `temuxAdd`)
`event`	: event that triggered the callback
`pDPV`	: formatted event buffer

Outputs None

Returns None

Reporting DS3 Events: sysTemuxCBackDS3

This callback function is provided by the user and is used by the DPR to report significant DS3 Section events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements. The user should free the DPV buffer.

Prototype
```
void sysTemuxCBackDS3 (void *usrCtxt
TMX_DPR_EVENT event, sTMX_DPV_DS3 *pDPV)
```

Inputs

`event`	: event that triggered the callback
`pDPV`	: formatted event buffer
`usrCtxt`	: user context (passed in `temuxAdd`)

Outputs None

Returns None

Reporting Framer Events: sysTemuxCBackFramer

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

Prototype

```
void sysTemuxCBackFramer (void *usrCtxt
TMX_DPR_EVENT event, sTMX_DPV_FRAMER *pDPV)
```

Inputs

`event`	: event that triggered the callback
`pDPV`	: formatted event buffer
`usrCtxt`	: user context (passed in `temuxAdd`)

Outputs None

Returns None

Reporting Mapper Events: sysTemuxCBackMapper (TEMUX/TEMAP only)

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

Prototype

```
void sysTemuxCBackMapper (void *usrCtxt
TMX_DPR_EVENT event, sTMX_DPV_MAPPER *pDPV)
```

Inputs

`event`	: event that triggered the callback
`pDPV`	: formatted event buffer
`usrCtxt`	: user context (passed in `temuxAdd`)

Outputs None

Returns None

6 HARDWARE INTERFACE

The TEMUX/TEMAP/TECT3 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 writing a function for each item listed. Care should be taken when matching parameters and return values.

6.1 Platform Specific MACROs

Reading a Device Register: sysTemuxSafeRead

The first read the driver makes to a newly added device allows the driver to check on the presence of that device via a 'safe' read. If the read fails, the driver will not continue to add the device and will return an error to the application. This macro should be `UINT1` oriented and should be defined by the user to reflect the target system's addressing logic.

Prototype `sysTemuxSafeRead (address, pData)`

Inputs

`address`	: register location to be read
`pData`	: (pointer to) user's variable

Outputs None

Returns Success = `0x00`
Failure = <any other value>

Reading from Registers: sysTemuxReadReg

This macro reads the contents of a specific register location. This macro should be `UINT1` oriented and should be defined by the user to reflect the target system's addressing logic. There is no need for error recovery in this function.

Prototype `sysTemuxReadReg (address)`

Inputs

`address`	: register location to be read

Outputs None

Returns value read from the addressed register location

Writing Register Values: sysTemuxWriteReg

This macro writes the supplied value to the specific register location. This macro should be `UINT1` oriented and should be defined by the user to reflect the target system's addressing logic. There is no need for error recovery in this function.

Prototype `sysTemuxWriteReg (address, data)`

Inputs

`address`	: register location to be written
`data`	: data to be written

Outputs None

Returns value written to the addressed register location

6.2 Interrupt Servicing

General ISR Routines

The porting of an ISR routine between platforms is a rather difficult task. There are so many different implementations of the hardware level routines that creating a universal routine is impossible.

In this driver, the user is responsible for creating a 'shell' (`sysTemuxISRHandler`) handler that in turn calls an API function, `temuxISR` , once for EACH device requesting service, to perform the ISR related housekeeping that is required by the device. This method was chosen because it places the burden of determining which device(s) is(are) requesting service on the user, rather than attempting to incorporate the many possible hardware scenarios into the driver.

During execution of the API function `temuxModuleStart` (`temuxModuleStop`), the driver informs the application that it is time to install (uninstall) this 'shell' via the user supplied function: `sysTemuxISRHandlerInstall` (`sysTemuxISRHandlerRemove`).

Note: A device can be initialized with interrupts disabled. In that mode, a polling routine can be invoked independently that in-turn processes the interrupt status in the device.

Installing Interrupt Handlers: sysTemuxISRHandlerInstall

This function installs the user-supplied Interrupt-Service Routine, `sysTemuxISRHandler`, into the processor's interrupt vector table.

Prototype `sysTemuxISRHandlerInstall (temuxISR)`

Inputs

`temuxISR`	: (pointer to) the function `temuxISR`

Outputs None

Returns Success = `0x00`
Failure = <any other value>

Invoking Interrupt Handlers: sysTemuxISRHandler

This routine is invoked when one or more TEMUX/TEMAP/TECT3 devices raise the interrupt line to the microprocessor. This routine invokes the driver-provided routine, `temuxISR`, for each device registered with the driver.

Prototype `sysTemuxISRHandler (void)`

Inputs None

Outputs None

Returns None

Removing Interrupt Handlers: sysTemuxISRHandlerRemove

This function removes the user-supplied Interrupt-Service Routine, `sysTemuxISRHandler`, from the processor's interrupt vector table.

Prototype `sysTemuxISRHandlerRemove (void)`

Inputs None

Outputs None

Returns None

Installing DPRTask: sysTemuxDPRTaskInstall

The driver calls this user-supplied function to inform the application that it is time to initialize and/or install the user-supplied function `sysTemuxDPRTask`. Note: In most cases, the user will install `sysTemuxDPRTask` as a task but that is not required.

Prototype `sysTemuxDPRTaskInstall (temuxDPR)`

Inputs `temuxDPR` : (pointer to) the function `temuxDPR`

Outputs None

Returns Success = 0x00
Failure = <any other value>

DPR Task: sysTemuxDPRTask

This user-supplied function is installed (generally as a separate task within the RTOS) when the driver invokes the `sysTemuxDPRTaskInstall` call. It runs periodically and is responsible for calling the API function `temuxDPR`. The user may choose to have this task perform the `sysTemuxBufferReceive` and `sysTemuxISVBufferRtn` functions (passing the ISV as a parameter to `temuxDPR`) or the user may pass a NULL to `temuxDPR`, in which case `temuxDPR` will receive and deallocate the ISV buffer.

Prototype `sysTemuxDPRTask (void)`

Inputs None

Outputs None

Returns None

Removing DPRTask: sysTemuxDPRTaskRemove

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

Prototype `sysTemuxDPRTaskRemove (void)`

Inputs None

Outputs None

Returns None

7 RTOS INTERFACE

The TEMUX/TEMAP/TECT3 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

Allocating Memory: sysTemuxMemAlloc

This macro allocates a specified number of bytes of memory.

Format `sysTemuxMemAlloc (number)`

Inputs `number` : number of bytes to be allocated

Outputs None

Returns Success = (pointer to) first byte of allocated memory
Failure = NULL (pointer)

Freeing Allocated Memory: sysTemuxMemFree

This macro frees memory allocated using `sysTemuxMemAlloc`.

Format `sysTemuxMemFree (address)`

Inputs `address` : (pointer to) first byte of the memory region being de-allocated

Outputs None

Returns None

7.2 Buffer Management

All operating systems provide some sort of buffer system, particularly for use in sending and receiving messages. Most operating systems provide both an internal set of buffers (usually small in size) and functions that allow the user to create additional buffer pools (especially when the buffer size needs to be large). It is the intention of this driver to use both types of buffers, the ISV being the large buffer that carries the entire exception state of the device and the DPV being the small buffer that carries individual block event flags and other simple bits of information.

The following calls, provided by the user, allow the driver to get and return these buffers from/to the RTOS. It is the user's responsibility to create any special resources or pools to handle buffers of these sizes. This creation is done by the application when the driver calls the user-supplied function `sysTemuxBufferStart`.

Starting Buffers: sysTemuxBufferStart

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

Prototype `sysTemuxBufferStart (void)`

Inputs None

Outputs None

Returns Success = `0x00`
Failure = <any other value>

Getting DPV Buffers: sysTemuxDPVBufferGet

This function gets a buffer from the RTOS that will be used by the DPR code to create a DPR 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.

Prototype `sysTemuxDPVBufferGet (void)`

Inputs None

Outputs None

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

Getting ISV Buffers: sysTemuxISVBufferGet

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

Prototype `sysTemuxISVBufferGet (void)`

Inputs None

Outputs None

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

Returning DPV Buffers: sysTemuxDPVBufferRtn

This function returns a DPV buffer to the RTOS when the information in the block is no longer needed. This buffer is usually returned to the buffer pool by the application, during the processing of a callback function.

Prototype `sysTemuxDPVBufferRtn (sTMX_DPV *pDPV)`

Inputs `pDPV` : (pointer to) a DPV Buffer

Outputs None

Returns None

Returning ISV Buffers: sysTemuxISVBufferRtn

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

Prototype `sysTemuxISVBufferRtn (sTMX_ISV *pISV)`

Inputs `pISV` : (pointer to) a ISV Buffer

Outputs None

Returns None

Sending an ISV buffer to the DPR task: sysTemuxBufferSend

This function sends an ISV message to the DPR task with the device handle and interrupt statuses for that device.

Prototype `sysTemuxBufferSend (sTMX_ISV *pISV)`

Inputs `pISV` : (pointer to) a ISV Buffer

Outputs None

Returns Success = 0
Failure = any non-zero value

Receiving an ISV buffer: sysTemuxBufferReceive

This function receives an ISV buffer from the RTOS. It is meant to be used by the DPR task to receive ISV messages from the ISR.

Prototype `sysTemuxBufferReceive (void)`

Inputs `None`

Outputs None

Returns Success = pointer to an ISV
Failure = NULL

Stopping ISV/DPV Buffers: sysTemuxBufferStop

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

Prototype `sysTemuxBufferStop (void)`

Inputs None

Outputs None

Returns None

8 PORTING DRIVERS

This section outlines how to port the TEMUX/TEMAP/TECT3 device driver to your hardware and RTOS platform. However, this manual can offer only guidelines for porting the driver because each platform and application is unique.

8.1 Driver Source Files

The C source files listed in Table 33 contain the code for the TEMUX/TEMAP/TECT3 driver. You may need to modify the code or develop additional code. The code is in the form of constants, macros, and functions. For the ease of porting, the code is grouped into source files (`src`) and include files (`inc`). The `src` files contain the functions and the `inc` files contain the constants and macros.

Table 33: Driver Source Files

`src`	`tmx_api.c` (contains all API functions)
	`tmx_util.c` (contains driver internal functions)
	`tmx_diag.c` (contains diagnostic functions)
	`tmx_hw.c` (contains hardware interface functions)
	`tmx_stat.c` (alarms status and statistics functions)
	`tmx_rtos.c` (contains RTOS interface functions)
	`tmx_app.c` (contains sample driver callback functions and sample code)
	`tmx_isr.c` (ISR functions)
	`tmx_dpr.c` (DPR functions)
`inc`	`tmx_api.h` (contains data-structure definitions and prototypes)
	`tmx_hw.h` (contains hardware-interface macro and constant definitions)
	`tmx_rtos.h` (contains RTOS-interface macro and constant definitions)
	`tmx_app.h` (contains data structure definitions and prototypes of sample code)
	`tmx_dev.h` (the device register mapping)
	`tmx_mdb.h` (the layout of the MDB)

	`temux.h` (contains general defines for compiling)
	`tmx_isr.h` (contains ISR definitions and prototypes)
	`tmx_dpr.h` (contains DPR definitions and prototypes)
	`tmx_util.h` (contains driver internal function definitions and prototypes)
`example`	`app.c` (simple example code for starting up the driver)
	`app.h` (declarations for the sample code)
	`hw_pci.h` (example hardware porting file for an Intel x86 Compact PCI based system)
	`rtos_vxw.h` (example RTOS porting file for the vxWorks RTOS)

8.2 Driver Porting Procedures

The following procedures summarize how to port the TEMUX/TEMAP/TECT3 driver to your platform. The subsequent sections describe these procedures in more detail.

To port the TEMUX/TEMAP/TECT3 driver to your platform:

Step 1: Port the driver's RTOS extensions (below):

Step 2: Port the driver to your hardware platform (on page 90):

Step 3: Port the driver's application-specific elements (on page 90):

Step 4: Build the driver (on page 91).

Step 1: Porting Driver RTOS Extensions

The RTOS extensions encapsulate RTOS specific services and data types used by the driver. The `temux.h` file contains data types and compiler-specific data-type definitions. The file `tmx_rtos.h` contains macros for RTOS specific services used by the RTOS extensions. These RTOS extensions include:

- Task management
- Message queues
- Memory Management

In addition, you may need to modify functions that use RTOS specific services, such as utility and interrupt-event handling functions. The `tmx_util.c` and `tmx_isr.c` files contain the utility and interrupt-event handler functions that use RTOS specific services.

To port the driver's RTOS extensions:

1. Modify the data types in `temux.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 `tmx_rtos.h`. Redefine the following macros to the corresponding system calls that your target system supports:

Service Type	Macro Name	Description
Memory	`sysTemuxMemAlloc`	Allocates the memory block
	`sysTemuxMemFree`	Frees the memory block
Task	`sysTemuxDPRTaskInstall`	Installs the DPR task in the RTOS
	`sysTemuxDPRTaskRemove`	Removes the DPR task from the RTOS

3. Modify the following RTOS specific function in tmx_rtos.c:

Service Type	Function Name	Description
Buffer	`sysTemuxBufferStart`	Starts buffer management
	`sysTemuxBufferStop`	Stops buffer management
	`sysTemuxBufferSend`	Sends a buffer to the DPR task
	`sysTemuxBufferReceive`	Receives a buffer from the RTOS
	`sysTemuxISVBufferGet`	Gets an ISV buffer from the ISV buffer queue
	`sysTemuxISVBufferRtn`	Returns an ISV buffer to the ISV buffer queue
	`sysTemuxDPVBufferGet`	Gets a DPV buffer from the DPV buffer queue
	`sysTemuxDPVBufferRtn`	Returns a DPV buffer to the DPV buffer queue

Step 2: Porting Drivers to Hardware Platforms

This section describes how to modify the TEMUX/TEMAP/TECT3 driver for your hardware platform.

To port the driver to your hardware platform:

1. Modify the low-level device read/write macros in the `tmx_hw.h` file.

Service Type	Function Name	Description
Device I/O	`sysTemuxReadReg`	Reads a device register given its real address in memory
	`sysTemuxWriteReg`	Writes to a device register given its real address in memory
	`sysTemuxSafeRead`	Reads a device register in 'safe' fashion when adding a device
Interrupt	`sysTemuxISRHandlerInstall`	Installs the interrupt handler for the RTOS
	`sysTemuxISRHandlerRemove`	Removes the interrupt handler from the RTOS
	`sysTemuxISRHandler`	Interrupt handler for the TEMUX/TEMAP/TECT3 device
	`sysTemuxDPRTask`	Task that calls the TEMUX/TEMAP/TECT3 DPR

Step 3: Porting Driver Application-Specific Elements

Application specific elements are configuration constants used by the API for developing an application. This section describes how to modify the application specific elements in the TEMUX/TEMAP/TECT3 driver.

To port the driver's application-specific elements:

1. Modify the type definition for the user context. The user context is used to identify a device in your application callbacks.

2. Modify the value of the base error code (`TMX_ERR_BASE`) in `temux.h`. This ensures that the driver error codes do not overlap other error codes used in your application.

3. Define the application-specific constants for your hardware configuration in `tmx_api.h`:

Device Constant	Description	Default
TMX_MAX_DEVICES	The maximum number of TEMUX/TECT3/TEMAP devices that can be supported by this driver	48
TMX_MAX_IPROFILES	The maximum number of device initialization profiles	16

4. Code the callback functions according to your application. There are four sample callback functions in the tmx_app file. You can use these callback functions or you can customize them before using the driver. The driver will call these callback functions when an event occurs on the device. These functions must conform to the following prototype:

   ° `void* sysTemuxCBackXX (void *usrCtxt, TMX_DPR_EVENT event, sTMX_DPV_XX *pDPV)`

Step 4: Building the Driver

This section describes how to build the TEMUX/TEMAP/TECT3 driver.

To build the driver:

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

2. Choose from among the different compile options supported by the driver as per your requirements.

3. Compile the source files and build the TEMUX/TEMAP/TECT3 API driver library using your make utility.

4. Link the TEMUX/TEMAP/TECT3 API driver library to your application code.

APPENDIX A: CODING CONVENTIONS

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

Variable Type Definitions

Table 34: 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
BOOLEAN	unsigned integer – 2 bytes

Naming Conventions

Table 35 presents a summary of the naming conventions followed by 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 35: Naming Conventions

Type	Case	Naming convention	Examples
Macros	Uppercase	prefix with "m" and device abbreviation	mTMX_WRITE
Constants	Uppercase	prefix with device abbreviation	TMX_REG
Structures	Hungarian Notation	prefix with "s" and device abbreviation	sTMX_DDB
API Functions	Hungarian Notation	prefix with device name	temuxAdd()
Porting Functions	Hungarian Notation	prefix with "sys" and device name	sysTemuxReadReg()
Other Functions	Hungarian Notation		myOwnFunction()
Variables	Hungarian Notation		maxDevs
Pointers to variables	Hungarian Notation	prefix variable name with "p"	pmaxDevs
Global variables	Hungarian Notation	prefix with device name	temuxMDB

Macros

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

- 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: mTEMUX_WRITE is a valid name for a macro.

Constants

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

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

Structures

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

- 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: sTMX_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: temuxAdd() is a valid name for an API function.

Porting Functions

Porting functions correspond to functions that are hardware and/or RTOS dependant.

- 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: sysTemuxReadReg() is a hardware/RTOS specific.

Other Functions

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

Variables

Naming of variables must follow the Hungarian notation.

- A pointer to a variable shall use "`p`" as a prefix followed by the variable name unchanged. If the variable name already starts with a "`p`", the first letter of the variable name may be capitalized, but this is not a requirement. Double pointers might be prefixed with "`pp`", but this is not required.
- Global variables must be identified with the device's name in lowercase as a prefix.
- Examples: `maxDevs` is a valid name for a variable, `pmaxDevs` is a valid name for a pointer to `maxDevs`, and `temuxBaseAddress` 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.

APPENDIX B: TEMUX/TECT3/TEMAP ERROR CODES

This appendix describes the error codes used in the TEMUX/TECT3/TEMAP device driver.

Table 36: TEMUX/TECT3/TEMAP Error Codes

Error Code	Description
TMX_OK	Success
TMX_FAIL	Failure
TMX_ERR_RTOS	RTOS system call failure
TMX_ERR_ALLOC	Memory allocation failure
TMX_ERR_BUFFER	Buffer management errror
TMX_ERR_STOP	Error stopping module
TMX_ERR_ISOPEN	Module is already open
TMX_ERR_CLOSED	Module is already closed
TMX_ERR_ADD	Error adding device
TMX_ERR_RESET	Error resetting device
TMX_ERR_DEACTIVATE	Error deactivating device
TMX_ERR_ISIDLE	Module state is already idle
TMX_ERR_ISREADY	Module state is already ready
TMX_ERR_ISSTART	Device state is already start
TMX_ERR_ISPRESENT	Device state is already present
TMX_ERR_NOTIDLE	Module state is not idle
TMX_ERR_NOTREADY	Module state is not ready
TMX_ERR_NOTACTIVE	Device state is not active
TMX_ERR_NOTPRESENT	Device state is not present
TMX_ERR_NOTINACTIVE	Device state is not inactive
TMX_ERR_ARG	Bad argument passed to function

Error Code	Description
TMX_ERR_CFG	Device configuration error
TMX_ERR_ADDR	Bad address passed to function
TMX_ERR_HNDL	Bad handle passed to function
TMX_ERR_MODE	Illegal ISR mode
TMX_ERR_RANGE	Address passed to function is out of range
TMX_ERR_HWFAIL	Error accessing device
TMX_ERR_INVALID	Module is invalid
TMX_ERR_ISTEMAP	Operation not allowed on TEMAP device
TMX_ERR_MAXPROF	Maximum number of init profiles already added
TMX_ERR_MAXDEVICE	Maximum devices have already been added
TMX_ERR_TECT3	Operation not allowed on TECT3 device

ACRONYMS

ALMI: Alarm Integrator

API: Application Programming Interface

APRM: Performance Report

BERT: Bit Error-Rate Test

BOC: Bit Oriented Code (RBOC, XBOC)

D3MA: DS3 Add-side Mapper

D3MD: DS3 Drop-side Mapper

DDB: Device Data Block

DIV: Device Initialization Vector

DLC: Data Link Controller (RDLC, TDLC)

DPR: Deferred-Processing Routine

DSB: Device Status Block

ELST: Elastic Store (`RX-ELST, TX_ELST`)

FCS: Frame Check Sequence

FEAC: Far End Alarm & Control

FIFO: First In, First Out

FRMR: Framer (usually the Receive Side)

HDLC: High-level Data Link Control

ISR: Interrupt-Service Routine

JAT: Jitter Attenuator (RJAT, TJAT)

MDB: Module Data Block

MIV: Module Initialization Vector

MSB: Module Status Block

MVIP: Multi-vendor integration protocol

PCI: Processor Connection Interface

PMON: Performance Monitor

PRBS: Pseudo Random Bit Sequence

PRGD: Pseudo Random (pattern) Generator & Detector

PSC: Per-Channel Serial Controller (RPSC, TPSC)

RHDL: Receive HDLC processor

RTDM: Receive Tributary DeMapper

RTOS: Real-Time Operating System

SBI: Scaleable Bandwidth Interconnect (INSBI (ingress), EXSBI (egress))

TAPI: Transmit Any-PHY packet interface

THDL: Transmit HDLC processor

TMP: Tributary Mapper (TTMP)

TOP: Tributary (path) Overhead Processor (RTOP, TTOP)

TRAN: Transmitter (of a Framer, usually)

TRAP: Transmit Alarm Processor

VTPP: Tributary Payload Processor (I-VTPP (ingress), E-VTPP (egress))

LIST OF TERMS

APPLICATION: Refers to protocol software used in a real system as well as validation software written to validate the TEMUX/TEMAP/TECT3 driver on a validation platform.

API (Application Programming Interface): Describes the connection between this module and the user's application code.

INGRESS: An older term for the line side of the device. The line side usually contains the larger aggregate connections and usually connects to the WAN portion of a network.

EGRESS: An older term for the system side of the device. The system side usually contains the smaller individual connections and usually connects to the LAN portion of a network

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

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

DEVICE : One TEMUX/TEMAP/TECT3 integrated circuit. There can be many devices, all served by this one driver module

- DIV (Device Initialization Vector): Structure passed from the API to the device during initialization; it contains parameters that identify the specific modes and arrangements of the physical device being initialized.
- DDB (Device Data Block): Structure that holds the configuration data for each device.
- DSB (Device Status Block): Structure that holds the alarms, status, and statistics 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 TEMUX/TEMAP/TECT3 chips.

- MIV (Module Initialization Vector): Structure passed from the API to the module during initialization, it contains parameters that identify the specific characteristics of the driver module being initialized.
- MDB (Module Data Block): Structure that holds the configuration data for this module.
- MSB (Module Status Block): Structure that holds the alarms, status and statistics for the module
- RTOS (Real-Time Operating System): The host for this driver

INDEX