PM5363

TUPP+622

SONET/SDH TRIBUTARY UNIT PAYLOAD PROCESSOR

DRIVER MANUAL

DOCUMENT ISSUE 2

ISSUED: FEBRUARY, 2001

ABOUT THIS MANUAL AND TUPP+622

This manual describes the TUPP+622 device driver. It describes the driver's functions, data structures, and architecture. This manual focuses on the driver's interfaces to your application, Real-Time Operating System, and the devices. It also describes in general terms how to modify and port the driver to your software and hardware platform.

Audience

This manual was written for people who need to:

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

References

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

Table 1: Related Documents

Document Name	Document Number
TUPP+622 Telecom Standard Product Data Sheet	PMC-1981421
SONET/SDH Tributary Unit Payload Processor / Monitor for 622 Mbit/s Interfaces (TUPP+622) Short Form Data Sheet	PMC-1981272

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

Revision History

Issue No.	Issue Date	Details of Change
Issue 1	January 2000	Document created
Issue 2	February 2001	1) Modified the alarm, status and statistics architecture (structures and APIs): a) removed MSB and DSB structures as well as `tuppClearStats()` API since statistics are no longer accumulated inside the driver. b) Added `sTUP_STATUS_XX` and `sTUP_CNT_XX` structures to add granularity. c) replaced `tuppGetStats()` API with `tuppGetCnt()` and `tuppGetStatus()` APIs. 2) Fixed various typos and formatting issues.”

Legal Issues

PMC-1991288 (R2), ref PMC-990877 (R2)

Contacting PMC-Sierra

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

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

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

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 DRIVER PORTING QUICK START

This section summarizes how to port the TUPP+622 device driver to your hardware and operating system (OS) platform. For more information about porting the TUPP+622 driver, see Section 8 (page 73). Since each platform and application is unique, this manual can only offer guidelines for porting the TUPP+622 driver.

The code for the TUPP+622 driver is organized into C source files. You may need to modify the code or develop additional code. The code is in the form of constants, macros, and functions. For ease of porting, the code is grouped into source files (`src`) and includes files (`inc`). The source files contain the functions and the include files contain the constants and macros.

To port the TUPP+622 driver to your platform:

1. Port the driver's OS extensions (page 75):
   - Data types
   - OS-specific services
   - Utilities and interrupt services that use OS-specific services
2. Port the driver to your hardware interface (page 76):
   - Port low-level device read-and-write macros.
   - Define hardware system-configuration constants.
3. Port the driver's application-specific elements (page 77):
   - Define the task-related constants.
   - Code the callback functions.
4. Build the driver (page 77).

2 DRIVER FUNCTIONS AND FEATURES

This section describes the main functions and features supported by the TUPP+622 driver.

Table 1: Driver Functions and Features

Function	Description
Open / Close Driver Module (page 42)	Opening the driver module allocates all the memory needed by the driver and initializes all module level data structures. Closing the driver module shuts down the driver module gracefully after deleting all devices that are currently registered with the driver, and releases all the memory allocated by the driver.
Start / Stop Driver Module (page 43)	Starting the driver module involves allocating all RTOS resources needed by the driver such as timers and semaphores (except for memory, which is allocated during the Open call). Stopping the driver module involves de-allocating all RTOS resources allocated by the driver without changing the amount of memory allocated to it.
Add / Delete Device (page 45)	Adding a device involves verifying that the device exists, associating a device Handle to the device, and storing context information about it. The driver uses this context information to control and monitor the device. Deleting a device involves shutting down the device and clearing the memory used for storing context information about this device.
Device Initialization (page 46)	The initialization function resets then initializes the device and any associated context information about it. The driver uses this context information to control and monitor the TUPP+622 device.
Activate / De-Activate Device (page 48)	Activating a device puts it into its normal mode of operation by enabling interrupts and other global registers. A successful device activation also enables other API invocations. On the contrary, de-activating a device removes it from its operating state, disables interrupts and other global registers.
Read / Write Device Registers (page 49)	These functions provide a ‘raw’ interface to the device. Device registers that are both directly and indirectly accessible are available for both inspection and modification via these functions. If applicable, block reads and writes are also available.

Function	Description
Interrupt Servicing / Polling (page 58)	Interrupt Servicing is an optional feature. The user can disable device interrupts and instead poll the device periodically to monitor status and check for alarm/error conditions. Both polling and interrupt driven approaches detect a change in device status and report the status to a Deferred-Processing Routine (DPR). The DPR then invokes application callback functions based on the status information retrieved. This allows the driver to report significant events that occur within the device to the application.
Statistics Collection (page 60)	Functions are provided to retrieve a snapshot of the various counts that are accumulated by the TUPP+622 device. Routines should be invoked often enough to avoid letting the counters to rollover.

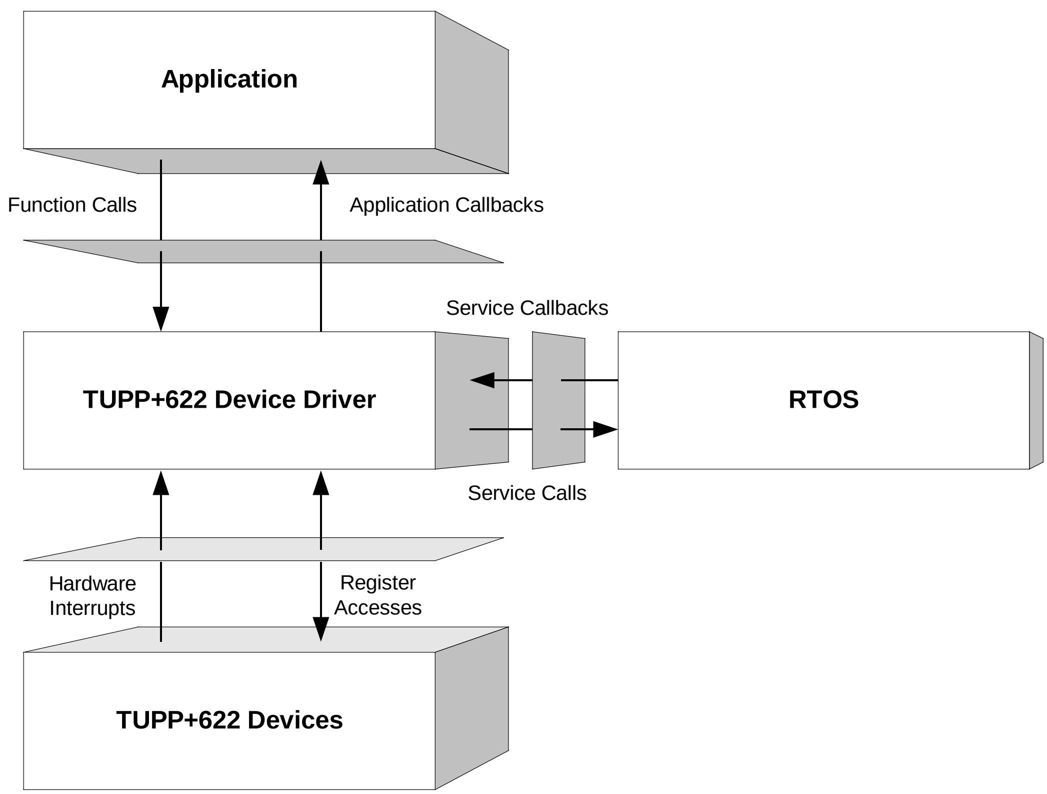
3 SOFTWARE ARCHITECTURE

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

3.1 Driver External Interfaces

Figure 1 illustrates the external interfaces defined for the TUPP+622 device driver.

Figure 1: Driver Interfaces



Application Programming Interface

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

- Initializing the device
- Validating device configuration

- Retrieving device status and statistics information.
- Diagnosing the device

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

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

Real-Time OS Interface

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

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

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

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

Driver Hardware Interface

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

3.2 Main Components

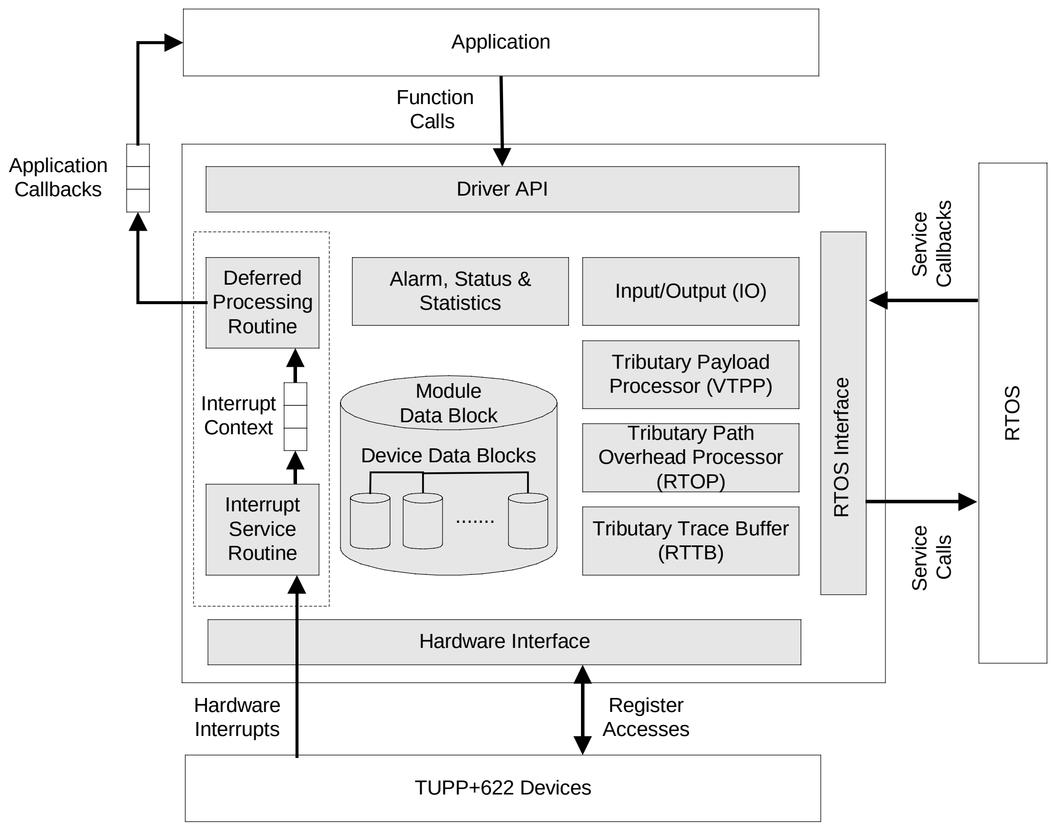
Figure 2 illustrates the top level architectural components of the TUPP+622 device driver. This applies in polled and interrupt driven operation. In polled operation the ISR is called periodically. In interrupt operation the interrupt directly triggers the ISR.

The driver includes the following main components:

- Module and Device(s) Data-Blocks
- Interrupt-Processing Routine
- Deferred-Processing Routine
- Alarm, Status and Statistics

- Input/Output
- Tributary Payload Processor (VTPP)
- Tributary Path Overhead Processor (RTOP)
- Tributary Trace Buffer (RTTB)

Figure 2: Driver Architecture



Alarms, Status and Statistics

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

Input / Output (IO)

The Input / Output is responsible for configuring the input de-multiplexer and output multiplexer. Functions are provided for monitoring the major TUPP+622 inputs.

Tributary Payload Processor (VTPP)

The Tributary Payload Processor (VTPP) detects and reports the path overhead errors. Functions are provided for configuring the J1 position. For diagnostics purposes at the tributary level, functions are also provided to force insertion of path AIS, path Idle, as well as inversion of the NDF field in the outgoing payload.

Tributary Path Overhead Processor (RTOP)

The Tributary Path Overhead Processor (RTOP) detects and report REI, RDI and RFI. Functions are provided to monitor the tributary performance by giving access to the REI and BIP-2 error counts. A function is also provided to give an easy read/write access to the Path Signal Label for each tributary.

Tributary Trace Buffer (RTTB)

Functions are provided to read and write the expected and captured tributary path trace message (J2).

Module Data Block (MDB)

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

Device Data Blocks (DDB)

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

Interrupt-Service Routine

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

The low-level interrupt-handler function that traps the hardware interrupt and calls `tuppISR` 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 66. You can customize these example implementations to suit your specific needs.

Deferred-Processing Routine

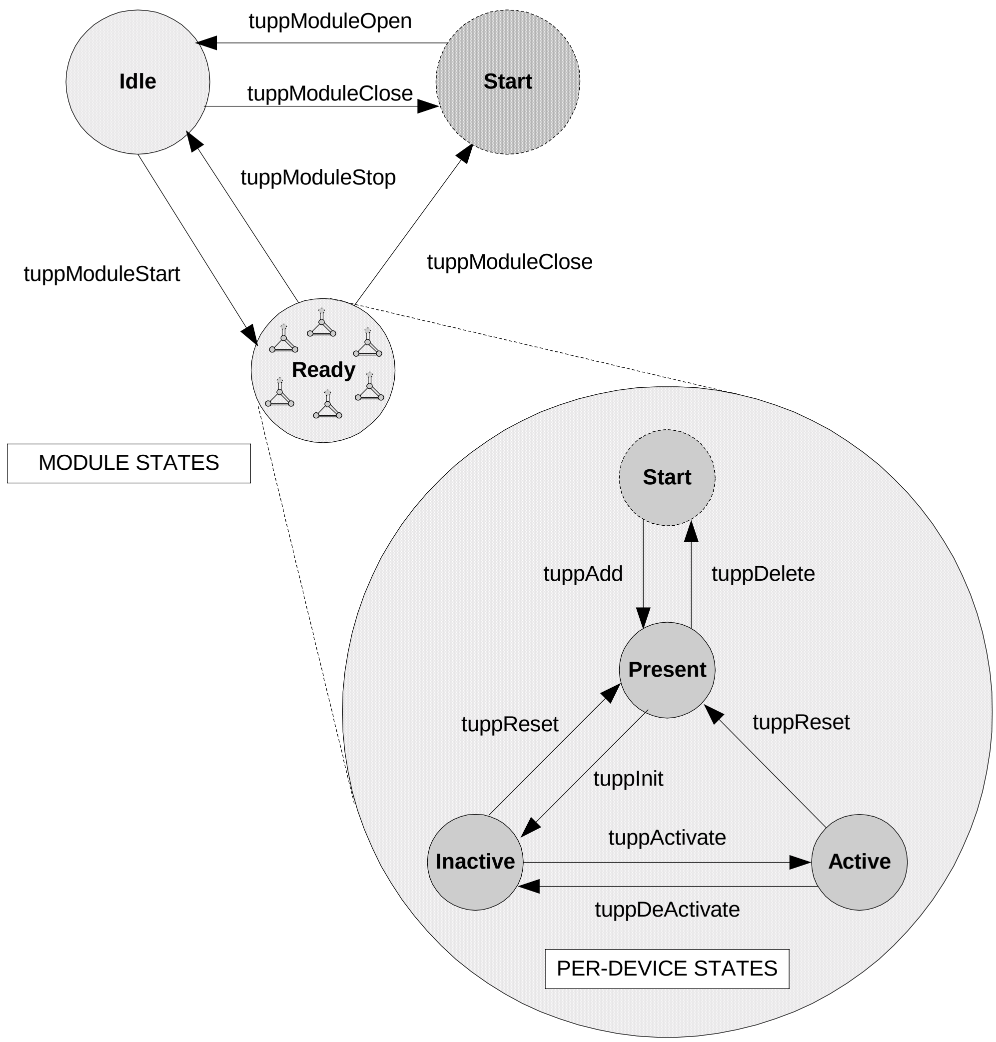
The DPR provided by the TUPP+622 driver (`tuppDPR`) clears and processes interrupt conditions for the device. Typically, a system specific function, which runs as a separate task within the RTOS, executes the DPR.

See page 24 for a detailed explanation of the DPR and interrupt-servicing model.

3.3 Software States

Figure 3 shows the software state diagrams for the TUPP+622 module and device(s) as maintained by the driver. State transitions occur on the successful execution of the corresponding transition functions shown. State information helps maintain the integrity of the driver's DDB by controlling the set of device operations allowed in each state.

Figure 3: Driver Software States



Module States

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

Start

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

Idle

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

Ready

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

Device States

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

Start

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

Present

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

Inactive

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

Active

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

3.4 Processing Flows

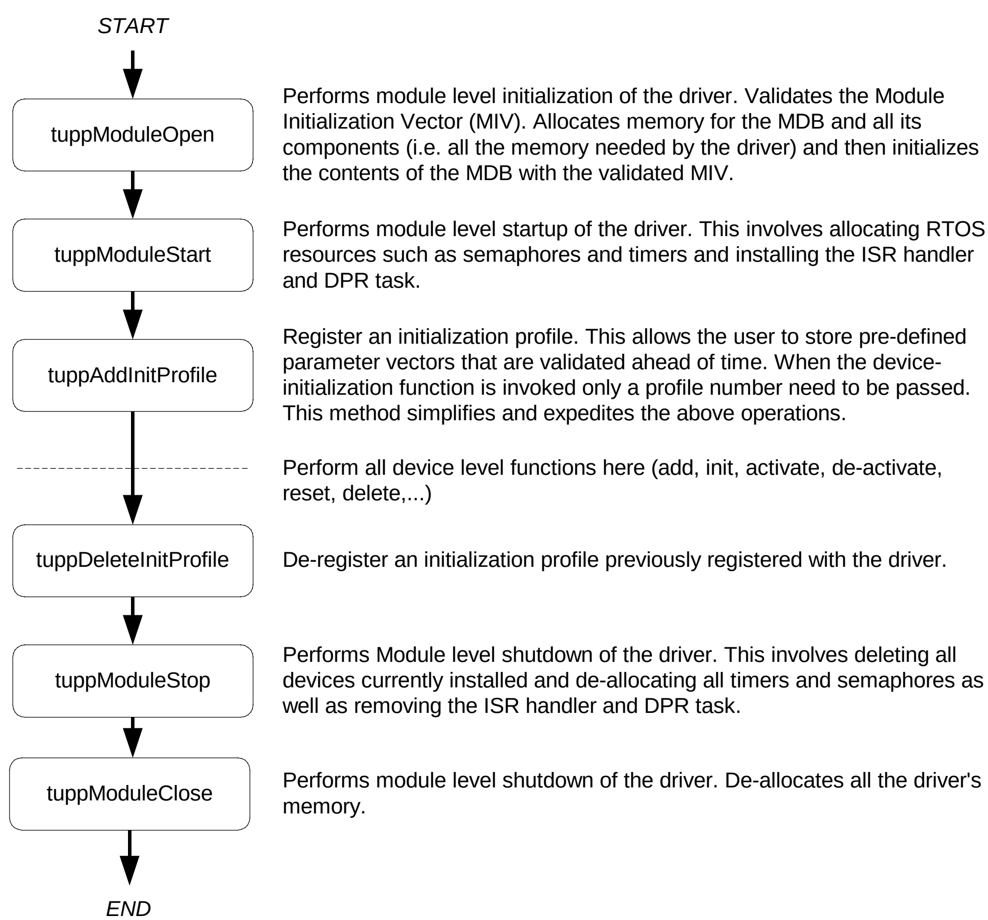
This section describes the main processing flows of the TUPP+622 driver modules.

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

Module Management

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

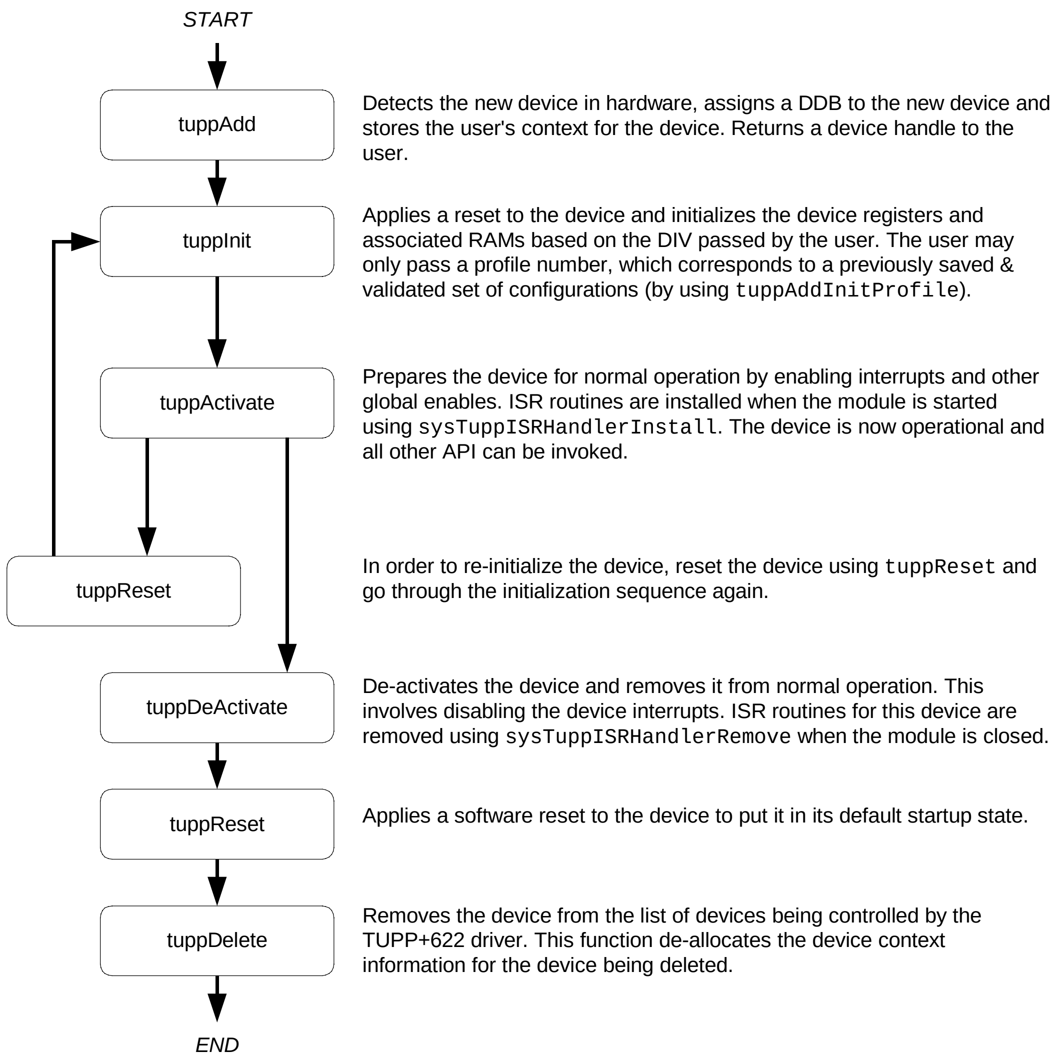
Figure 4: Module Management Flow Diagram



Device Management

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

Figure 5: Device Management Flow Diagram



3.5 Interrupt Servicing

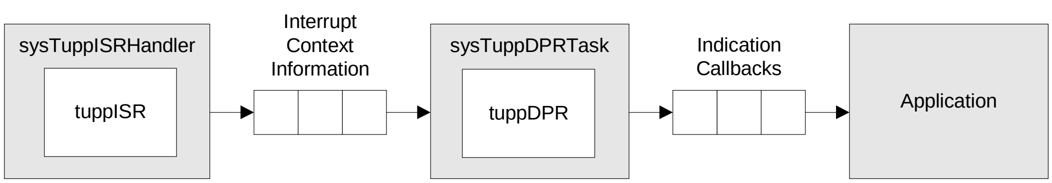
The TUPP+622 driver services device interrupts using an Interrupt-Service Routine (ISR) that traps interrupts and a Deferred-Processing Routine (DPR) that actually processes the interrupt conditions and clears them. This lets the ISR execute quickly and exit. Most of the time-consuming processing of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

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

The driver provides the system-independent functions, `tuppISR` and `tuppDPR`. You must fill in the corresponding system-specific functions, `sysTuppISRHandler` and `sysTuppDPRTask`. The system-specific functions isolate the system-specific communication mechanism (between the ISR and DPR) from the system-independent functions, `tuppISR` and `tuppDPR`.

Figure 6 illustrates the interrupt service model used in the TUPP+622 driver design.

Figure 6: Interrupt Service Model



Note: Instead of using an interrupt service model, you can use a polling service model in the TUPP+622 driver to process the device's event-indication registers (see page 26).

Calling tuppISR

An interrupt handler function, which is system dependent, must call `tuppISR`. But first, the low-level interrupt-handler function must trap the device interrupts. You must implement this function to fit your own system. As a reference, an example implementation of the interrupt handler (`sysTuppISRHandler`) appears on page 66. You can customize this example implementation to suit your needs.

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

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

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

Calling tuppDPR

The sysTuppDPRTask 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 TUPP+622 driver. In the message-queue implementation model, this task has an associated message queue. The task waits for messages from the ISR on this message queue. When a message arrives, sysTuppDPRTask calls the DPR (tuppDPR) with the received ISV.

Then tuppDPR 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, tuppDPR 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 tuppDelete. Also, ensure that the indication function is non-blocking because the DPR task executes while TUPP+622 interrupts are disabled. You can customize these callbacks to suit your system. See page 62 for example implementations of the callback functions.

Note: Since the tuppISR and tuppDPR 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 RTOS' provide.

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

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

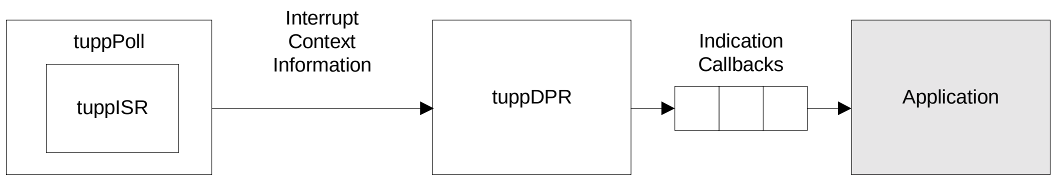
As a reference, this manual provides example implementations of the interrupt installation and removal functions on page 66. You can customize these prototypes to suit your specific needs.

Calling tuppPoll

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

Figure 7 illustrates the polling service model used in the TUPP+622 driver design.

Figure 7: Polling Service Model



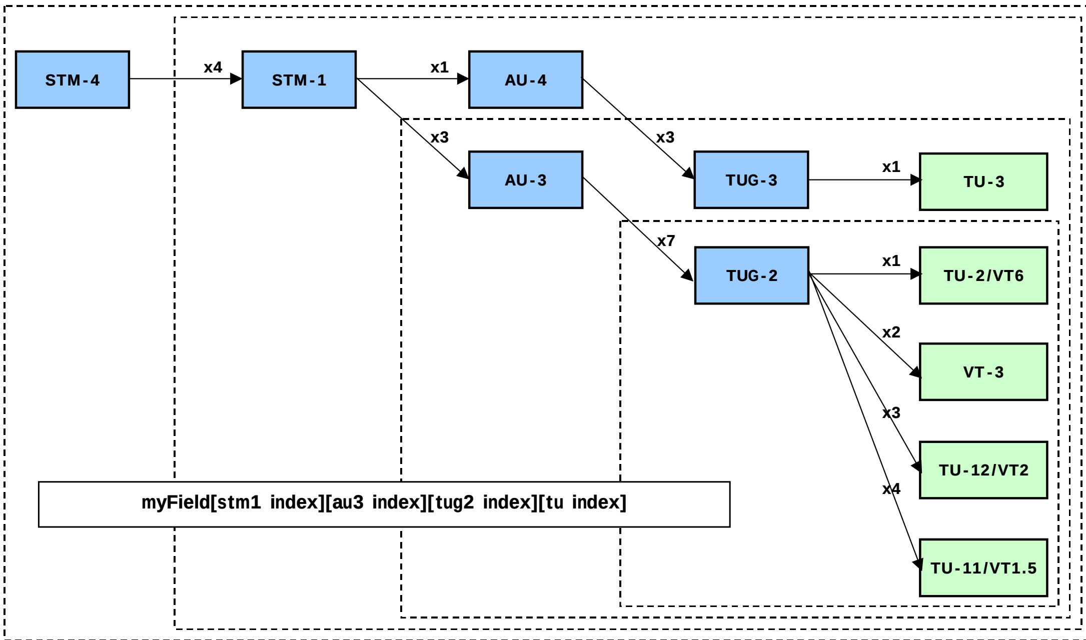
In polling mode, the application is responsible for calling `tuppPoll` often enough to service any pending error or alarm conditions. When `tuppPoll` is called, the `tuppISR` function is called internally.

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

4 DATA STRUCTURES

The TUPP+622 driver allows the User to configure the behavior of each tributary. The same structures are used independently of the mapping currently in use.

Figure 8: Simplified SDH/Sonet Multiplexing Structures



Whenever a field within a structure refers to a specific TU, this field is declared as an array of [4][3][7][4], so that you can retrieve the value of this field by providing the STM1 #, AU3 #, TUG2 #, and TU # respectively in the indexes.

If the mapping is not all TU11s, there will be some unused fields inside the structure. For example, if the mapping is all TU12s, there are only 3 TU12 per TUG2, so that all the [x][x][x][4] elements are unused and therefore invalid. If the mapping is all TU3s, a specific TU3 is defined by its STM1 #, TU3 # and therefore only the [x][x][1][1] are valid inside our structures, and all the remaining elements are unused.

Note that the driver also uses substructures inside the high-level structures so that when a field is for a specific TU, the actual [4][3][7][4] array is often separated into arrays of [4] substructures that contain fields that are arrays of [3][7][4].

4.1 Constants

The following Constants are used throughout the driver code:

- <TUPP+622 ERROR CODES> error codes used throughout the driver code, returned by the API functions and used in the global error number field of the MDB and DDB. For more information on possible driver return codes, see Appendix A (page 79).
- TUP_MAX_DEVS defines the maximum number of devices that can be supported by this driver. This constant must not be changed without a thorough analysis of the consequences to the driver code.
- TUP_MOD_START, TUP_MOD_IDLE, TUP_MOD_READY are the three possible module states (stored in stateModule).
- TUP_START, TUP_PRESENT, TUP_ACTIVE, TUP_INACTIVE are the four possible device states (stored in stateDevice).

4.2 Structures Passed by the Application

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

Module Initialization Vector: MIV

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

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

Table 2: Module Initialization Vector: sTUP_MIV

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

Device Initialization Vector: DIV

Passed via the `tuppInit` call, this structure contains all the information needed by the driver to initialize a TUPP+622 device. This structure is also passed via the `tuppAddInitProfile` call when used as an initialization profile.

- `valid` indicates that this initialization profile has been properly initialized and may be used by the user. This field should be ignored when the DIV is passed directly.
- `pollISR` is a flag that indicates the type of interrupt servicing the driver is to use. The choices are 'polling' (`TUP_POLL_MODE`), and 'interrupt driven' (`TUP_ISR_MODE`). When configured in polling the Interrupt capability of the device is NOT used, and the user is responsible for calling `tuppPoll` periodically. The actual processing of the event information is the same for both modes.
- `cbackIO`, `cbackVTPP`, `cbackRTOP` and `cbackRTTB` are used to pass the address of application functions that will be used by the DPR to inform the application code of pending events. If these fields are set as NULL, then any events that might cause the DPR to 'call back' the application will be processed during ISR processing but ignored by the DPR.

Table 3: Device Initialization Vector: sTUP_DIV

Field Name	Field Type	Field Description
`valid`	`UINT2`	Indicates that this profile is valid
`initMode`	`TUP_MODE`	Mode used for Initialization: `TUP_NORM`, `TUP_COMP` or `TUP_FRM`
`pinitData`	`UINT1*`	(pointer to) initialization data. Depending on the specified mode of initialization, this is in fact a pointer to `sTUP_INIT_DATA_NORM`, `sTUP_INIT_DATA_COMP` or `sTUP_INIT_DATA_FRM`
`pollISR`	`TUP_POLL`	Indicates the type of ISR / polling to do
`cbackIO`	`sTUP_CBACK`	Address for the callback function for IO Events
`cbackVTPP`	`sTUP_CBACK`	Address for the callback function for VTPP Events
`cbackRTOP`	`sTUP_CBACK`	Address for the callback function for RTOP Events
`cbackRTTB`	`sTUP_CBACK`	Address for the callback function for RTTB Events

Initialization Profile: INIT_PROF

Initialization Profile Top-Level Structure

Passed via the tuppAddInitProfile call, this structure contains all the information needed by the driver to initialize and activate a TUPP+622 device. This is in fact the same structure as sTUP_DIV.

Table 4: Initialization Profile: sTUP_INIT_PROF

Field Name	Field Type	Field Description
valid	UINT2	Indicates that this profile is valid
initMode	TUP_MODE	Mode used for Initialization: TUP_NORM, TUP_COMP or TUP_FRM
pinitData	UINT1*	(pointer to) initialization data. Depending on the specified mode of initialization, this is in fact a pointer to sTUP_INIT_DATA_NORM, sTUP_INIT_DATA_COMP or sTUP_INIT_DATA_FRM.
pollISR	TUP_POLL	Indicates the type of ISR / polling to do
cbackIO	sTUP_CBACK	Address for the callback function for IO Events
cbackVTPP	sTUP_CBACK	Address for the callback function for VTPP Events
cbackRTOP	sTUP_CBACK	Address for the callback function for RTOP Events
cbackRTTB	sTUP_CBACK	Address for the callback function for RTTB Events

Initialization Data in Normal Mode (TUP_NORM)

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

Table 5: Initialization Data: sTUP_INIT_DATA_NORM

Field Name	Field Type	Field Description
au4Mode[4]	UINT1	Selects between AU4 and AU3 mode of operation for the incoming and outgoing data stream

Field Name	Field Type	Field Description
`tug3Mapping[4][3]`	`UINT1`	Selects the type of mapping for the incoming and outgoing data stream. When non-zero, enables processing a TU3 or TUG2s that have been mapped into a TUG3. When zero, enables processing TUG2s that have been mapped into a VC3
`tu3Mapping[4][3]`	`UINT1`	Enables processing a single TU3 that have been mapped into a TUG3.
`tug2Mapping[4][3][7]`	`UINT1`	Selects between TU2 (VT6) , VT3, TU12 (VT2), and TU11 (VT1.5) mapping for each TUG2 (VTG) by specifying how many tributaries constitute each TUG2
`byPass[4][3]`	`UINT1`	When non-zero, the corresponding AU3 is bypassed by the TUPP+622, the corresponding processors are left in reset.

Initialization Data in Compatibility Mode (TUP_COMP)

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

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

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

Table 6: Initialization Data: sTUP_INIT_DATA_COMP

Field Name	Field Type	Field Description
`numBlocks`	`UINT2`	Number of provided blocks
`ppblk[]`	`UINT1*`	(pointer to) an array of pointer to a data block

Field Name	Field Type	Field Description
ppmask[]	UINT1*	(pointer to) an array of pointer to a mask
pblkSize[]	UINT2	(pointer to) an array of block size
pstartReg[]	UINT2	Array of register numbers

Initialization Data in FRM Mode (TUP_FRM)

In Flat Register Mode (FRM), for each of the hardware blocks (IO, VTPP, RTOP and RTTB), the user needs to fill a structure that holds a mapping of all the configuration bits for this hardware block. They are used to fully configure the TUPP+622 device. This structure is pointed to by `pinitData` inside the initialization profile. The reader is referred to the code for the definitions of the configuration blocks (`sTUP_CFG_XXX`).

Table 7: Initialization Data: sTUP_INIT_DATA_FRM

Field Name	Field Type	Field Description
au4Mode[4]	UINT1	Selects between AU4 and AU3 mode of operation for the incoming and outgoing data stream
tug3Mapping[4][3]	UINT1	Selects the type of mapping for the incoming and outgoing data stream. When non-zero, enables processing a TU3 or TUG2s that have been mapped into a TUG3. When zero, enables processing TUG2s that have been mapped into a VC3
tu3Mapping[4][3]	UINT1	Enables processing a single TU3 that have been mapped into a TUG3
tug2Mapping[4][3][7]	UINT1	Selects between TU2 (VT6) , VT3, TU12 (VT2), and TU11 (VT1.5) mapping for each TUG2 (VTG) by specifying how many tributaries constitute each TUG2
byPass[4][3]	UINT1	When non-zero, the corresponding AU3 is bypassed by the TUPP+622, the corresponding processors are left in reset.

Field Name	Field Type	Field Description
cfgIO[4]	sTUP_CFG_IO	Input/Output (IO) configuration block
cfgVTPP[4]	sTUP_CFG_VTPP	Tributary Payload Processor (VTPP) configuration block
cfgRTOP[4]	sTUP_CFG_RTOP	Tributary Path Overhead Processor (RTOP) configuration block
cfgRTTB[4]	sTUP_CFG_RTTB	Tributary Trace Buffer (RTTB) configuration block

ISR Enable/Disable Mask

Passed via the `tuppSetMask`, `tuppGetMask` and `tuppClearMask` calls, this structure contains all the information needed by the driver to enable and disable any of the interrupts in the TUPP+622.

Table 8: ISR Mask: sTUP_MASK

Field Name	Field Type	Field Description
ioIpe[4]	UINT1	Incoming parity error
lom[4][3]	UINT1	Loss of Multiframe
vtppMaster[4][3]	UINT1	VTPP master interrupt
rtopMaster[4][3]	UINT1	RTOP master interrupt
rttbMaster[4][3]	UINT1	RTTB master interrupt
vtppLop[4][3][7][4]	UINT1	Loss of pointer
vtppNje[4][3][7][4]	UINT1	Negative pointer justification event.
vtppPje[4][3][7][4]	UINT1	Positive pointer justification event.
vtppEse[4][3][7][4]	UINT1	Elastic store error
vtppAis[4][3][7][4]	UINT1	Path AIS
rtopPslu[4][3][7][4]	UINT1	Tributary path signal label unstable

Field Name	Field Type	Field Description
rtopPslm[4][3][7][4]	UINT1	Tributary path signal label mismatch
rtopCopsl[4][3][7][4]	UINT1	Change of tributary path signal label
rtopRfi[4][3][7][4]	UINT1	Remote failure indication
rtopRdi[4][3][7][4]	UINT1	Remote defect indication
rttbTim[4][3][7][4]	UINT1	Trail trace identifier mismatch
rttbTiu[4][3][7][4]	UINT1	Trail trace identifier unstable

Device and Alarm Status

This structure as well as its component structures is used by `tuppGetStatus` to retrieve all the status information for a given STM1.

Table 9: Alarm Status (sTUP_STATUS)

Field Name	Field Type	Field Description
statIO	sTUP_STATUS_IO	Alarm status of the input/output (IO)
statVTPP[3]	sTUP_STATUS_VTPP	Alarm status of the Section Overhead (VTPP)
statRTOP[3]	sTUP_STATUS_RTOP	Alarm status of the Received Tributary Overhead Processor (RTOP)
statRTTB[3]	sTUP_STATUS_RTTB	Alarm status of the Received Tributary Trace Buffer (RTTB)

Input / Output (IO) Status

Table 10: Input/Output Status (sTUP_STATUS_IO)

Field Name	Field Type	Field Description
otmfActiv	UINT1	Monitors for low to high transition on the OTMF input
gsclkfpActiv	UINT1	Monitors for low to high transition on the GSCLK_FP input

Field Name	Field Type	Field Description
idActiv	UINT1	Monitors for low to high transition on the input data bus (ID)
itmfActiv	UINT1	Monitors for low to high transition on the ITMF input
iplActiv	UINT1	Monitors for low to high transition on the IPL input
ic1j1Activ	UINT1	Monitors for low to high transition on the IC1J1 input
sclkActiv	UINT1	Monitors for low to high transition on the SCLK input
hsclkActiv	UINT1	Monitors for low to high transition on the HSCLK input
itv5Activ	UINT1	Monitors for low to high transition on the ITV5 input
itplActiv	UINT1	Monitors for low to high transition on the ITPL input
iaisActiv	UINT1	Monitors for low to high transition on the IAIS input

Tributary Payload Processor (VTPP) Status

Table 11: VTPP Status (sTUP_STATUS_VTPP)

Field Name	Field Type	Field Description
ss[7][4]	UINT1	Value of the size bits in the V1 byte of the corresponding tributary

Tributary Path Overhead Processor (RTOP) Status

Table 12: RTOP Status (sTUP_STATUS_RTOP)

Field Name	Field Type	Field Description
expPSL[7][4]	UINT1	Expected Path Signal Label for the corresponding tributary
accPSL[7][4]	UINT1	Accepted Path Signal Label for the corresponding tributary
rdi[7][4]	UINT1	Remote Defect Indication

Field Name	Field Type	Field Description
rfi[7][4]	UINT1	Remote Failure Indication
erdi[7][4]	UINT1	Enhanced Remote Defect Indication
pslm[7][4]	UINT1	Path Signal Label Mistmatch
pslu[7][4]	UINT1	Path Signal Label Unstable

Tributary Trace Buffer (RTTB) Status

Table 13: RTTB Status (sTUP_STATUS_RTTB)

Field Name	Field Type	Field Description
expTraceMsg[7][4]	UINT1[64]	Expected tributary trace message
capTraceMsg[7][4]	UINT1[64]	Captured tributary trace message
tim[7][4]	UINT1	Trace Identifier Mismatch
tiu[7][4]	UINT1	Trace Identifier Unstable

Statistic Counters: CNT

This structure is used by the statistics collection APIs to retrieve the device counts. The user can call `tuppGetCnt` to collect all the device counts for a given STM-1.

Table 14: Statistic Counters (sTUP_STAT_CNT)

Field Name	Field Type	Field Description
rtopBip[3][7][4]	UINT2	Number of block interleave parity errors (BIP-8 or BIP-2).
rtopRei[3][7][4]	UINT2	Number of remote error indication (REI).

Statistic Counter Configuration: CFG_CNT

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

Table 15: Counters Config (sTUP_CFG_CNT)

Field Name	Field Type	Field Description
`rtopBlkBip[4][3][7][4]`	`UINT1`	Controls the accumulation of block BIP-8/BIP-2 errors. When non-zero, one or more errors in the tributary BIP-8/BIP-2 byte results in a single error accumulated in the error counter. When zero, all errors are accumulated in the error counter.
`rtopBlkRei[4][3]`	`UINT1`	Controls the accumulation of REI's in the incoming TU3 stream on a block or bit basis. When non-zero, REI count codes in the range of 1 to 8 are accumulated on a block basis as a single REI event. All other codes are counted zero events. When zero, REI count codes in the range of 1 to 8 are accumulated on a bit basis as a up to 8 REI events. All other codes are counted zero events.

4.3 Structures in 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.

Table 16: Module Data Block: sTUP_MDB

Field Name	Field Type	Field Description
errModule	INT4	Global error Indicator for module calls
valid	UINT2	Indicates that this structure has been initialized
maxDevs	UINT2	Maximum number of devices supported
numDevs	UINT2	Number of devices currently registered
maxInitProfs	UINT2	Maximum number of initialization profiles
stateModule	TUP_MOD_STATE	Module state; can be one of the following TUP_MOD_START, TUP_MOD_IDLE or TUP_MOD_READY
pddb	sTUP_DDB *	(array of) Device Data Blocks (DDB) in context memory
pinitProfs	sTUP_INIT_PROF *	(array of) initialization profiles

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.

Table 17: Device Data Block: sTUP_DDB

Field Name	Field Type	Field Description
errDevice	INT4	Global error indicator for device calls
valid	UINT2	Indicates that this structure has been initialized
baseAddr	UINT1*	Base address of the device
usrCtxt	sTUP_USR_CTXT	Stores the user’s context for the device. It is passed as an input parameter when the driver invokes an application callback
profileNum	UINT2	Profile number used at initialization

Field Name	Field Type	Field Description
stateDevice	TUP_DEV_STATE	Device State; can be one of the following TUP+START, TUP_PRESENT, TUP_ACTIVE or TUP_INACTIVE
au4Mode[4]	UINT1	Selects between AU4 and AU3 mode of operation for the incoming and outgoing data stream
tug3Mapping[4][3]	UINT1	Selects the type of mapping for the incoming and outgoing data stream. When non-zero, enables processing a TU3 or TUG2s that have been mapped into a TUG3. When zero, enables processing TUG2s that have been mapped into a VC3
tu3Mapping[4][3]	UINT1	Enables processing a single TU3 that have been mapped into a TUG3.
tug2Mapping[4][3][7]	UINT1	Selects between TU2 (VT6) , VT3, TU12 (VT2), and TU11 (VT1.5) mapping for each TUG2 (VTG) by specifying how many tributaries constitute each TUG2
byPass[4][3]	UINT1	When non-zero, the corresponding AU3 is bypassed by the TUPP+622, the corresponding processors are left in reset.
cfgIO[4]	sTUP_CFG_IO	Input/Output (IO) configuration block
cfgVTPP[4]	sTUP_CFG_VTPP	Tributary Payload Processor (VTPP) configuration block
cfgRTOP[4]	sTUP_CFG_RTOP	Tributary Path Overhead Processor (RTOP) configuration block
cfgRTTB[4]	sTUP_CFG_RTTB	Tributary Trace Buffer (RTTB) configuration block
pollISR	TUP_POLL	Indicates the current type of ISR/polling
cbackIO	sTUP_CBACK	Address for the callback function for IO Events

Field Name	Field Type	Field Description
cbackVTPP	sTUP_CBACK	Address for the callback function for VTPP Events
cbackRTOP	sTUP_CBACK	Address for the callback function for RTOP Events
cbackRTTB	sTUP_CBACK	Address for the callback function for RTTB Events
mask	sTUP_MASK	Interrupt Enable Mask

4.4 Structures Passed Through RTOS Buffers

Interrupt-Service Vector: ISV

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

Table 18: Interrupt-Service Vector: sTUP_ISV

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

Deferred-Processing Vector: DPV

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

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

Table 19: Deferred-Processing Vector: sTUP_DPV

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

4.5 Global Variable

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

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

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

5 APPLICATION PROGRAMMING INTERFACE

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

5.1 Module Initialization

Opening the Driver Module: tuppModuleOpen

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

Prototypes `INT4 tuppModuleOpen(sTUP_MIV *pmiv)`

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

Outputs None

Returns Success = `TUP_SUCCESS`
Failure = <`TUPP+622 ERROR CODE`>

Valid States START

Side Effects Changes MODULE state to IDLE

Closing the Driver Module: tuppModuleClose

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

Prototype `INT4 tuppModuleClose(void)`

Inputs None

Outputs None

Returns Success = `TUP_SUCCESS`
Failure = `<TUPP+622 ERROR CODE>`

Valid States ALL STATES

Side Effects Changes MODULE state to START

5.2 Module Activation

Starting the Driver Module: tuppModuleStart

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

Prototype `INT4 tuppModuleStart(void)`

Inputs None

Outputs None

Returns Success = `TUP_SUCCESS`
Failure = <`TUPP+622 ERROR CODE`>

Valid States IDLE

Side Effects Changes MODULE state to READY

Stopping the Driver Module: tuppModuleStop

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

Prototype `INT4 tuppModuleStop(void)`

Inputs None

Outputs None

Returns Success = `TUP_SUCCESS`
Failure = < `TUPP+622 ERROR CODE`>

Valid States READY

Side Effects Changes MODULE state to IDLE

5.3 Initialization Profile Management

Creating an Initialization Profile: tuppAddInitProfile

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

Prototype `INT4 tuppAddInitProfile(sTUP_INIT_PROF *pProfile, UINT2 *pProfileNum)`

Inputs

`pProfile`	: (pointer to) initialization profile being added
`pProfileNum`	: (pointer to) profile number to be assigned by the driver

Outputs

`profileNum`	: profile number assigned by the driver

Returns Success = `TUP_SUCCESS`
Failure = <`TUPP+622 ERROR CODE`>

Valid States IDLE, READY

Side Effects None

Retrieving an Initialization Profile: tuppGetInitProfile

This function retrieves the contents of the initialization profile.

Prototype `INT4 tuppGetInitProfile(UINT2 profileNum, sTUP_INIT_PROF *pProfile)`

Inputs

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

Outputs

`pProfile`	: contents of the corresponding profile

Returns Success = `TUP_SUCCESS`
Failure = <`TUPP+622 ERROR CODE`>

Valid States IDLE, READY

Side Effects None

Deleting an Initialization Profile: tuppDeleteInitProfile

This function deletes an initialization profile given its profile number.

Prototype `INT4 tuppDeleteInitProfile(UINT2 profileNum)`

Inputs `profileNum` : initialization profile number

Outputs None

Returns Success = `TUP_SUCCESS`
Failure = <`TUPP+622 ERROR CODE`>

Valid States IDLE, READY

Side Effects None

5.4 Device Addition and Deletion

Adding a Device: tuppAdd

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

Prototype `sTUP_HNDL tuppAdd(void *usrCtxt, UINT1 *baseAddr, INT4 **perrDevice)`

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

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

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

Valid States READY

Side Effects Changes the DEVICE state to PRESENT

Deleting a Device: tuppDelete

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

Prototype `INT4 tuppDelete(sTUP_HNDL deviceHandle)`

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

Outputs None

Returns Success = `TUP_SUCCESS`
Failure = <`TUPP+622 ERROR CODE`>

Valid States PRESENT, ACTIVE, INACTIVE

Side Effects None

5.5 Device Initialization

Initializing a Device: tuppInit

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

Prototype `INT4 tuppInit(sTUP_HNDL deviceHandle, sTUP_DIV *pdiv, UINT2 profileNum)`

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

Outputs None

Returns Success = `TUP_SUCCESS`
Failure = <`TUPP+622 ERROR CODE`>

Valid States PRESENT

Side Effects Changes DEVICE state to INACTIVE

Updating the Configuration of a Device: tuppUpdate

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

Prototype `INT4 tuppUpdate(sTUP_HNDL deviceHandle, sTUP_DIV *pdiv, UINT2 profileNum)`

Inputs

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

Outputs None

Returns Success = `TUP_SUCCESS`
Failure = <`TUPP+622 ERROR CODE`>

Valid States PRESENT

Side Effects Changes DEVICE state to INACTIVE

Resetting a Device: tuppReset

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

Prototype `INT4 tuppReset(sTUP_HNDL deviceHandle)`

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

Outputs None

Returns Success = `TUP_SUCCESS`
Failure = <`TUPP+622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects Changes DEVICE state to PRESENT

5.6 Device Activation and De-Activation

Activating a Device: tuppActivate

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

Prototype `INT4 tuppActivate(sTUP_HNDL deviceHandle)`

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

Outputs None

Returns Success = `TUP_SUCCESS`
Failure = <`TUPP+622 ERROR CODE`>

Valid States INACTIVE

Side Effects Change the device state to ACTIVE

DeActivating a Device: tuppDeActivate

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

Prototype `INT4 tuppDeActivate(sTUP_HNDL deviceHandle)`

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

Outputs None

Returns Success = `TUP_SUCCESS`
Failure = <`TUPP+622 ERROR CODE`>

Valid States ACTIVE

Side Effects Changes the device state to INACTIVE

5.7 Device Reading and Writing

Reading from a Device Register: tuppRead

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

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

Prototype `UINT1 tuppRead(sTUP_HNDL deviceHandle, UINT2 regNum)`

Inputs
`deviceHandle` : device Handle (from `tuppAdd`)
`regNum` : register number

Outputs ERROR code written to the DDB

Returns Success = the register value
Failure = `0x00`

Valid States All Device States

Side Effects May affect registers that change after a read operation

Writing to a Device: tuppWrite

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

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

Prototype `UINT1 tuppWrite(sTUP_HNDL deviceHandle, UINT2 regNum, UINT1 value)`

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

Outputs ERROR code written to the DDB

Returns Success = previous value
Failure = `0x00`

Valid States All Device States

Side Effects May change the configuration of the Device

Reading a Block of Registers: tuppReadBlock

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

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

Prototype
```
void tuppReadBlock(sTUP_HNDL deviceHandle, UINT2
startRegNum, UINT2 size, UINT1 *pblock)
```

Inputs

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

Outputs ERROR code written to the DDB

`pblock`	: (pointer to) the block read

Returns None

Valid States ALL DEVICE STATES

Side Effects May affect registers that change after a read operation

Writing a Block of Registers: tuppWriteBlock

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

Note: Any error indication is written to the DDB

Prototype
```
void tuppWriteBlock(sTUP_HNDL deviceHandle, UINT2
```

```
startRegNum, UINT2 size, UINT1 *pblock, UINT1 *pmask)
```

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

Outputs ERROR code written to the DDB

Returns None

Valid States ALL DEVICE STATES

Side Effects May change the configuration of the Device

5.8 Input/Output

Configuring Auto-Responses: tuppAutoResponse

This function configures the per STM1 auto-response behavior of the TUPP+622. The device is optionally configured to automatically insert AIS, RDI or ARDI upon detection of one or more of the following alarms LOM, LOP, AIS, UNEQ, PSLM, PSLU, TIM and TIU.

Prototype

```
INT4 tuppAutoResponse(sTUP_HNDL deviceHandle, UINT2
stm1, UINT1 ais, UINT1 rdi, UINT1 ardi)
```

Inputs	`deviceHandle`	: device Handle (from `tuppAdd`)
	`stm1`	: STM-1 (STS-3) index
	`ais`	: mask to write to STP Tributary Alarm auxiliary AIS Control Register (register 010H)
	`rdi`	: mask to write to STP Tributary Remote defect Indication Control Register:(register 011H)
	`ardi`	: mask to write to STP Tributary Auxiliary Remote Defect Indication Control Register: (register 012H)

Outputs None

Returns Success = `TUP_SUCCESS`
Failure = <`TUPP+622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

5.9 Tributary Payload Processor

Configuring Position of the J1 Byte: tuppVTPPConfigJ1

This function configures the position of the J1 byte for a given STM-1 (STS-3). For example, setting `posJ1` to a value of zero will force J1 to immediately follow H3, while a value of 522 will force J1 to immediately follow J0/Z0. `posJ1` can be set to any value between 0 and 782.

Prototype
```
INT4 tuppVTPPConfigJ1(sTUP_HNDL deviceHandle, UINT2
stm1, UINT2 posJ1)
```

Inputs

`deviceHandle`	: device Handle (from `tuppAdd`)
`stm1`	: STM-1 (STS-3) index
`posJ1`	: J1 position

Outputs None

Returns Success = `TUP_SUCCESS`
Failure = <`TUPP+622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Inserting all 0s in H4: tuppVTPPSquelchH4

This function enables insertion of an all-zeros byte in the H4 position of the outgoing payload.

Prototype
```
INT4 tuppVTPPSquelchH4(sTUP_HNDL deviceHandle, UINT2
stm1, UINT2 au3, UINT2 enable)
```

Inputs

`deviceHandle`	: device Handle (from `tuppAdd`)
`stm1`	: STM-1 (STS-3) index
`au3`	: AU-3 (STS-1) index
`enable`	: enable zeroing of H4 byte

Outputs None

Returns Success = `TUP_SUCCESS`
Failure = `<TUPP+622 ERROR CODE>`

Valid States ACTIVE, INACTIVE

Side Effects None

Forcing Path AIS: tuppVTPPForcePAIS

This function enables insertion of path AIS in the outgoing payload.

Prototype

```
INT4 tuppVTPPForcePAIS(sTUP_HNDL deviceHandle, UINT2
stm1, UINT2 au3, UINT2 tug2, UINT2 tu, UINT2 enable)
```

Inputs

`deviceHandle`	: device Handle (from `tuppAdd`)
`stm1`	: STM-1 (STS-3) index
`au3`	: AU-3 (STS-1) index
`tug2`	: TUG2 (VTG) index
`tu`	: TU (VT) index
`enable`	: enable forcing insertion of path AIS

Outputs None

Returns Success = `TUP_SUCCESS`
Failure = `<TUPP+622 ERROR CODE>`

Valid States ACTIVE, INACTIVE

Side Effects None

Forcing Path IDLE: tuppVTPPForceIDLE

This function enables insertion of path IDLE in the outgoing payload.

Prototype

```
INT4 tuppVTPPForceIDLE(sTUP_HNDL deviceHandle, UINT2
stm1, UINT2 au3, UINT2 tug2, UINT2 tu, UINT2 enable)
```

Inputs

`deviceHandle`	: device Handle (from `tuppAdd`)
`stm1`	: STM-1 (STS-3) index
`au3`	: AU-3 (STS-1) index
`tug2`	: TUG2 (VTG) index
`tu`	: TU (VT) index
`enable`	: enable forcing insertion of path IDLE

Outputs None

Returns Success = `TUP_SUCCESS`
Failure = `<TUPP+622 ERROR CODE>`

Valid States ACTIVE, INACTIVE

Side Effects None

Forcing Loss of Pointer: tuppVTPPDiagLOP

This function enables inversion of the new data flag (NDF) field of the outgoing payload pointer to cause downstream pointer processing elements to enter a loss of pointer state.

Prototype
```
INT4 tuppVTPPDiagLOP(sTUP_HNDL deviceHandle, UINT2
stm1, UINT2 au3, UINT2 tug2, UINT2 tu, UINT2 enable)
```

Inputs

`deviceHandle`	: device Handle (from `tuppAdd`)
`stm1`	: STM-1 (STS-3) index
`au3`	: AU-3 (STS-1) index
`tug2`	: TUG2 (VTG) index
`tu`	: TU (VT) index
`enable`	: enable

Outputs None

Returns Success = `TUP_SUCCESS`
Failure = `<TUPP+622 ERROR CODE>`

Valid States ACTIVE, INACTIVE

Side Effects None

5.10 Tributary Path Overhead Processors

Forcing PDIV Output High: tuppRTOPForcePDIVHigh

This function forces the state of the PDIV output. When enable flag is set, the PDIV output is set high independently of the tributary’s defect status.

Prototype
```
INT4 tuppRTOPForcePDIVHigh(sTUP_HNDL deviceHandle,
UINT2 stm1, UINT2 au3, UINT2 tug2, UINT2 tu, UINT2
```

enable)

Inputs

	`deviceHandle`	: device Handle (from `tuppAdd`)
	`stm1`	: STM-1 (STS-3) index
	`au3`	: AU-3 (STS-1) index
	`tug2`	: TUG2 (VTG) index
	`tu`	: TU (VT) index
	`enable`	: enable forcing PDIV high

Outputs None

Returns Success = `TUP_SUCCESS`
Failure = <`TUPP+622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Getting/Setting Path Signal Label: tuppRTOPPathSigLabel

This function gets/sets the path signal label in the Tributary Path Overhead Processor. It is the user's responsibility to make sure that the path signal label pointer points to an area of memory large enough to hold all the data returned.

Prototype

```
INT4 tuppRTOPPathSigLabel(sTUP_HNDL deviceHandle,
UINT2 stm1, UINT2 au3, UINT2 tug2, UINT2 tu, UINT2
rw, UINT2 type, UINT1 *psl)
```

Inputs

	`deviceHandle`	: device Handle (from `tuppAdd`)
	`stm1`	: STM-1 (STS-3) index
	`au3`	: AU-3 (STS-1) index
	`tug2`	: TUG2 (VTG) index
	tu	: TU (VT) index
	`rw`	: read/write flag, write if zero
	`type`	: type of access: `0`: accepted path signal label `1`: expected path signal label
	`psl`	: (pointer to) path signal label

Outputs `psl` : updated path signal label

Returns Success = `TUP_SUCCESS`
Failure = <`TUPP+622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Configuring Tributary AIS Auto-Responses: tuppAutoResponseTribAIS

This function configures the TUPP+622 to automatically insert AIS on a given TU upon detection of one or more of the following alarms UNEQ, PSLM, PSLU, TIM and TIU, as configured by `tuppAutoResponse`.

Prototype

```
INT4 tuppAutoResponseTribAIS(sTUP_HNDL deviceHandle,
UINT2 stm1, UINT2 au3, UINT2 tug2, UINT2 tu, UINT2
enable)
```

Inputs

`deviceHandle`	: device Handle (from `tuppAdd`)
`stm1`	: STM-1 (STS-3) index
`au3`	: AU-3 (STS-1) index
`tug2`	: TUG2 (VTG) index
`tu`	: TU (VT) index
`enable`	: flag to enable/disable AIS insertion

Outputs None

Returns Success = `TUP_SUCCESS`
Failure = <`TUPP+622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Configuring In-Band Error Reporting: tuppAutoResponseTribIBER

This function enables/disables in-band error reporting of the TUPP+622 for a given TU.

Prototype

```
INT4 tuppAutoResponseTribIBER(sTUP_HNDL deviceHandle,
UINT2 stm1, UINT2 au3, UINT2 tug2, UINT2 tu, UINT2
enable)
```

Inputs

`deviceHandle`	: device Handle (from `tuppAdd`)
`stm1`	: STM-1 (STS-3) index
`au3`	: AU-3 (STS-1) index
`tug2`	: TUG2 (VTG) index
`tu`	: TU (VT) index
`enable`	: flag to enable/disable in band error reporting

Outputs None

Returns Success = `TUP_SUCCESS`
Failure = <`TUPP+622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

5.11 Tributary Trace Buffer

Getting/Setting Tributary Trace Messages: tuppTributaryTraceMsg

This function Gets or Sets the current tributary trace message (J2) in the Tributary Trace Buffer. It is the user's responsibility to make sure that the message pointer points to an area of memory large enough to hold all the data returned.

Prototype

```
INT4 tuppTributaryTraceMsg(sTUP_HNDL deviceHandle,
UINT2 stm1, UINT2 au3, UINT2 tug2, UINT2 tu, UINT2
rw, UINT2 type, UINT1 *pJ2)
```

Inputs

`deviceHandle`	: device Handle (from `tuppAdd`)
`stm1`	: STM-1 (STS-3) index
`au3`	: AU-3 (STS-1) index
`tug2`	: TUG2 (VTG) index
`tu`	: TU (VT) index
`rw`	: read/write flag, write if zero
`type`	: type of access: 0: captured tributary trace 1: expected tributary trace
`pJ2`	: (pointer to) the tributary trace message

Outputs `pJ2` : updated tributary trace message

Returns Success = `TUP_SUCCESS`
Failure = <`TUPP+622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

5.12 Interrupt Service Functions

Getting the Interrupt Mask: tuppGetMask

This function returns the contents of the interrupt mask registers of the TUPP+622 device.

Prototype	`INT4 tuppGetMask(sTUP_HNDL deviceHandle, sTUP_MASK *pmask)`	
Inputs	`deviceHandle`	: device Handle (from `tuppAdd`)
	`pmask`	: (pointer to) mask structure
Outputs	ERROR code written to the DDB	
Returns	Success = `TUP_SUCCESS` Failure = <`TUPP+622 ERROR CODE`>	
Valid States	ACTIVE, INACTIVE	
Side Effects	None	

Setting the Interrupt Mask: tuppSetMask

This function sets the contents of the interrupt mask registers of the TUPP+622 device.

Prototype	`INT4 tuppSetMask(sTUP_HNDL deviceHandle, sTUP_MASK *pmask)`	
Inputs	`deviceHandle`	: device Handle (from `tuppAdd`)
	`pmask`	: (pointer to) mask structure
Outputs	ERROR code written to the DDB	
Returns	Success = `TUP_SUCCESS` Failure = <`TUPP+622 ERROR CODE`>	
Valid States	ACTIVE, INACTIVE	
Side Effects	May change the operation of the ISR / DPR	

Clearing the Interrupt Mask: tuppClearMask

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

Prototype `INT4 tuppClearMask(sTUP_HNDL deviceHandle, sTUP_MASK *pmask)`

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

Outputs ERROR code written to the DDB

Returns
Success = `TUP_SUCCESS`
Failure = <`TUPP+622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects May change the operation of the ISR / DPR

Polling Interrupt Status Registers: tuppPoll

This function commands the driver to poll the interrupt registers in the device. The call will fail unless the device is initialized into polling mode. An additional parameter is available that starts automatic polling on one second boundaries. 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 tuppPoll(sTUP_HNDL deviceHandle)`

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

Outputs None

Returns
Success = `TUP_SUCCESS`
Failure = <`TUPP+622 ERROR CODE`>

Valid States ACTIVE

Side Effects None

Interrupt-Service Routine: tuppISR

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

Prototype `void * tuppISR(sTUP_HNDL deviceHandle)`

Inputs	`deviceHandle`	: device Handle (from `tuppAdd`)
Outputs	None	
Returns	(pointer to) ISV buffer if any valid interrupt condition was found	
Valid States	ACTIVE	
Side Effects	None	

Deferred-Processing Routine: tuppDPR

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

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

5.13 Alarm, Status and Statistics Functions

Configuring Statistical Counts: tuppCfgStats

This function configures the behavior of the device counts.

Prototype	`INT4 tuppCfgStats(sTUP_HNDL deviceHandle, sTUP_CFG_CNT cfgCnt)`	
Inputs	`deviceHandle` `cfgCnt`	: device Handle (from `tuppAdd`) : counters configuration block
Outputs	None	

Returns Success = `TUP_SUCCESS`
Failure = <`TUPP+622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Getting the Current Status: tuppGetStatus

This function retrieves all the device and alarm status for a given STM-1.

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

Prototype

```
INT4 tuppGetStatus(sTUP_HNDL deviceHandle, UINT2
stm1, sTUP_STATUS *palm)
```

Inputs

`deviceHandle`	: device Handle (from `tuppAdd`)
`stml`	: STM-1 (STS-3) index
`palm`	: (pointer to) allocated memory

Outputs

`palm`	: updated status structure for this STM-1

Returns Success = `TUP_SUCCESS`
Failure = <`TUPP+622 ERROR CODE`>

Valid States ACTIVE, INACTIVE

Side Effects None

Reading the Device Counters: tuppGetCnt

This function retrieves all the device counts for a given STM-1.

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

Prototype

```
INT4 tuppGetCnt(sTUP_HNDL deviceHandle, UINT2 stm1,
sTUP_STAT_CNT *pcnt)
```

Inputs

`deviceHandle`	: device Handle (from `tuppAdd`)
`stml`	: STM-1 (STS-3) index
`pcnt`	: (pointer to) allocated memory

Outputs	pcnt	: updated count structure for this STM-1
Returns	Success = `TUP_SUCCESS` Failure = <`TUPP+622 ERROR CODE`>	
Valid States	ACTIVE, INACTIVE	
Side Effects	None	

5.14 Device Diagnostics

Verifying Register Access: tuppTestReg

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

Prototype	`INT4 tuppTestReg(sTUP_HNDL deviceHandle)`	
Inputs	`deviceHandle`	: device Handle (from `tuppAdd`)
Outputs	None	
Returns	Success = `TUP_SUCCESS` Failure = <`TUPP+622 ERROR CODE`>	
Valid States	PRESENT, INACTIVE	
Side Effects	None	

5.15 Callback Functions

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

IO Section Callbacks: cbackTuppIO

This 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 cbackTuppIO(sTUP_USR_CTXT usrCtxt, sTUP_DPV *pdpv)`	
Inputs	`usrCtxt`	: user context (from `tuppAdd`)
	`pdpv`	: (pointer to) formatted event buffer
Outputs	None	
Returns	None	
Valid States	ACTIVE	
Side Effects	None	

VTPP Section Callbacks: cbackTuppVTPP

This function is used by the DPR to report significant VTPP 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 cbackTuppVTPP(sTUP_USR_CTXT usrCtxt, sTUP_DPV *pdpv)`	
Inputs	`usrCtxt`	: user context (from `tuppAdd`)
	`pdpv`	: (pointer to) formatted event buffer
Outputs	None	
Returns	None	
Valid States	ACTIVE	
Side Effects	None	

RTOP Section Callbacks: cbackTuppRTOP

This function is used by the DPR to report significant RTOP 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 cbackTuppRTOP(sTUP_USR_CTXT usrCtxt, sTUP_DPV *pdpv)`	
Inputs	`usrCtxt`	: user context (from `tuppAdd`)
	`pdpv`	: (pointer to) formatted event buffer
Outputs	None	
Returns	None	
Valid States	ACTIVE	
Side Effects	None	

RTTB Section Callbacks: cbackTuppRTTB

This function is used by the DPR to report significant RTTB 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 cbackTuppRTTB(sTUP_USR_CTXT usrCtxt, sTUP_DPV *pdpv)`	
Inputs	`usrCtxt`	: user context (from `tuppAdd`)
	`pdpv`	: (pointer to) formatted event buffer
Outputs	None	
Returns	None	
Valid States	ACTIVE	
Side Effects	None	

6 HARDWARE INTERFACE

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

6.1 Device I/O

Reading Registers: sysTuppRead

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

Prototype	`UINT1 sysTuppRead(addr)`
Inputs	`addr` : register location to be read
Outputs	None
Returns	value read from the addressed register location
Format	`#define sysTuppRead(addr)`

Writing Values: sysTuppWrite

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

Prototype	`void sysTuppWrite(addr, value)`
Inputs	`addr` : register location to be read
Outputs	None
Returns	value read from the addressed register location
Format	`#define sysTuppWrite(addr, value)`

6.2 Interrupt Servicing

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

During execution of the API function `tuppModuleStart` / `tuppModuleStop` the driver informs the application that it is time to install / uninstall this shell via `sysTuppISRHandlerInstall` / `sysTuppISRHandlerRemove`, that needs to be supplied by the user.

Note: A device can be initialized with ISR disabled. In that mode, the user should periodically invoke a provided 'polling' routine (`tuppPoll`) that in turn calls `tuppISR`.

Installing the ISR Handler: sysTuppISRHandlerInstall

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

Prototype	`void sysTuppISRHandlerInstall(void *func)`	
Inputs	`func`	: (pointer to) the function `tuppISR`
Outputs	None	
Returns	None	
Valid States	None	
Format	`#define sysTuppISRHandlerInstall(func)`	

ISR Handler: sysTuppISRHandler

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

Prototype	`void sysTuppISRHandler(void)`
Inputs	None
Outputs	None

Returns None

Format `#define sysTuppISRHandler()`

Removing Handlers: sysTuppISRHandlerRemove

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

Prototype `void sysTuppISRHandlerRemove(void)`

Inputs None

Outputs None

Returns None

Format `#define sysTuppISRHandlerRemove()`

DPR Task: sysTuppDPRTask

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

Prototype `void sysTuppDPRTask(void)`

Inputs None

Outputs None

Returns None

Format `#define sysTuppDPRTask()`

7 RTOS INTERFACE

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

7.1 Memory Allocation / De-Allocation

Allocating Memory: sysTuppMemAlloc

This function allocates specified number of bytes of memory.

Prototype `UINT1 *sysTuppMemAlloc(UINT4 numBytes)`

Inputs `numBytes` : number of bytes to be allocated

Outputs None

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

Format `#define sysTuppMemAlloc(numBytes)`

Freeing Memory: sysTuppMemFree

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

Prototype `void sysTuppMemFree(UINT1 *pfirstByte)`

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

Outputs None

Returns None

Format `#define sysTuppMemFree(pfirstByte)`

7.2 Buffer Management

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

Starting Buffer Management: sysTuppBufferStart

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

Prototype `INT4 sysTuppBufferStart(void)`

Inputs None

Outputs None

Returns Success = `0`
Failure = any other value

Format `#define sysTuppBufferStart()`

Getting DPV Buffers: sysTuppDPVBufferGet

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

Prototype `sTUP_DPV *sysTuppDPVBufferGet(void)`

Inputs None

Outputs None

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

Format `#define sysTuppDPVBufferGet()`

Getting ISV Buffers: sysTuppISVBufferGet

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

Prototype `sTUP_ISV *sysTuppISVBufferGet(void)`

Inputs None

Outputs None

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

Format `#define sysTuppISVBufferGet()`

Returning DPV Buffers: sysTuppDPVBufferRtn

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

Prototype `void sysTuppDPVBufferRtn(sTUP_DPV *pdpv)`

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

Outputs None

Returns None

Format `#define sysTuppDPVBufferRtn(pdpv)`

Returning ISV Buffers: sysTuppISVBufferRtn

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

Prototype `void sysTuppISVBufferRtn(sTUP_ISV *pisv)`

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

Outputs None

Returns None

Format `#define sysTuppISVBufferRtn(pisv)`

Stopping Buffer Management: sysTuppBufferStop

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

Prototype `void sysTuppBufferStop(void)`

Inputs None

Outputs None

Returns None

Format `#define sysTuppBufferStop()`

7.3 Preemption

Disabling Preemption: sysTuppPreemptDisable

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

Prototype `INT4 sysTuppPreemptDisable(void)`

Inputs None

Outputs None

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

Format `#define sysTuppPreemptDisable()`

Re-Enabling Preemption: sysTuppPreemptEnable

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

Prototype `void sysTuppPreemptEnable(INT4 key)`

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

Outputs None

Returns None

Format `#define sysTuppPreemptEnable(key)`

7.4 Timers

Suspending a Task Execution: sysTuppTimerSleep

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

Prototype `void sysTuppTimerSleep(UINT4 msec)`

Inputs `msec` : sleep time in milliseconds

Outputs None

Returns None

Format `#define sysTuppTimerSleep(msec)`

8 PORTING DRIVERS

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

8.1 Driver Source Files

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

Directory	File	Description
`src`	`tup_api1.c`	All API functions that take care of module, device and profile management
	`tup_api2.c`	All TUPP+622 specific API functions.
	`tup_hw.c`	Hardware interface functions
	`tup_isr.c`	Internal functions that deal with interrupt servicing
	`tup_prof.c`	Internal functions that deal with profiles
	`tup_rtos.c`	RTOS interface functions
	`tup_stat.c`	Internal functions that deal with statistics
	`tup_util.c`	All the remaining internal functions
`inc`	`tup_api.h`	All API headers
	`tup_defs.h`	Driver macros, constants and definitions (such as register mapping and bit masks)
	`tup_err.h`	TUPP+622 error codes
	`tup_fns.h`	Prototype of non-API functions
	`tup_hw.h`	HW interface macros and prototype
	`tup_rtos.h`	RTOS interface macros and prototypes

Directory	File	Description
inc	tup_strs.h	Driver structures
	tup_typs.h	Types definitions
example	tup_app.c	Sample driver callback functions
	tup_app.h	Prototypes, macros and structures used inside the example code
	tup_debug.c	Functions to implement a debug diagnostic task
	tup_debug.h	Prototypes and structures used inside the debug task code

8.2 Driver Porting Procedures

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

To port the TUPP+622 driver to your platform:

Step 1: Port the driver's RTOS interface (page 75):

Step 2: Port the driver's hardware interface (page 76):

Step 3: Port the driver's application-specific elements (page 77):

Step 4: Build the driver (page 77).

Porting Assumptions

The following porting assumptions have been made:

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

Step 1: Porting the RTOS interface

The RTOS interface functions and macros consist of code that is RTOS dependent and needs to be modified as per your RTOS' characteristics.

To port the driver's OS extensions:

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

Service Type	Macro Name	Description
Memory	`sysTuppMemAlloc`	Allocates a memory block
	`sysTuppMemFree`	Frees a memory block
	`sysTuppMemCpy`	Copies the contents of one memory block to another
	`sysTuppMemSet`	Fills a memory block with a specified value
Timer	`sysTuppTimerSleep`	Delays the task execution for a given number of milliseconds
Pre-emption Lock/Unlock	`sysTuppPreemptDisable`	Disables pre-emption of the currently executing task by any other task or interrupt
	`sysTuppPreemptEnable`	Re-enables pre-emption of a task by other tasks and/or interrupts

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

Service Type	Macro Name	Description
Buffer	`sysTuppBufferStart`	Starts buffer management
	`sysTuppBufferStop`	Stops buffer management
	`sysTuppISVBufferGet`	Gets an ISV buffer from the ISV buffer queue
	`sysTuppISVBufferRtn`	Returns an ISV buffer to the ISV buffer queue
	`sysTuppDPVBufferGet`	Gets a DPV buffer from the DPV buffer queue
	`sysTuppDPVBufferRtn`	Returns a DPV buffer to the DPV buffer queue

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

Task Constant	Description	Default
TUP_DPR_TASK_PRIORITY	Deferred Task (DPR) task priority	85
TUP_DPR_TASK_STACK_SZ	DPR task stack size, in bytes	8192
TUP_MAX_ISV_BUF	The queue message depth of the queue used for pass interrupt context between the ISR task and DPR task	50
TUP_MAX_DPV_BUF	The queue message depth of the queue used for pass interrupt context between the ISR task and DPR task	950

Step 2: Porting the Hardware Interface

This section describes how to modify the TUPP+622 driver for your hardware platform.

To port the driver to your hardware platform:

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

Service Type	Function Name	Description
Register Access	sysTuppRead	Reads a device register given its real address in memory
	sysTuppWrite	Writes to a device register given its real address in memory
Interrupt	sysTuppISRHandlerInstall	Installs the interrupt handler for the OS
	sysTuppISRHandlerRemove	Removes the interrupt handler from the OS
	sysTuppISRHandler	Interrupt handler for the TUPP+622 device
	sysTuppDPRTask	Task that calls the TUPP+622 DPR

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

Device Constant	Description	Default
`TUP_MAX_DELAY`	Delay between two consecutive polls of a busy bit	100us
`TUP_MAX_POLL`	Maximum number of times a busy bit will be polled before the operation times out	100

Step 3: Porting the Application-Specific Elements

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

To port the driver's application-specific elements:

1. Modify the base value of `TUP_ERR_BASE` (default = -300) in `tup_err.h`.
2. Define the following constants for your OS-specific services in `tup_rtos.h`:

Task Constant	Description	Default
`TUP_MAX_DEVS`	The maximum number of TUPP+622 devices that can be supported by the driver	24
`TUP_MAX_INIT_PROFS`	The maximum number of initialization profiles that can be added to the driver	5

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

Step 4: Building the Driver

This section describes how to build the TUPP+622 driver.

To build the driver:

1. Modify the `Makefile` to reflect the absolute path of your code, your compiler and compiler options.

2. Choose from among the different compile options supported by the driver as per your requirements.
3. Compile the source files and build the TUPP+622 API driver library using your make utility.
4. Link the TUPP+622 API driver library to your application code.

APPENDIX A: DRIVER RETURN CODES

Table 20 describes the driver's return codes.

Table 20: Return Codes

Return Type	Description
TUP_ERR_MEM_ALLOC	Memory allocation failure
TUP_ERR_INVALID_ARG	Invalid argument
TUP_ERR_INVALID_MODULE_STATE	Invalid module state
TUP_ERR_INVALID_MIV	Invalid Module Initialization Vector
TUP_ERR_PROFILES_FULL	Maximum number of profiles already added
TUP_ERR_INVALID_PROFILE	Invalid profile
TUP_ERR_INVALID_PROFILE_MODE	Invalid profile mode selected
TUP_ERR_INVALID_PROFILE_NUM	Invalid profile number
TUP_ERR_INVALID_DEVICE_STATE	Invalid device state
TUP_ERR_DEVS_FULL	Maximum number of devices already added
TUP_ERR_DEV_ALREADY_ADDED	Device already added
TUP_ERR_INVALID_DEV	Invalid device handle
TUP_ERR_INVALID_DIV	Invalid Device Initialization Vector
TUP_ERR_INT_INSTALL	Error while installing interrupts
TUP_ERR_INVALID_MODE	Invalid ISR/polling mode
TUP_ERR_INVALID_REG	Invalid register number
TUP_ERR_POLL_TIMEOUT	Time-out while polling

APPENDIX B: CODING CONVENTIONS

This section describes the coding conventions used in the implementation of the TUPP+622 driver software.

Variable Type Definitions

Table 21: Variable Type Definitions

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

Naming Conventions

Table 22 presents a summary of the naming conventions followed by the TUPP+622 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 name or abbreviation appears in prefix.

Table 22: Naming Conventions

Type	Case	Naming convention	Examples
Macros	Uppercase	Prefix with “m” and device abbreviation	`mTUP_IO_OFFSET`
Constants	Uppercase	Prefix with device abbreviation	`TUP_REG_OFFSET_NEXT_STM1`

Type	Case	Naming convention	Examples
Structures	Hungarian Notation	Prefix with "s" and device abbreviation	sTUP_DDB
API Functions	Hungarian Notation	Prefix with device name	tuppAdd()
Porting Functions	Hungarian Notation	Prefix with "sys" and device name	sysTuppRead()
Other Functions	Hungarian Notation		utilTuppResetDev()
Variables	Hungarian Notation		maxDevs
Pointers to variables	Hungarian Notation	Prefix variable name with "p"	pmaxDevs
Global variables	Hungarian Notation	Prefix with device name	tuppMdb

Macros

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

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

Constants

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

- Constant names can be uppercase.
- Words can be separated by an underscore.
- The device abbreviation can appear as a prefix.
- Example: TUP_REG_OFFSET_NEXT_STM1 is a valid name for a constant.

Structures

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

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

Functions

API Functions

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

Porting Functions

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

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

Other Functions

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

Variables

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

- Note: Both `pprevBuf` and `pPrevBuf` are accepted names for a pointer to the `prevBuf` variable, and that both `pmatrix` and `ppmatrix` are accepted names for a double pointer to the variable matrix.

File Organization

Table 23 presents a summary of the file naming conventions. All file names must start with the device abbreviation, followed by an underscore and the actual file name. File names should convey their purpose with a minimum amount of characters. If a file size is getting too big one might separate it into two or more files, providing that a number is added at the end of the file name (e.g. `tup_api1.c` or `tup_api2.c`).

There are 4 different types of files:

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

Table 23: File Naming Conventions

File Type	File Name
API	`tup_api1.c, tup_api.h`
Hardware Dependent	`tup_hw.c, tup_hw.h`
RTOS Dependent	`tup_rtos.c, tup_rtos.h`
Other	`tup_isr.c, tup_defs.h`

API Files

- The name of the API files starts with the device abbreviation followed by an underscore and ‘`api`’. For more than one API file, a number is appended to the file name.
- Examples: `tup_api1.c` is the only valid name for the file that contains the first part of the API functions; `tup_api.h` is the only valid name for the file that contains all of the API functions headers.

Hardware Dependent Files

- The name of the hardware dependent files starts with the device abbreviation followed by an underscore and ‘`hw`’. For more than one hardware dependent file, a number is appended to the file name.
- Examples: `tup_hw.c` is the only valid name for the file that contains all of the hardware dependent functions; `tup_hw.h` is the only valid name for the file that contains all of the hardware dependent functions headers.

RTOS Dependent Files

- The name of the RTOS dependent files starts with the device abbreviation followed by an underscore and ‘`rtos`’. For more than one RTOS dependent file, a number is appended to the file name.
- Examples: `tup_rtos.c` is the only valid name for the file that contains all of the RTOS dependent functions; `tup_rtos.h` is the only valid name for the file that contains all of the RTOS dependent functions headers.

Other Driver Files

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

ACRONYMS

API: Application Programming Interface

DDB: Device Data Block

DIV: Device Initialization Vector

DPR: Deferred-Processing Routine

DPV: Deferred-Processing (routine) Vector

FIFO: First In, First Out

IO: Input/Output

ISR: Interrupt-Service Routine

ISV: Interrupt-Service (routine) Vector

MDB: Module Data Block

MIV: Module Initialization Vector

RTOP: Tributary Path Overhead Processor

RTOS: Real-Time Operating System

RTTB: Tributary Trace Buffer

VTPP: Tributary Payload Processor

LIST OF TERMS

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

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

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

DPR (Deferred-Processing Routine): This function is installed as a task, at a user configurable priority, that serves as the next logical step in Interrupt processing. Data that was collected by the ISR is analyzed and then calls are made into the Application 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 TUPP+622 Integrated Circuit. There can be many devices, all served by this one driver module

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

MODULE: All of the code that is part of this driver, there is only one instance of this module connected to one or more TUPP+622 chips.

- MIV (Module Initialization Vector): Structure passed from the API to the module during initialization, it contains parameters that identify the specific characteristics of the driver module being initialized.
- MDB (Module Data Block): Structure that holds the Configuration Data for this module.

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

INDEX