PM7324

S/UNI-ATLAS

DRIVER MANUAL

DOCUMENT ISSUE 3

ISSUED: JANUARY, 2002

Document ID 2000949, Issue 3

ABOUT THIS MANUAL AND S/UNI-ATLAS PM7324 (ATLAS)

This manual describes the S/UNI-ATLAS PM7324 (ATLAS) device driver. It describes the driver's functions, data structures, and architecture. This manual focuses on the driver's interfaces to your application, real-time operating system, and to the device. It also describes in general terms how to modify and port the driver to your software and hardware platform.

Audience

This manual was written for people who need to:

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

References

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

Table 1: Related Documents

Document Name	Document Number
S/UNI- ATLAS Long Form Data Sheet.	PMC-1971154
S/UNI-ATLAS Long Form Data Sheet Errata.	PMC-1981505

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	June 2000	Document created
Issue 2	August 2000	Valid states in atlasReadReg and atlasWriteReg have been changed from All states to ATLS_PRESENT
Issue 3	January 2002	Added atlasAddEgressDummyConnection API.

LEGAL INFORMATION

Copyright

PMC-2000949 (R3)

Disclaimer

None of the information contained in this document constitutes an express or implied warranty by PMC-Sierra, Inc. as to the sufficiency, fitness or suitability for a particular purpose of any such information or the fitness, or suitability for a particular purpose, merchantability, performance, compatibility with other parts or systems, of any of the products of PMC-Sierra, Inc., or any portion thereof, referred to in this document. PMC-Sierra, Inc. expressly disclaims all representations and warranties of any kind regarding the contents or use of the information, including, but not limited to, express and implied warranties of accuracy, completeness, merchantability, fitness for a particular use, or non-infringement.

In no event will PMC-Sierra, Inc. be liable for any direct, indirect, special, incidental or consequential damages, including, but not limited to, lost profits, lost business or lost data resulting from any use of or reliance upon the information, whether or not PMC-Sierra, Inc. has been advised of the possibility of such damage.

Patents

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

Canadian Patent No. 2,209,887, Canadian Patent No. 2,164,546, Canadian Patent No. 2,167,757, US Patent No. 5,668,797, US Patent No. 5,815,737, US Patent No. 6,108,303,

US Patent No. 6,128,766, UK Patent No. 2,301,913

Relevant patent applications and other patents may also exist.

CONTACTING PMC-SIERRA

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

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

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

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

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

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

For your convenience, Section 8of this manual provides a brief guide for porting the device ATLAS driver to your hardware and RTOS platform. In addition, an extensive Appendix (page 136) and Index (page 142) provides you with useful reference information.

2 DRIVER FUNCTIONS AND FEATURES

This section describes the main functions and features supported by the ATLAS driver.

Table 2: Driver Functions and Features

Function	Description
Device Initialization (page 57)	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 ATLAS device.
Read / Write Device Registers (page 58)	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.
Add / Delete Device (page 59)	Adding a device involves verifying that the device exists, associating a device Handle with the device, and then 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.
Activate / De-Activate Device (page 60)	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. De-activating a device removes it from its operating state; it also disables interrupts and other global registers.
Statistics Collection (page 114)	Functions are provided to retrieve a snapshot of the various counts that are being accumulated by the ATLAS device. These routines should be invoked often enough to avoid letting the counters to rollover.
Interrupt Servicing / Polling (page 116)	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.

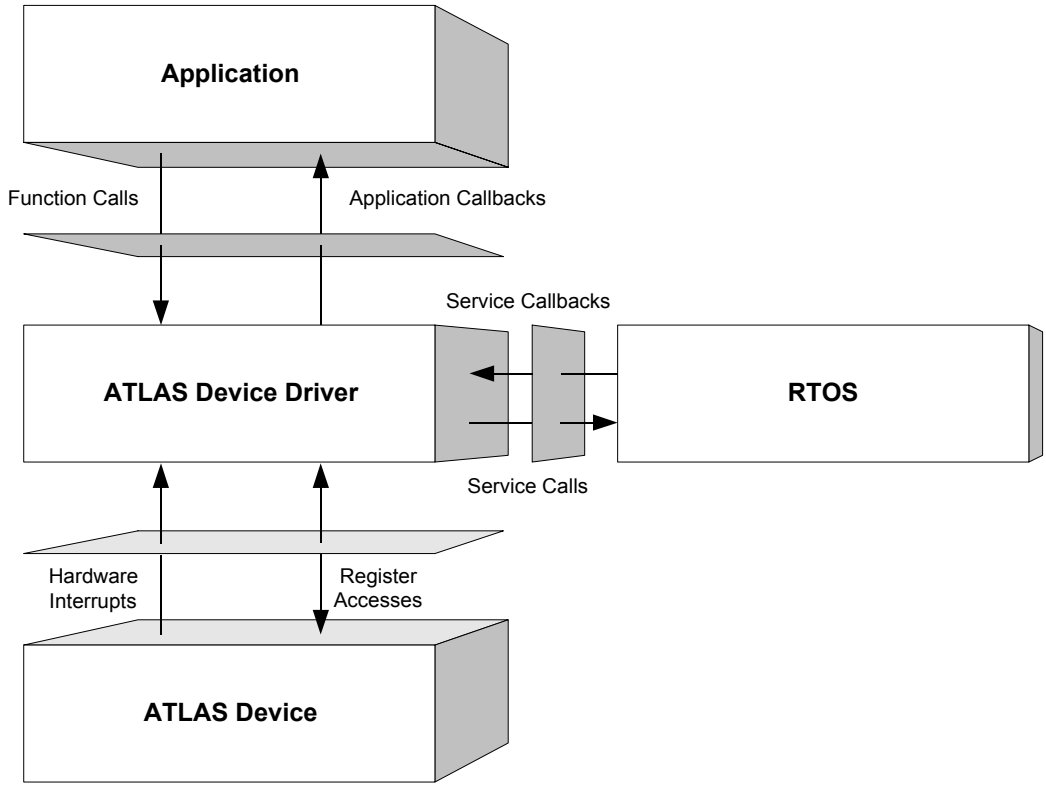
3 SOFTWARE ARCHITECTURE

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

3.1 Driver Interfaces

Figure 1 illustrates the external interfaces defined for the ATLAS device driver.

Figure 1: Driver Interfaces



3.2 Application Programming Interface

The driver's API is a collection of high level functions that can be called by application code to configure, control, and monitor the ATLAS 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 (see below).

3.3 Driver Hardware Interface

The Hardware Interface provides routines to read and write ATLAS registers. The Hardware Interface also provides a template for an Interrupt Service Routine (ISR) that is called when a hardware interrupt is raised. This routine needs to be modified based on the interrupt configuration of the system being used.

3.4 RTOS Interface

The RTOS interface module provides functions that enable the driver to use RTOS services. The ATLAS driver requires memory, interrupt, and preemption services from the RTOS.

The RTOS interface functions perform the following tasks for the ATLAS device and driver:

- Allocate and deallocate memory
- Manage buffers for the DPR and ISR
- Start and stop task execution

The RTOS interface also includes service callbacks. These functions are called by the driver in order to use RTOS service calls, such as install interrupts and start timers.

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

3.5 Main Components

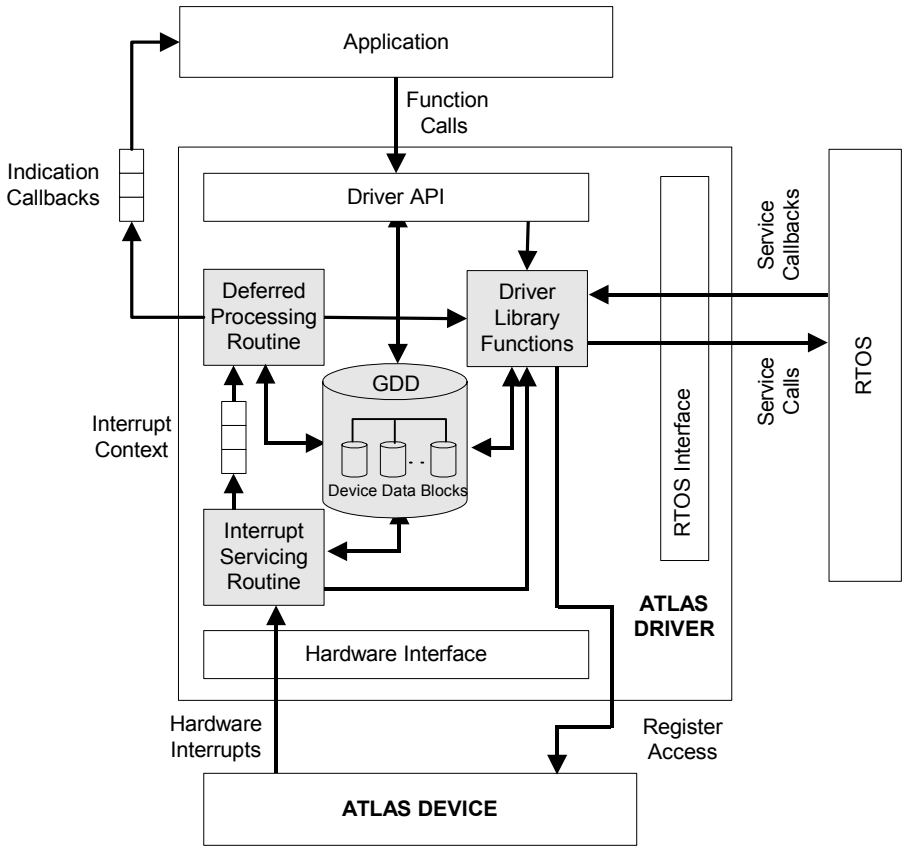
Figure 2 illustrates the top level architectural components of the ATLAS device driver. This applies in both polled and interrupt driven operation. Polled operations call ISR periodically. When in interrupt operation mode, the interrupt directly triggers the ISR.

The driver includes four main modules:

- Driver library module

- Device data-block module
- Interrupt-service routine module
- Deferred-processing routine module

Figure 2: ATLAS driver architecture



Driver Library Module

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

Global Driver Database and Device Data-Blocks

The Global Driver Database (GDD) is the top layer data structure. It is created by the ATLAS device driver to keep track of its initialization and operating parameters, modes, and dynamic data.

The Device Data Block is contained in the GDD and is initialized by the device Module for each ATLAS device that is registered. There is one DDB per device and there is a limit on the number of DDBs, and that limit is set by the user when the module is initialized. The DDB is used to store context information about one device, such as :

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

The driver allocates context memory for the DDB when the driver registers a new device.

Interrupt-Service Routine Module

The ATLAS driver provides an ISR called `atlasISR` that checks if any valid interrupt conditions are 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 `atlasISR`, is system and RTOS dependent. Therefore, it is outside the scope of the driver.

See page 120 for a detailed explanation of the platform specific routines that must be supplied by the user.

Deferred-Processing Routine Module

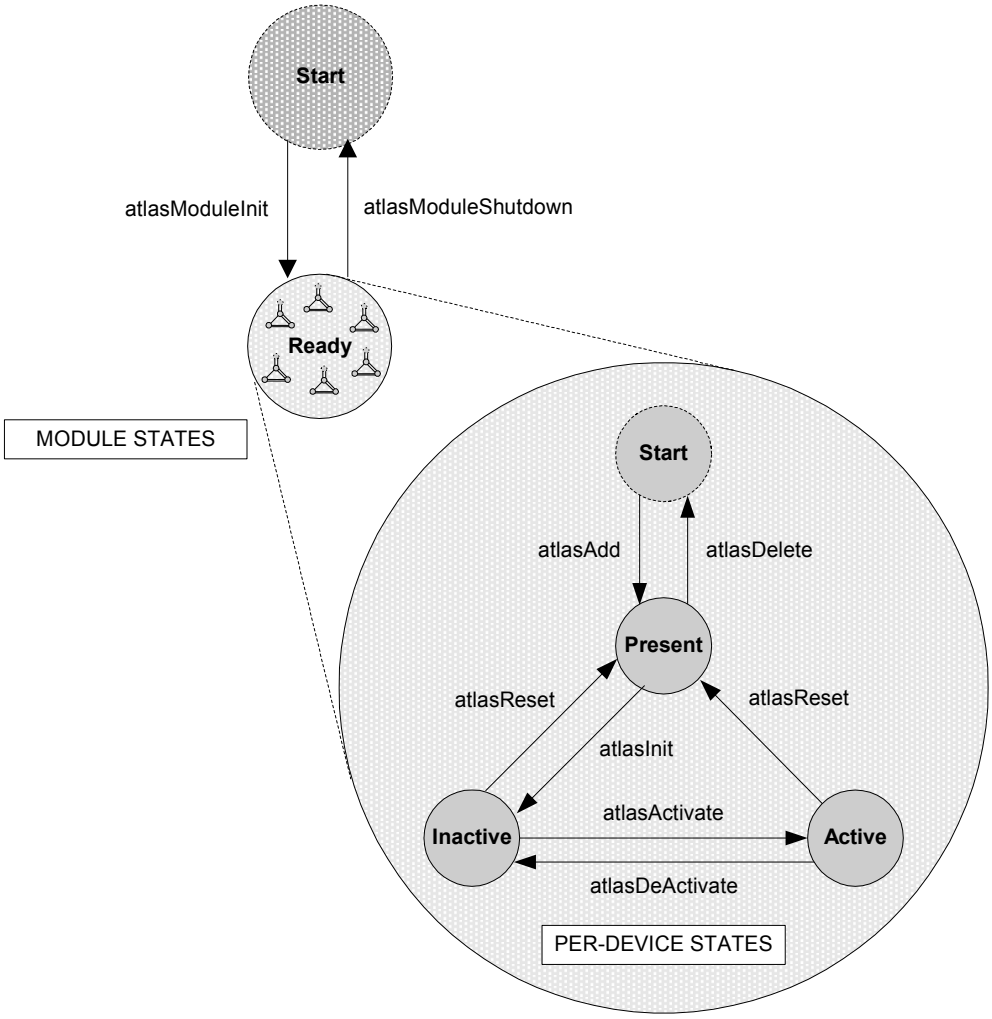
The Deferred-Processing Routine Module provided by the ATLAS driver (`atlasDPR`) clears and processes interrupt conditions for the device. Typically a system specific function, which runs as a separate task within the RTOS, executes the DPR.

See page 122 for a detailed explanation of the DPR and interrupt-servicing model.

3.6 Software State Description

Figure 3 shows the software state diagrams for the ATLAS module and device(s) as maintained by the driver.

Figure 3: Software States



The diagram shows state transitions made on the successful execution of the corresponding transition routines. State information helps maintain the integrity of the GDD and DDB(s) by controlling the set of operations allowed in each state.

3.7 Module States

Start

The ATLAS driver Module is not initialized. The only API function accepted in this state is `atlasModuleInit`. In this state the driver does not hold any RTOS resources (memory, timers, etc), has no running tasks, and performs no actions.

Ready

The normal operating state for the driver module is "Ready" and can be entered by a call to `atlasModuleInit`. The Global Driver Database has been allocated and loaded with current data; and the RTOS has responded favorably to all the requests sent to it by the driver. The only API functions accepted in this state is `atlasModuleShutdown`. The driver Module remains in this state while devices are in operation. Add devices via `atlasAdd`.

3.8 Device States

The following is a description of the ATLAS per-device states.

Start

The ATLAS device is not initialized. The only API function accepted in this state is `atlasAdd`. In this state the device is unknown by the driver and performs no actions.

Present

The ATLAS Device has been successfully added via the API function `atlasAdd`. A Device Data Block (DDB) is associated to the device and a device handle is provided for the USER. In this state, the device performs no actions. The only API functions accepted in this state are `atlasInit` and `atlasDelete`.

Active

The normal operating state for the device(s) enters by a call to `atlasActivate`. State changes initiate from the ACTIVE state via `atlasDeActivate`, `atlasReset` and `atlasDelete`.

Inactive

Enter "Inactive" via the `atlasInit` or `atlasDeActivate` function calls. In this state the device remains configured but all data functions de-activate. This includes interrupts and Alarms, Status and Statistics functions. `atlasActivate` will return the device to the ACTIVE state, while `atlasReset` or `atlasDelete` will de-configure the device. Queues are torn down.

3.9 Processing Flows

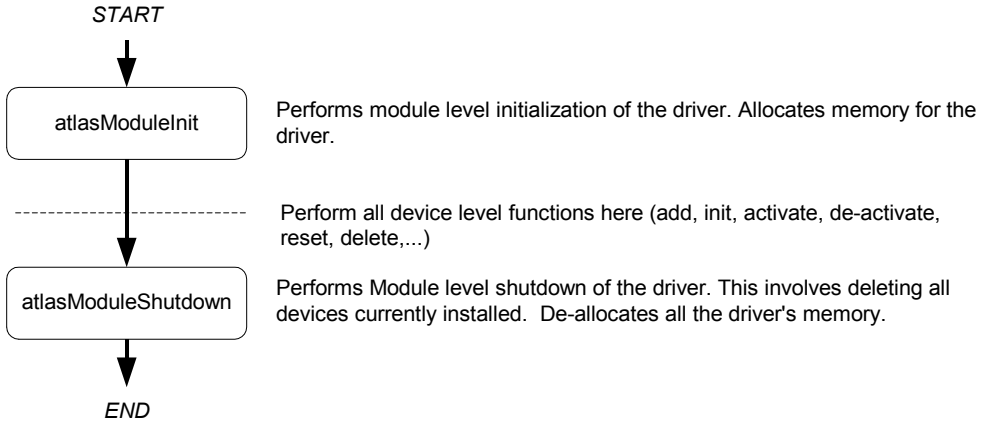
This section describes the main processing flows of the ATLAS driver.

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 driver API must be invoked.

Module Management

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

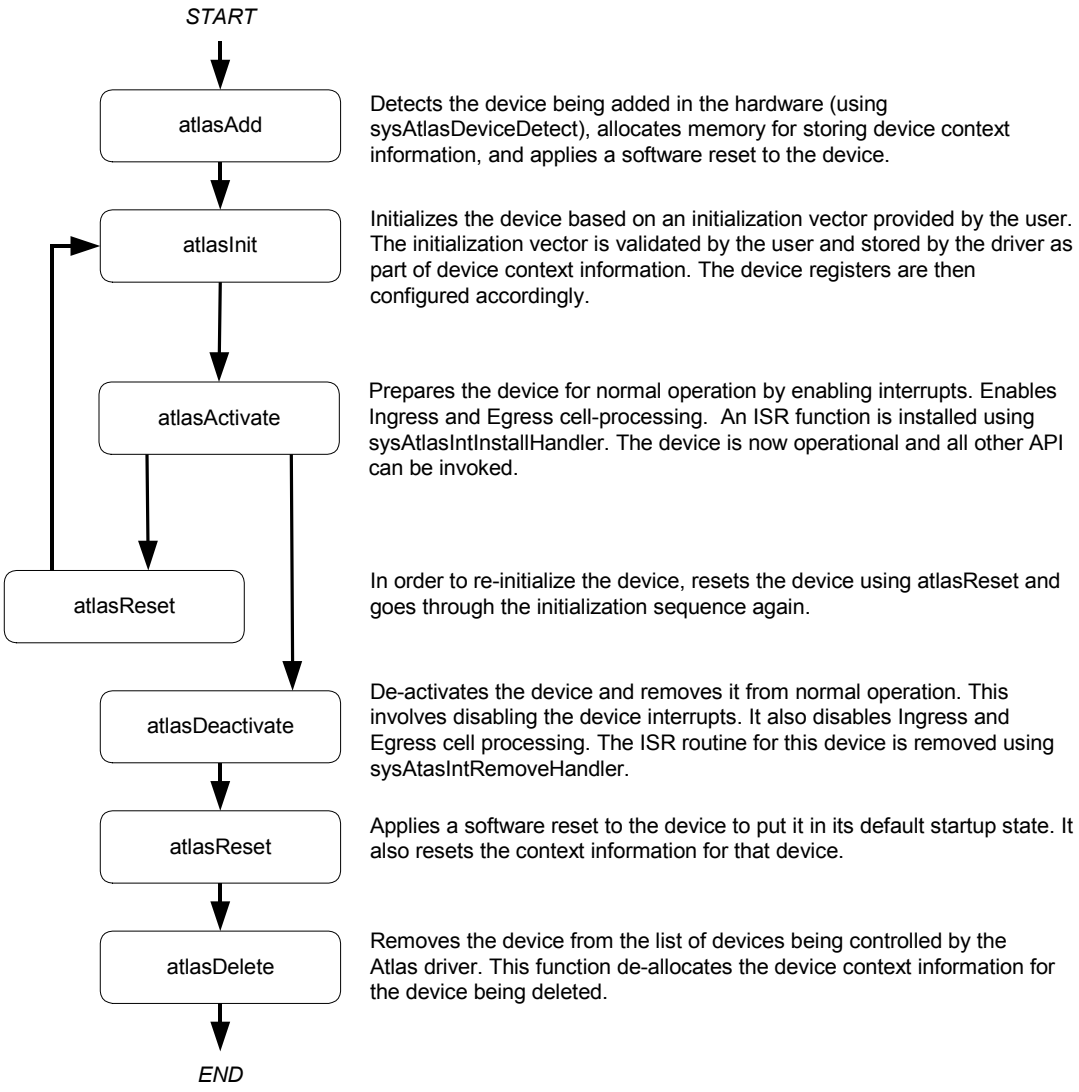
Figure 4: Module Management Flow Diagram



Device Initialization, Re-initialization, and Shutdown

The following figure shows the functions and process that the driver uses to initialize, re-initialize, and shutdown the ATLAS device.

Figure 5: Device Initialization, Re-initialization, and Shutdown



Cell Extraction

The cell extraction is performed in `atlasDPR`. When a cell is received in the microprocessor cell FIFO, the `sysTxMsg` task is started and the `atlasRxCell` API is called. Only the OAM Loopback and Activation/Deactivation cells are processed in `atlasRxCell`. The user must supply two callback functions: `indOamRx` and `indBackEci`. See Section 7.2, Callback Functions, to learn more about these callbacks.

3.10 Interrupt Servicing

The interrupt servicing code includes some system specific code (routines prefixed by `sys`) that is typically implemented by the user for their system as well some generic code (prefixed by `ATLAS`) provided by the driver that does not change from system to system.

`The sysAtlasIntHandler` is a system-specific interrupt handler routine implemented by the user that is installed in the interrupt vector table of the system processor. This routine is invoked when one or more ATLAS devices interrupt the processor. `sysAtlasIntHandler` then invokes a driver provided routine, `atlasISR`, for each device in the active state. `atlasISR` reads the Master Interrupt Status register of the ATLAS and returns with this status information if a valid status bit is set. This status information is then saved by `sysAtlasIntHandler`. Saving the status information for deferred processing is typically implemented as a message queue. Status information is sent to this queue by the `sysAtlasIntHandler`. This information is then dequeued for later processing.

`The sysAtlasDPRtask` is a system-specific routine that runs as a separate task within the RTOS. In the message queue implementation model, this task has an associated message queue. The task waits for messages on this message queue. When a message is received, the driver-supplied Deferred Processing Routine (`atlasDPR`) is invoked. The `atlasDPR` processes the status information and takes appropriate action based on the specific interrupt condition detected. Since the nature of this processing can differ from system to system, the DPR invokes different indication callbacks for different interrupt conditions. These callbacks can then be customized to fit the user's specific requirements.

Note that since the `atlasISR` and `atlasDPR` routines themselves do not specify a communication mechanism, the user is given full flexibility in choosing a communication mechanism between the two. A convenient way to implement this communication mechanism is to use a message queue, a service that is provided by most RTOSs.

The two system specific routines, `sysAtlasIntHandler` and `sysAtlasDPRtask`, are implemented by the user. `sysAtlasIntHandler` is installed in the interrupt vector table of the processor when the `sysAtlasIntInstallHandler` is called for the first time. The `sysAtlasDPRtask` routine is also spawned as a task during this first time invocation of the `sysAtlasIntInstallHandler`. In addition, `sysAtlasIntInstallHandler` also creates the communication channel between `sysAtlasIntHandler` and `sysAtlasDPRtask`. This communication channel is most commonly a message queue associated with the `sysAtlasDPRtask`.

Similarly, during removal of interrupts, the `sysAtlasIntHandler` routine is removed from the microprocessor's interrupt vector table and the task associated with the `sysAtlasDPRtask` is deleted.

If interrupts are not available they can be simulated by providing a high priority task that polls the active ATLAS devices periodically.

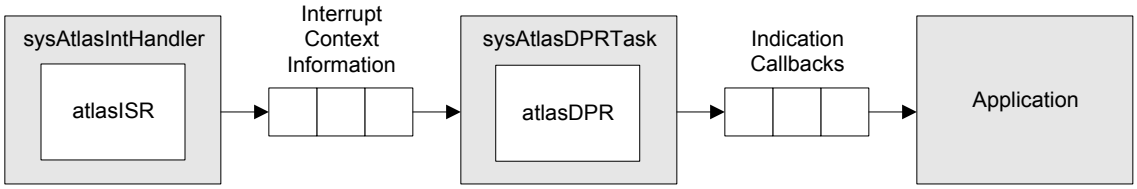
For each active ATLAS device, this high priority task calls the `atlasISR`. The ISR will then process the state of each device and may, if necessary, pass the device status to the `atlasDPR`. The `altasDPR` behaves the same in either model.

The high priority polling task should be at a higher priority than the `altasDPR`; also, it should poll frequently enough to handle device conditions in a timely manner.

For the polling model, rather than installing the `sysAtlasIntHandler` as an interrupt, the high priority polling task should be activated. Similarly, rather than removing the interrupt handler, the polling task should be stopped. The DPR processing should be the same in both cases.

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

Figure 6: Interrupt Service Model



Note: Instead of using an interrupt service model, you can use a polling service model in the ATLAS driver to process the device's event-indication registers (see page 28).

Calling atlasISR

An interrupt handler function, which is system dependent, calls `atlasISR`. Before this, however, the low-level interrupt-handler function traps the device interrupts. You must implement this function for your system. For your reference, an example implementation of the interrupt handler (`sysAtlasIntHandler`) appears on page 120. You can customize this example to suit your needs.

The interrupt handler (`sysAtlasIntHandler`) installs in the interrupt vector table of the system processor. It calls when one or more ATLAS devices interrupt the processor. The interrupt handler subsequently calls `atlasISR` for each device in the active state. `atlasISR` reads from the ATLAS interrupt-status registers.

If there are interrupts outstanding, this is indicated by `atlasISR`. The `sysAtlasIntHandler` then sends a message to the DPR task that consists of the device handle of the ATLAS device and the indicated interrupts.

The ISR also examines the interrupt itself for exception conditions. If these are present, it will call the system exception callback directly.

Note: Normally you should save the status information for deferred processing by implementing a message queue. The interrupt handler uses `sysAtlasIntHandler` to send the status information to the queue.

Calling atlasDPR

`sysAtlasDPRTask` is a system specific function that runs as a separate task within the RTOS. You should set the DPR task's priority higher than that of the application task(s) interacting with the ATLAS 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, `sysAtlasDPRTask` calls the DPR (`atlasDPR`). Then `atlasDPR` processes the status information to ensure a valid indication is present; after that it makes the callback.

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 `atlasDelete`. You can customize this callback to suit your system. See page 60 for a description of the callback function.

Note: Since the `atlasISR` and `atlasDPR` 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's provide.

You must implement the two system specific routines, `sysAtlasIntHandler` and `sysAtlasDPRTask`. When `sysAtlasIntInstallHandler` is called for the first time, `sysAtlasIntHandler` is installed in the interrupt vector table of the processor. The `sysAtlasDPRTask` routine is also spawned as a task during this first time invocation of `sysAtlasIntInstallHandler`. In addition, `sysAtlasIntInstallHandler` creates the communication channel between `sysAtlasIntHandler` and `sysAtlasDPRTask`. This communication channel is most commonly a message queue associated with `sysAtlasDPRTask`.

Similarly, during the removal of interrupts, the `sysAtlasIntHandler` function is removed from the microprocessor's interrupt vector table and the task associated with `sysAtlasDPRTask` is deleted when the last atlas device is removed.

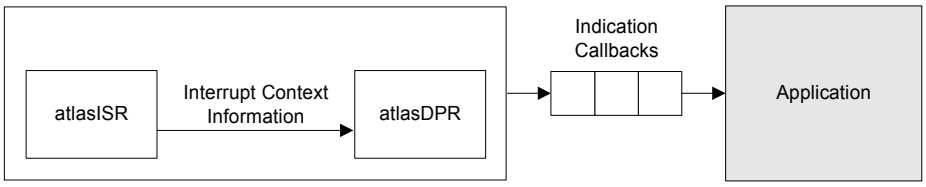
As a reference, this manual provides example implementations of the interrupt installation and removal functions. For more information about the interrupt removal function and prototype, see page 121. You can customize these prototypes to suit your specific needs.

Polling Servicing

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

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

Figure 7: Polling Service Model



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

In `sysAtlasIntPollTask`, the driver-supplied ISR (`atlasISR`) is periodically called for each device in the active state. The `atlasISR` reads from the interrupt-status registers of the ATLAS. If some valid status bits are set, it processes the status information and takes appropriate action based on the specific interrupt condition detected.

Similarly, during removal of polling service, the task associated with `sysAtlasPollTask` is deleted when the last ATLAS device is deactivated.

4 DATA STRUCTURES

The following are the main data structures employed by the ATLAS driver.

4.1 Global Driver Database

The Global Driver Database (GDD) stores module level data, such as the number of devices that the driver controls and an array of pointers to the individual Device Data Blocks. The GDD is allocated when the ATLAS module is started

Table 1: Global Driver Database: sATLS_GDD

Field Name	Field Type	Field Description
numDevs	UINT1	Number of Devices currently managed
isrManager	sATLS_ISR_MANAGER	ISR Manager.
pDdb[ATLS_MAX_NUM_DEVS]	sATLS_DDB	Array of pointers to the individual DDBs

Interrupt Service routine manager

The ISR manager is responsible for installing and uninstalling the ISR into the vector table.

Table 2: Interrupt service routine manager: sATLS_ISR_MANAGER

Field Name	Field Type	Field Description
numDevs	UINT1	Number of Devices currently managed

Device Data Block

The Device Data Block (DDB) stores control and status information for a single ATLAS device. The DDB is allocated when a new device is added and is de-allocated when an existing device is deleted.

Table 3: Device Data Block: sATLS_DDB

Field Name	Field Type	Field Description
EdevState	eATLS_STATE	Device state, which can be one of the following: ATLS_EMPTY ATLS_PRESENT ATLS_INIT ATLS_ACTIVE
SinitVector	sATLS_INIT_VECTOR	Device configuration information passed by the user to the driver. The driver writes to the appropriate ATLAS registers based on the contents of this vector.
UsrCtxt	ATLS_USR_CTXT	This variable is used to store the user's context for the device. It is passed as an input parameter when the driver invokes an application callback.
PsysInfo	VOID *	System specific information.
deviceAddr	UINT1 *	Address of the ATLAS device in the memory map
ingressMaxVCs	UINT4	Maximum number of ingress VCs
egressMaxVCs	UINT4	Maximum number of egress VCs
evcInitialised	UINT1	Indicates that the Egress VC has been initialized.
ivcInitialised	UINT1	Indicates that the Ingress VC has been initialized.
ivcPrimaryTable	sATLS_IVC_PRIMARY_TABLE	This represents the Ingress Primary table for the Device.
ivcSecondaryTable	sATLS_IVC_SECONDARY_TABLE	This represents the Ingress Secondary table for the Device.
ivcSecondaryAddrList	sATLS_IVC_SECONDARY_ADDR_LIST	Ingress Secondary Address List.
ivcRecordAddrFree	sATLS_IVC_RECORD_ADDR_FREE	Indicates whether an Ingress secondary record is free.
ivcSecondaryKeys	sATLS_IVC_SECONDARY_KEYS	The Ingress Secondary Address Keys.
evcRecordAddrExists	sATLS_EVC_RECORD_ADDR_EXISTS	Indicates whether an Egress record is in use.

Field Name	Field Type	Field Description
ingressBank1PMFree	sATLS_INGRESS_PM_FREE	Indicates whether a PM Record is free in the Ingress Bank 1 list.
ingressBank2PMFree	sATLS_INGRESS_PM_FREE	Indicates whether a PM Record is free in the Ingress Bank 2 list.
egressBank1PMFree	sATLS_EGRESS_PM_FREE	Indicates whether a PM Record is free in the Egress Bank 2 list.
egressBank2PMFree	sATLS_EGRESS_PM_FREE	Indicates whether a PM Record is free in the Egress Bank 2 list.
ivcPhyIdMask	UINT2	Ingress phy Id mask
ivcPhyIdLength	UINT2	Ingress phy Id length
ivcMaxPhyIdValue	UINT2	Ingress Max phy Id
ivcFieldAMask	UINT2	Ingress Field A Mask
ivcFieldALength	UINT2	Ingress Field A Length
ivcMaxFieldAValue	UINT2	Ingress Max Field A Value
ivcFieldBMask	UINT2	Ingress Field B Mask
ivcFieldBLength	UINT2	Ingress Field B Length
ivcMaxFieldBValue	UINT2	Ingress Max Field B Value
evcPhyIdMask	UINT2	Egress phy Id mask
evcPhyIdLength	UINT2	Egress phy Id length
evcMaxPhyIdValue	UINT2	Egress Max phy Id
evcFieldAMask	UINT2	Egress Field A Mask
evcFieldALength	UINT2	Egress Field A Length
evcMaxFieldAValue	UINT2	Egress Max Field A Value
evcFieldBMask	UINT2	Egress Field B Mask
evcFieldBLength	UINT2	Egress Field B Length
evcMaxFieldBValue	UINT2	Egress Max Field B Value
defaultEvcRam	sATLS_EVCRAM_TYPE	Default EVC Ram contents. This is the device encoded version of the Default EVC record in the Initialization Vector
sCb	sATLS_CB	Callback control block.

Initialization Vector

The user defines the initialization vector before initializing an ATLAS device. The initialization vector contains various configuration parameters that are used by the driver to program the ATLAS control registers.

The initialization vector specifies many configuration parameters. These are broken into functional areas.

Table 4: ATLAS Initialization Vector: sATLS_INIT_VECTOR

Field Name	Field Type	Field Description
masterCfg	UINT2	Master configuration register
intEnRegs	sATLS_INT_EN_REGS	Interrupt enables registers
ingressCfg	sATLS_INGRESS_REGS	Ingress interface configuration
egressCfg	sATLS_EGRESS_REGS	Egress interface configuration
mPCellCfg	sATLS_MPCELL_REGS	Microprocessor cell interface registers
pMCfg	sALTS_PM_REGS	Performance memory registers
defEgRecord	sATLS_EGRESS_RECORD	Default Egress Record. This is used to populate inactive Egress records.
indOamRx	ATLS_IND_OAM_RX	OAM support callback.
indException	ATLS_IND_SYS_EXCEPT	Exceptions callback
indIndication	ATLS_IND_INDICATION	General purpose Callback.
indBackEci	ATLS_IND_BACKWARD_ECI	Backward connection ID request callback.

Interrupt Enable Registers

The ATLAS Interrupt Enable Register settings are stored in the following structure.

Table 5: Interrupt Enable Registers: sATLS_INT_EN_REGS

Field Name	Field Type	Field Description
intEn[0]	UINT2	Interrupt enable register 1
intEn[1]	UINT2	Interrupt enable register 2
intEn[2]	UINT2	Interrupt enable register 3
intEn[3]	UINT2	Interrupt enable register 4

Field Name	Field Type	Field Description
intEn[4]	UINT2	Interrupt enable register 5

Ingress Registers

The Ingress Register information collects together all the registers used to configure the Ingress portion of the device. This includes the following functional areas:

- Ingress cell interface
- Global ingress input/output PHY
- Ingress OAM
- Ingress PHY/connection policing
- Ingress VC table configuration

Table 6: ATLAS Ingress Register Information: sATLS_INGRESS_REGS

Field Name	Field Type	Field Description
ingInCellCfg1	UINT2	Ingress Input Cell Interface Configuration 1
ingInCellCfg2	UINT2	Ingress Input Cell Interface Configuration 2
ingOutCellCfg1	UINT2	Ingess Output Cell Interface Configuration 1
ingOutCellCfg2	UINT2	Ingess Output Cell Interface Configuration 2
ingRDIBackOAMcfg	UINT2	Ingress RDI Backward OAM Cell Interface Configuration
ingBackRptCfg	UINT2	Ingress Backward Reporting OAM Cell Interface Configuration
ingSearchCfg	UINT2	Ingress Search Engine Configuration
ingFieldACfg	UINT2	FieldA Location and Length
ingFieldBCfg	UINT2	Field B Location and Length
ingCellProcCfg1	UINT2	Ingress Cell Processor Configuration 1
ingPHYPolice1	UINT2	Ingress PHY Policing 1
ingPHYPolice2	UINT2	Ingress PHY Policing 2
ingPHYPolCfg1_2	UINT2	Ingress PHY Policing Configuration 1&2
ingPHYPolCfg3_4	UINT2	Ingress PHY Policing Configuration 3&4
ingConPolCfg1	UINT2	Ingress Connection Policing Configuration 1

Field Name	Field Type	Field Description
ingConPolCfg2	UINT2	Ingress Connection Policing Configuration 2
ingConPolCfg3	UINT2	Ingress Connection Policing Configuration 3
ingConPolCfg4	UINT2	Ingress Connection Policing Configuration 4
ingConPolCfg5	UINT2	Ingress Connection Policing Configuration 5
ingConPolCfg6	UINT2	Ingress Connection Policing Configuration 6
ingConPolCfg7	UINT2	Ingress Connection Policing Configuration 7
ingConPolCfg8	UINT2	Ingress Connection Policing Configuration 8
ingPolCfgNonCompCnt	UINT2	Ingress Policing Configuration and Non-Compliant Cell Counting
ingCellRtgCfg	UINT2	Ingress Cell Routing Configuration
ingOAMCellGenCfg	UINT2	Ingress OAM Cell Generation Configuration
ingPerPHYAISCfg1	UINT2	Ingress Per-PHY AIS Cell Generation Control 1
ingPerPHYAISCfg2	UINT2	Ingress Per-PHY AIS Cell Generation Control 2
ingPerPHYRDICfg1	UINT2	Ingress Per-PHY RDI Cell Generation Control 1
ingPerPHYRDICfg2	UINT2	Ingress Per-PHY RDI Cell Generation Control 2
ingOAMDefType0_1	UINT2	Ingress OAM Defect Type 0&1
ingOAMDefType2_3	UINT2	Ingress OAM Defect Type 2&3
ingOAMDefType4_5	UINT2	Ingress OAM Defect Type 4&5
ingOAMDefType6_7	UINT2	Ingress OAM Defect Type 6&7
ingOAMDefType8_9	UINT2	Ingress OAM Defect Type 8&9
ingOAMDefType10_11	UINT2	Ingress OAM Defect Type 10&11
ingOAMDefType12_13	UINT2	Ingress OAM Defect Type 12&13
ingOAMDefType14_15	UINT2	Ingress OAM Defect Type 14&15
ingDefLoc0_1	UINT2	Ingress Defect Location Octets 0&1
ingDefLoc2_3	UINT2	Ingress Defect Location Octets 2&3
ingDefLoc4_5	UINT2	Ingress Defect Location Octets 4&5
ingDefLoc6_7	UINT2	Ingress Defect Location Octets 6&7
ingDefLoc8_9	UINT2	Ingress Defect Location Octets 8&9
ingDefLoc10_11	UINT2	Ingress Defect Location Octets 10&11
ingDefLoc12_13	UINT2	Ingress Defect Location Octets 12&13
ingDefLoc14_15	UINT2	Ingress Defect Location Octets 14&15

Field Name	Field Type	Field Description
ingCellCntCfg1	UINT2	Ingress Cell Counting Configuration 1
ingCellCntCfg2	UINT2	Ingress Cell Counting Configuration 2
ingCellProcCfg2	UINT2	Ingress Cell Processor Configuration 2
ingMaxFrame	UINT2	Ingress Maximum Frame Length Count
ingVCTableMax	UINT2	Ingress VC Table Maximum Index
ingPerPHYAPSIndCfg1	UINT2	Ingress Per-PHY APS Indication Configuration 1
ingPerPHYAPSIndCfg2	UINT2	Ingress Per-PHY APS Indication Configuration 2
ingPerPHYCountCfg	UINT2	Ingress Per-PHY Counter Configuration

Egress Registers

The Egress Register information collects together all the registers used to configure the Egress portion of the device. This includes the following functional areas:

- Egress cell interface
- Global egress input/output PHY
- Egress OAM
- Egress Per-PHY counter configuration
- Egress VC table configuration

Table 7: ATLAS Egress Register Information: sATLS_EGRESS_REGS

Field Name	Field Type	Field Description
egInCellCfg1	UINT2	Egress Input Cell Interface Configuration 1
egInCellCfg2	UINT2	Egress Input Cell Interface Configuration 2
egOutCellCfg1	UINT2	Egress Output Cell Interface Configuration 1
egOutCellCfg2	UINT2	Egress Output Cell Interface Configuration 2
egRDIBackOAMCfg	UINT2	Egress RDI Backward OAM Cell Interface Configuration
egBackRptCfg	UINT2	Egress Backward Reporting OAM Cell Interface Configuration
egCellProcCfg	UINT2	Egress Cell Processor Configuration
egCellRtgCfg	UINT2	Egress Cell Routing Configuration

Field Name	Field Type	Field Description
egCellDirCfg1	UINT2	Egress Cell Processor Direct Lookup Index Configuration 1
egCellDirCfg2	UINT2	Egress Cell Processor Direct Lookup Index Configuration 2
egBackOAMHOLCfg	UINT2	Egress Backward OAM Cell Interface Head Of Line Time Out Configuration
egPacedAISCCCfg	UINT2	Egress Paced AIS/CC Cell Generation Configuration
egPacedFwdPMCfg	UINT2	Egress Paced Fwd PM Cell Generation Configuration
egPerPHYAISCfg1	UINT2	Egress Per-PHY AIS Cell Generation Control 1
egPerPHYAISCfg2	UINT2	Egress Per-PHY AIS Cell Generation Control 2
egPerPHYRDICfg1	UINT2	Egress Per-PHY RDI Cell Generation Control 1
egPerPHYRDICfg2	UINT2	Egress Per-PHY RDI Cell Generation Control 2
egPerPHYAPSInd1	UINT2	Egress PerPHY APS Indication Configuration 1
egPerPHYAPSInd2	UINT2	Egress PerPHY APS Indication Configuration 2
egVCCountingCfg1	UINT2	Egress VC Table Counting Configuration 1
egVCCountingCfg2	UINT2	Egress VC Table Counting Configuration 2
egOAMDefType0_1	UINT2	Egress OAM Defect Type 0&1
egOAMDefType2_3	UINT2	Egress OAM Defect Type 2&3
egOAMDefType4_5	UINT2	Egress OAM Defect Type 4&5
egOAMDefType6_7	UINT2	Egress OAM Defect Type 6&7
egOAMDefType8_9	UINT2	Egress OAM Defect Type 8&9
egOAMDefType10_11	UINT2	Egress OAM Defect Type 10&11
egOAMDefType12_13	UINT2	Egress OAM Defect Type 12&13
egOAMDefType14_15	UINT2	Egress OAM Defect Type 14&15
egDefLoc0_1	UINT2	Egress Defect Location Octets 0&1
egDefLoc2_3	UINT2	Egress Defect Location Octets 2&3
egDefLoc4_5	UINT2	Egress Defect Location Octets 4&5
egDefLoc6_7	UINT2	Egress Defect Location Octets 6&7
egDefLoc8_9	UINT2	Egress Defect Location Octets 8&9
egDefLoc10_11	UINT2	Egress Defect Location Octets 10&11

Field Name	Field Type	Field Description
egDefLoc12_13	UINT2	Egress Defect Location Octets 12&13
egDefLoc14_15	UINT2	Egress Defect Location Octets 14&15
egVCTableMax	UINT2	Egress VC Table Maximum Index
egPerPHYCountCfg	UINT2	Egress Per-PHY Counter Configuration

Performance Monitoring Configuration

The performance Monitoring Configuration information collects together all the registers used to configure the Global control of the Performance Monitoring Memory.

This includes the following functional areas:

- Ingress Monitoring Thresholds
- Egress Monitoring Thresholds

Table 8: ATLAS Performance Monitoring Register Information: sATLS_PM_REGS

Field Name	Field Type	Field Description
ingBackOAMPacingCfg	UINT2	Ingress Backward OAM Cell Pacing Configuration
ingCellProcF4PMMap	UINT2	Ingess Cell Processor F4-PM Flow VCI Map
ingCellProcF5PMMap	UINT2	Ingress Cell Processor f5-PM Flow PTI Map
ingPacedFwdPMCell	UINT2	Ingress Paced Fwd PM Cell Generation
ingPMThreshA1	UINT2	Ingress Performance Monitoring Threshold A1
ingPMThreshA2	UINT2	Ingress Performance Monitoring Threshold A2
ingPMThreshB1	UINT2	Ingress Performance Monitoring Threshold B1
ingPMThreshB2	UINT2	Ingress Performance Monitoring Threshold B2
ingPMThreshC1	UINT2	Ingress Performance Monitoring Threshold C1
ingPMThreshC2	UINT2	Ingress Performance Monitoring Threshold C2
ingPMThreshD1	UINT2	Ingress Performance Monitoring Threshold D1
ingPMThreshD2	UINT2	Ingress Performance Monitoring Threshold D2
egBackOAMPacingCfg	UINT2	Egress Backward OAM Cell Pacing Configuration
egCellProcF4PMMap	UINT2	Egess Cell Processor F4-PM Flow VCI Map
egCellProcF5PMMap	UINT2	Egress Cell Processor f5-PM Flow PTI Map

Field Name	Field Type	Field Description
egPMThreshA1	UINT2	Egress Performance Monitoring Threshold A1
egPMThreshA2	UINT2	Egress Performance Monitoring Threshold A2
egPMThreshB1	UINT2	Egress Performance Monitoring Threshold B1
egPMThreshB2	UINT2	Egress Performance Monitoring Threshold B2
egPMThreshC1	UINT2	Egress Performance Monitoring Threshold C1
egPMThreshC2	UINT2	Egress Performance Monitoring Threshold C2
egPMThreshD1	UINT2	Egress Performance Monitoring Threshold D1
egPMThreshD2	UINT2	Egress Performance Monitoring Threshold D2

Microprocessor Cell Interface Configuration

Microprocessor Cell Interface information collects together all the registers used to configure the Microprocessor Ingress/Egress Cell interfaces.

This includes the following functional areas:

- Ingress Microprocessor cell interface
- Egress Microprocessor cell interface

Table 9: ATLAS Micro Processor Register Information: sATLS_MPCELL_REGS

Field Name	Field Type	Field Description
ingMPCExtCfg	UINT2	Ingress Microprocessor Extract Cell Configuration
ingMPCInsCfg	UINT2	Ingress Microprocessor Insert Cell Configuration
egMPCExtCfg	UINT2	Egress Microprocessor Extract Cell Configuration
egMPCInsCfg	UINT2	Egress Microprocessor Insert Cell Configuration

Interrupt Status information

The ATLAS Interrupt Status register information is returned to the application in the following structure.

Table 10: Interrupt Status Registers: sATLS_INT_STATUS

Field Name	Field Type	Field Description
intSt[0]	UINT2	Interrupt status register 1
intSt[1]	UINT2	Interrupt status register 2
intSt[2]	UINT2	Interrupt status register 3
intSt[3]	UINT2	Interrupt status register 4
intSt[4]	UINT2	Interrupt status register 5

Ingress Record Type

This record mirrors the Ingress VC record documented in the ATLAS Hardware Specification, with some addressing information prepended.

Table 11: Ingress Record Type: sATLS_INGRESS_RECORD

Field Name	Field Type	Field Description
fieldA	UINT2	Field a, used as search key only, not part of the device record.
cLPccEn	UINT1	CLP Conformance Check Enable
f4ToF5Ais	UINT1	F4 to F5 AIS Generation Control
phyId	UINT1	PHY ID
row1Unused0	UINT1	Unused
pmActive2	UINT1	PM2 on/off flag
pmAddr2	UINT1	PM2 Session Address
pmActive1	UINT1	PM1 on/off flag
pmAddr1	UINT1	PM1 Session Address
nni	UINT1	Network-Network Interface flag
fieldB	UINT2	Field B
vpi	UINT2	Virtual Path Identifier
vci	UINT2	Virtual Connection Identifier
status	UINT1	Ingress internal Status indicator
configuration	UINT2	Ingress VC Table configuration field
internalStatus	UINT4	Ingress internal status field

Field Name	Field Type	Field Description
oamConfiguration	UINT2	Ingress VC table OAM configuration field
vpcPointer	UINT2	VPC pointer
cocup	UINT1	Conditional Conformance update
row3Reserved	UINT1	Unused
tat2	UINT4	Theoretical Arrival Time 2
tat1	UINT4	Theoretical Arrival Time 1
gfr	UINT1	Guaranteed Frame Rate
policeConfiguration	UINT1	Policing Configuration
action2	UINT1	Action on Non-conforming GCRA2
action1	UINT1	Action on Non-conforming GCRA1
i2	UINT2	Increment 2
l2	UINT2	Limit 2
i1	UINT2	Increment 1
l1	UINT2	Limit 1
phyPolice	UINT1	PHY Policing on/off
remainingFrameCount	UINT2	Remaining Frame Count
violate	UINT1	Tag all cells as Violating
gfrState	UINT1	Internal GFR State
nonCompliant3	UINT2	Non-compliant Cell count 1
nonCompliant2	UINT2	Non-compliant Cell count 2
nonCompliant1	UINT2	Non-compliant Cell count 3
ingressCellCount2	UINT4	Ingress Cell Count 2
ingressCellCount1	UINT4	Ingress Cell Count 1
header[ATLS_IVC_HEADER_SIZE]	UINT1	New cell Header for translation
udf	UINT1	User Defined Field
prePo1	UINT1	PrePend/PostPend 1
prePo2	UINT1	PrePend/PostPend 2
prePo3	UINT1	PrePend/PostPend 3
prePo4	UINT1	PrePend/PostPend 4
prePo5	UINT1	PrePend/PostPend 5

Field Name	Field Type	Field Description
prePo6	UINT1	PrePend/PostPend 6
prePo7	UINT1	PrePend/PostPend 7
prePo8	UINT1	PrePend/PostPend 8
prePo9	UINT1	PrePend/PostPend 9
prePo10	UINT1	PrePend/PostPend 10
alternateIngressCellCount2	UINT4	Alternate Ingress Cell Counter 2
alternateIngressCellCount1	UINT4	Alternate Ingress Cell Counter 1
row10Unused1	UINT4	Unused
row10Unused0	UINT1	Unused
maximumFrameLength	UINT2	Maximum Frame Length
rxSegmentDefectType	UINT1	Segment AIS Type
rxEndToEndDefectType	UINT1	End to End AIS Type
rxEndToEndAisDefectLoc[ATLS_IVC_AIS_DEFECT_LOCATION_SIZE]	UINT1	End to End AIS Location
rxSegmentAisDefectLoc[ATLS_IVC_AIS_DEFECT_LOCATION_SIZE]	UINT1	Segment AIS Location

Ingress OAM Defect Record

This record collects together all the fields used to configure the OAM defect support in the Ingress VC table documented in the ATLAS Hardware Specification.

Table 12: Ingress OAM Defect Record: sATLS_INGRESS_OAMDEFECT

Field Name	Field Type	Field Description
rxSegmentDefectType	UINT1	Segment AIS type.
rxEndToEndDefectType	UINT1	End-to-end AIS type.
rxEndToEndDefectLoc[ATLS_IVC_AIS_DEFECT_LOCATION_SIZE]	UINT1	End-to-End AIS Location.
rxSegmentAisDefectLoc[ATLS_IVC_AIS_DEFECT_LOCATION_SIZE]	UINT1	Segment AIS Location

Ingress Configuration Record

This record collects together all the fields used to configure the VC connection in the Ingress VC table documented in the ATLAS Hardware Specification.

Table 13: Ingress Configuration Record: sATLS_INGRESS_CONFIG

Field Name	Field Type	Field Description
nni	UINT1	Network-network Interface flag
status	UINT1	Ingress VC table status field
configuration	UINT2	Ingress VC table configuration field
internalStatus	UINT4	Ingress internal status field
gfr	UINT1	Guaranteed frame rate
gfrState	UINT1	Internal gfr state
maximumFrameLength	UINT2	Maximum Frame length

Ingress OAM Configuration Record

This record collects together all the fields used to configure the OAM support in the Ingress VC table documented in the ATLAS Hardware Specification.

Table 14: Ingress OAM Configuration Record: sATLS_INGRESS_OAMCONFIG

Field Name	Field Type	Field Description
oamConfiguration	UINT2	Ingress VC table OAM configuration field
vpcPointer	UINT2	VPC pointer
f4ToF5AIS	UINT1	F4 to F5 AIS generation control

Ingress Policing Record

This record collects together all the fields used to configure the Policing parameters in the Ingress VC table documented in the ATLAS Hardware Specification.

Table 15: Ingress Policing Record: sATLS_INGRESS_POLICING

Field Name	Field Type	Field Description
policeConfiguration	UINT2	Policing Configuration
phyPolice	UINT1	PHY Policing ON/OFF
cLPccEn	UINT1	CLP Conformance Check enable
cocup	UINT1	Conditional Conformance update
tat2	UINT4	Theoretical Arrival Time 2
tat1	UINT4	Theoretical Arrival Time 1
action2	UINT1	Action of non-conforming GCRA2
action1	UINT1	Action of non-conforming GCRA1
i2	UINT2	Increment 2
l2	UINT2	Limit 2
i1	UINT2	Increment 1
l1	UINT2	Limit 1
violate	UINT1	Tag all cells as violating

Ingress Counts Record

This record collects together all the fields used to read the Counts parameters in the Ingress VC table documented in the ATLAS Hardware Specification.

Table 16: Ingress Counts Record: sATLS_INGRESS_COUNTS

Field Name	Field Type	Field Description
nonCompliant3	UINT2	Non-Compliant cell count 3
nonCompliant2	UINT2	Non-Compliant cell count 2
nonCompliant1	UINT2	Non-Compliant cell count 1
ingressCellCount2	UINT4	Ingress Cell Count 2
ingressCellCount1	UINT4	Ingress Cell Count 1
alternateIngressCellCount2	UINT4	Alternate Ingress Cell Counter 2
alternateIngressCellCount1	UINT4	Alternate Ingress Cell Counter 1
remainingFrameCount	UINT2	Remaining Frame count

Ingress PM Record

This record collects together all the fields used to configure the PM support in the Ingress VC table documented in the ATLAS Hardware Specification.

Table 17: Ingress PM Record: sATLS_INGRESS_PM

Field Name	Field Type	Field Description
pmActive2	UINT1	PM2 on/off flag
pmAddr2	UINT1	PM2 Session Address
pmActive1	UINT1	PM1 on/off flag
pmAddr1	UINT1	PM1 Session Address

Ingress Translation Record

This record collects together all the fields used to configure the Translation parameters in the Ingress VC table documented in the ATLAS Hardware Specification.

Table 18 : Ingress Translation Record: sATLS_INGRESS_TRANSLATION

Field Name	Field Type	Field Description
header[ATLS_IVC_HEADER_SIZE]	UINT1	New cell header for Translation
udf	UINT1	User defined field

Field Name	Field Type	Field Description
prePo1	UINT1	Prepend/PostPend 1
prePo2	UINT1	Prepend/PostPend 2
prePo3	UINT1	Prepend/PostPend 3
prePo4	UINT1	Prepend/PostPend 4
prePo5	UINT1	Prepend/PostPend 5
prePo6	UINT1	Prepend/PostPend 6
prePo7	UINT1	Prepend/PostPend 7
prePo8	UINT1	Prepend/PostPend 8
prePo9	UINT1	Prepend/PostPend 9
prePo10	UINT1	Prepend/PostPend 10

Ingress Key Record

This record collects together all the fields used to configure the search Key parameters in the Ingress VC table documented in the ATLAS Hardware Specification.

Table 19: Ingress Key Record: sATLS_INGRESS_KEY

Field Name	Field Type	Field Description
fieldA	UINT2	Field A
phyId	UINT1	PHY ID
fieldB	UINT2	Field B
vpi	UINT2	Virtual Path Identifier
vci	UINT2	Virtual Connection Identifier

Egress Record Type

This record is as documented in the ATLAS hardware specification.

Table 20: Egress Record Type: sATLS_EGRESS_RECORD

Field Name	Field Type	Field Description
fieldA	UINT2	Field A, used as Search Key only, not part of the VCRecord.
fieldB	UINT2	Field B, used as Search Key only, not part of the VCRecord.
active	UINT1	Active Record
pmActive2	UINT1	PM session 1
pmActive1	UINT1	PM session 2
nni	UINT1	network identifier
vpi	UINT2	virtual path identifier
vci	UINT2	virtual channel identifier
reserved	UINT1	Unused
cosEnable	UINT1	Enable Change of State indications
vpcPointer	UINT2	channel pointer
pmAddr2	UINT1	PM session address
pmAddr1	UINT1	PM session address
status	UINT1	Egress VC Status field
internalStatus	UINT2	Egress Internal Status field
oamConfiguration	UINT2	OAM Configuration Field
rxEndToEndAisDefectType	UINT1	End to End AIS defect type
rxSegmentAisDefectType	UINT1	Segment AIS defect type
configuration	UINT2	Egress Configuration Field
phyId	UINT1	PHY ID number
egressCellCount1	UINT4	egress cell count
egressCellCount2	UINT4	egress cell count
alternateEgressCellCount1	UINT4	egress cell alternate count
alternateEgressCellCount2	UINT4	egress cell alternate count
rxEndToEndAisDefectLoc [AIS_DEFECT_LOCATION_SIZE]	UINT1	End to End AIS location
rxSegmentToEndAisDefectLoc [AIS_DEFECT_LOCATION_SIZE]	UINT1	Segment AIS location

Egress OAM Defect Record

This record collects together all the fields used to configure the OAM defect support in the Egress VC table documented in the ATLAS Hardware Specification.

Table 21: Egress OAM Defect Record: sATLS_EGRESS_OAMDEFECT

Field Name	Field Type	Field Description
rxSegmentDefectType	UINT1	Segment AIS Defect type
rxEndToEndDefectType	UINT1	End-to-end AIS Defect type
rxEndToEndDefectLoc[ATLS_IVC_AIS_DEFECT_LOCATION_SIZE]	UINT1	End-to-End AIS Location
rxSegmentAisDefectLoc[ATLS_IVC_AIS_DEFECT_LOCATION_SIZE]	UINT1	Segment AIS Location

Egress Configuration Record

This record collects together all the fields used to configure the VC connection in the Egress VC table documented in the ATLAS Hardware Specification.

Table 22: Egress Configuration Record: sATLS_EGRESS_CONFIG

Field Name	Field Type	Field Description
active	UINT1	Connection activation on/off
nni	UINT1	Network-network Identifier
status	UINT1	Egress VC table status field
configuration	UINT2	Egress VC table configuration field
internalStatus	UINT2	Egress internal status field
cosEnable	UINT1	Change of state on/off

Egress OAM Configuration Record

This record collects together all the fields used to configure the OAM support in the Egress VC table documented in the ATLAS Hardware Specification.

Table 23: Egress OAM Configuration Record: sATLS_EGRESS_OAMCONFIG

Field Name	Field Type	Field Description
oamConfiguration	UINT2	Egress VC table OAM configuration field

Egress Counts Record

This record collects together all the fields used to read the Counts parameters in the Egress VC table documented in the ATLAS Hardware Specification.

Table 24: Egress Counts Record: sATLS_EGRESS_COUNTS

Field Name	Field Type	Field Description
egressCellCount1	UINT4	Egress Cell Count 1
egressCellCount2	UINT4	Egress Cell Count 2
alternateEgressCellCount1	UINT4	Alternate Egress Cell Count 1
alternateEgressCellCount2	UINT4	Alternate Egress Cell Count 2

Egress PM Record

This record collects together all the fields used to configure the PM support in the Egress VC table documented in the ATLAS Hardware Specification.

Table 25: Egress PM Record: sATLS_EGRESS_PM

Field Name	Field Type	Field Description
pmActive2	UINT1	PM2 on/off flag
pmAddr2	UINT1	PM2 Session Address
pmActive1	UINT1	PM1 on/off flag
pmAddr1	UINT1	PM1 Session Address

Egress Key Record

This record collects together all the fields used to configure the Key parameters in the Egress VC table documented in the ATLAS Hardware Specification.

Table 26: Egress Key Record: sATLS_EGRESS_KEY

Field Name	Field Type	Field Description
fieldA	UINT2	Field A
phyId	UINT1	PHY ID
fieldB	UINT2	Field B
vpi	UINT2	Virtual Path Identifier
vci	UINT2	Virtual Connection Identifier

Performance Monitoring

This structure contains the configuration and statistical information for a Performance Monitoring session.

Table 27: Performance Monitoring Record: sATLS_PM_RECORD

Field Name	Field Type	Field Description
configStatus	UINT2	Configuration and status bits for this PM record.
bip16	UINT2	Bit interleaved parity 16
currCellCountCLP0	UINT2	Current count of CLP0 cells
currCellCountCLP0_1	UINT2	Current count of CLP0+1 cells
blerStored	UINT1	Stored Block Error Result
fwdBMCSN	UINT1	Forward BR Monitoring Cell Sequence Number
fwdTRCC0	UINT2	Forward Total Received CLP0 Cell Count
fwdTRCC0_1	UINT2	Forward Total Received CLP0+1 Cell Count
fwdTUC0	UINT2	Forward Total CLP0 User Cell Count
fwdTUC0_1	UINT2	Forward Total CLP0+1 User Cell Count
fwdFMCSN	UINT1	Forward FM Cell Sequence Number
unused1	UINT1	Unused
bwdTRCC0	UINT2	Backward Total Received CLP0 Cell Count
bwdTRCC0_1	UINT2	Backward Total Received CLP0+1 Cell Count

Field Name	Field Type	Field Description
bwdTUC0	UINT2	Backward Total CLP0 User Cell Count
bwdTUC0_1	UINT2	Backward Total CLP0+1 User Cell Count
bwdFMCSN	UINT1	Backward FM Cell Sequence Number
bwdBMCSN	UINT1	Backward BR Monitoring Cell Sequence Number
fwdErrors	UINT1	Forward Error Cell Count
fwdImpaired	UINT1	Forward Impaired Block Count
fwdLostOrImpaired	UINT1	Forward Lost or Impaired Block Count
fwdSECBErrored	UINT1	Forward SECB Errored count
fwdSECBLost	UINT1	Forward SECB Lost count
fwdSECBMisins	UINT1	Forward SECB misinserted count
fwdSECBC	UINT1	Forward SECB Combined count
fwdLostFMCells	UINT1	Forward Lost FM Cell count
fwdTaggedCLP0	UINT2	Forward Tagged CLP0 count
fwdMisinserted	UINT2	Forward Misinserted cell count
fwdLostCLP0	UINT2	Forward Lost CLP0 cell count
fwdLostCLP0_1	UINT2	Forward Lost CLP0+1 cell count
fwdTotalLostCLP0	UINT2	Total Forward Lost CLP0 cells for this PM session
fwdTotalLostCLP0_1	UINT2	Total Forward Lost CLP0+1 cells for this PM session
bwdErrors	UINT1	Backward Error Cell Count
bwdImpaired	UINT1	Backward Impaired Block Count
bwdLostOrImpaired	UINT1	Backward Lost or Impaired Block Count
bwdSECBErrored	UINT1	Backward SECB Errored count
bwdSECBLost	UINT1	Backward SECB Lost count
bwdSECBMisins	UINT1	Backward SECB misinserted count
bwdSECBC	UINT1	Backward SECB Combined count
bwdSECBCAccum	UINT1	Backward Accumulating SECBC count
bwdTaggedCLP0	UINT2	Backward Tagged CLP0 count
bwdMisinserted	UINT2	Backward Misinserted cell count
bwdLostCLP0	UINT2	Backward Lost CLP0 cell count

Field Name	Field Type	Field Description
bwdLostCLP0_1	UINT2	Backward Lost CLP0+1 cell count
bwdTotalLostCLP0	UINT2	Total Backward Lost CLP0 cells for this PM session
bwdTotalLostCLP0_1	UINT2	Total Backward Lost CLP0+1 cells for this PM session
txedCLP0Count	UINT4	Transmitted CLP0 user cell count
txedCLP0_1Count	UINT4	Transmitted CLP0+1 user cell count
bwdLostBRCells	UINT1	Backward Lost BR cells
bwdLostFMCells	UINT1	Backward Lost FM cells

Microprocessor Cell Interface

This structure contains the parameters needed to control cell insertion at the microprocessor interface.

Table 28: Microprocessor Cell Interface Insert Control: sATLS_CELL_INS_CTRL

Field Name	Field Type	Field Description
length	UINT1	Only hexadecimal value between 0x02 and 0x07 are valid and represent cell lengths of 27-32 words respectively.
crc	UINT1	Non-zero: forces generation of CRC10 in the last word of the payload. Zero: last word of the payload is not overwritten.
uphdrx	UINT1	Non-zero – header translation is enabled. Zero – header translation is disabled.
phyId	UINT1	Identifies the output PHY device

Per-PHY Policing

This structure contains the configuration and status information for per-PHY policing.

Table 29: Per-PHY Policing: sATLS_PER_PHY_POLICING

Field Name	FieldType	Field Description

Field Name	FieldType	Field Description
phyTAT	UINT4	Theoretical Arrival Time for the PHY interface
phyI	UINT2	PHY Increment field
phyL	UINT2	PHY Limit field
phyNonCompliant2	UINT2	PHY Non-compliant count 2
phyNonCompliant1	UINT2	PHY Non-compliant count 1
phyVCCount	UINT1	Determines (non-zero value) whether cells which are non-compliant with the per-VC policer are counted in the per-PHY non-compliant cell counts and vice versa
phyPoliceConfig	UINT1	Selects one of four possible police configurations
phyAction	UINT1	Selects one of four possible police actions for non-conforming cells
phyNonCompliant3	UINT2	PHY Non-compliant count 3

Statistics Information

The ATLAS device maintains the following sets of device statistics. These statistics can only be referenced as the full set.

The following VC and PM Statistics are covered in Section 4.

- Aggregate Cells
- Ingress Interface Physical Cells
- Ingress Interface input/output Cells
- Egress Interface input/output Cells
- Ingress Per-PHY Counts [32]
- Egress Per-PHY Counts [32]

Device Statistics information is returned to the application in the following structures.

Table 30: Atlas Statistics: sATLS_AGGR_COUNTS

Field Name	Field Type	Field Description
ingPhysCellCount	UINT1	Physical Layer Cells erroneously arriving at the ATLAS
ingInCellCount	UINT4	Ingress Input Cells

ingOutCellCount	UINT4	Ingress Output Cells
egInCellCount	UINT4	Egress Input Cells
egOutCellCount	UINT4	Egress Output Cells
ingPerPhyCounters[32]	sATLS_I_PERPHY_COUNTERS	32 Per-Phy Ingress Counter sets
egPerPhyCounters[32]	sATLS_E_PERPHY_COUNTERS	32 Per-Phy Egress Counter sets

The following structure describes a single set of Ingress Per–PHY statistics. There will be 32 versions of these.

Table 31: Ingress Per-PHY Statistics: sATLS_I_PERPHY_COUNTERS

Field Name	Field Type	Field Description
ingPhyCLP0Count	UINT4	CLP0 Cell Count
ingPhyCLP1Count	UINT4	CLP1 Cell Count
ingPhyOAMCount	UINT2	Valid OAM Cell Count
ingPhyRMCount	UINT2	Valid RM Cell Count
ingPhyOAMRMErrorCount	UINT2	Errored OAM/RM Cell Count
ingPhyInvVpiVciPtiCount	UINT2	Invalid VPI/VCI/PTI Cell count
ingPhyNonZeroGFCCount	UINT2	Non-Zero GFC Cell Count
ingPhyLastUnknownVpi	UINT2	Last Unknown VPI on this PHY
ingPhyLastUnknownVci	UINT2	Last Unknown VCI on this PHY

The following structure describes a single set of Egress Per–PHY statistics. There will be 32 versions of these.

Table 32: Egress Per-PHY Statistics: sATLS_E_PERPHY_COUNTERS

Field Name	Field Type	Field Description
egPhyCLP0Count	UINT4	CLP0 Cell Count
egPhyCLP1Count	UINT4	CLP1 Cell Count
egPhyOAMCount	UINT2	Valid OAM Cell Count
egPhyRMCount	UINT2	Valid RM Cell Count
egPhyOAMRMErrorCount	UINT2	Errored OAM/RM Cell Count

Field Name	Field Type	Field Description
egPhyInvVciPtiCount	UINT2	Invalid VCI/PTI Cell count

Driver Ingress VC Table Data Structures

The driver maintains its own copy of the information that allows it to perform all access operations on the Ingress VC Table without excessive reads of the Atlas registers. It keeps a mirror copy of:

- The primary table
- The secondary table
- A list of unused secondary table records
- A list of free record addresses
- A list of all the secondary keys

Each of the above items in the list are kept on a per device basis. They are located in the Device Data Block.

These constitute a mirror copy of the Ingress VC Tree structures and the Egress Vector. The contents of each VC record are not maintained.

Ingress VC Primary Table (`ivcPrimaryTable`)

The `ivcPrimaryTable` is an array of secondary root node addresses. The `ivcPrimaryTable` is a mirror copy of the VC Primary Table in the Ingress SRAM. The driver allocates this array when `atlasAdd` is called; it contains `ingressMaxVCs` elements. The root node addresses are 16 bits wide to cater for up to 65536 root nodes. The root node is the top node of a binary tree of secondary keys.

Ingress VC Secondary Table (`ivcSecondaryTable`)

The `ivcSecondaryTable` is an array of `sATLS_NODE_TYPEs`. The `ivcSecondaryTable` is a mirror copy of the secondary table elements of the Ingress VC Table Row0 (ISA[19:16]=0000). The driver allocates this array when `atlasAdd` is called; it contains `ingressMaxVCs` elements. There can be up to 65536 secondary branches. The `sATLS_IVC_NODE_TYPE` is defined below.

Table 33: Ingress Node Type: sATLS_IVC_NODE_TYPE

Field Name	Field Type	Field Description
Selector	UINT1	Selector used to decide direction while traversing the binary tree.

Field Name	Field Type	Field Description
LeftLeaf	UINT1	0: indicates leftBranch identifies a branch 1: indicates leftBranch identifies a leaf
LeftBranch	UINT2	Identifies either a vcRecord or another branch.
RightLeaf	UINT1	0: indicates rightBranch identifies a branch 1: indicates rightBranch identifies a leaf
RightBranch	UINT2	Identifies either a vcRecord or another branch.

Ingress VC Secondary Address List (ivcSecondaryAddrList)

The SRAM addresses for the Secondary Search Table entries can be assigned arbitrarily. The driver implements a stack of available Secondary Search Table entries (ivcSecondaryAddrList) and uses a stack pointer (ivcSecondaryAddrIndex) to point to the next available Secondary Search Table entry. The ivcSecondaryAddrList array contains up to 65, 536 elements. Each element is a 16-bit number that indexes into the ivcSecondaryTable array.

Ingress VC Record Address Free (ivcRecordAddrFree)

The SRAM addresses for the VC Table Records can be assigned arbitrarily. The driver implements an array of Boolean values which identify whether or not a VC Record is in use. The address of the VC Record is used to index into the array. The ivcRecordAddrFree array contains up to 65536 elements.

Ingress VC Secondary Keys (ivcSecondaryKeys)

A search on a secondary search table always yields a leaf. The only exception to this is when the tree is empty. Even though a leaf is found, it may not be the actual leaf being searched for. To verify that the leaf found matches the one sought, the secondary keys of the leaf that was found and the one sought must be compared. The driver maintains a mirror copy of the SRAM secondary keys. The address of the secondary key (which comes from the ivcSecondaryAddrList) is used to index into the array. The ivcSecondaryKeys array contains up to 65536 secondary keys. Each secondary key is a 5-byte array (enough to store the 39-bit secondary key).

User Context

The user context ATLS_USR_CTXT is defined as a void* data type.

5 APPLICATION PROGRAMMING INTERFACE

5.1 Function Calls

This section provides a detailed description of each function that is a member of the ATLAS driver Application Programming Interface.

5.2 Driver Initialization and Shutdown

Initializing Device Drivers: atlasModuleInit

This function performs module level initialization of the device driver. This involves allocating the GDB and initializing the data-structure.

Prototype `INT4 atlasModuleInit(void)`

Inputs None

Outputs None

Returns
```
ATLS_SUCCESS
ATLS_ERR_MEM_ALLOC
ATLS_ERR_MODULE_ALREADY_INIT
```

Shutting Down Drivers: atlasModuleShutdown

This function performs module level shutdown of the driver. This involves deleting all devices controlled by the driver and deallocating the GDD.

Prototype `void atlasModuleShutdown(void)`

Inputs None

Outputs None

Returns None

5.3 Hardware Access

Reading Registers: atlasReadReg

This function can be used to read a register of a specific ATLAS device by providing the register identifier. This function derives the actual address location based on the device handle and register identifier inputs. It then reads the contents of this address location using the system specific function, `sysAtlasRawRead`.

Prototype `INT4 atlasReadReg(ATLAS atlas, UINT2 regId, UINT2 *pVal)`

Valid States `ATLS_PRESENT`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`regId`	: register identifier

Outputs

`pVal`	: register value

Returns

`ATLS_SUCCESS`
`ATLS_ERR_INVALID_DEVICE`
`ATLS_ERR_INVALID_REG_ID`

Writing Registers: atlasWriteReg

This function can be used to write to a register of a specific ATLAS device by providing the register identifier. This function derives the actual address location based on the device handle and register identifier inputs. It then writes the contents of this address location using the system specific function, `sysAtlasRawWrite`.

Prototype `INT4 atlasWriteReg(ATLAS atlas, UINT2 regId, UNIT2 pu2Val)`

Valid Sates `ATLS_PRESENT`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`regId`	: register identifier
`pval`	: value to be written

Outputs None

Returns
```
ATLS_SUCCESS
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_REG_ID
```

5.4 Device addition and deletion

Locating Devices and Allocating Memory: atlasAdd

This function calls `sysAtlasDetectDevice` to locate the device, allocates memory for the device data block (DDB), stores the user's context for the device, and outputs the pointer to the DDB as a handle back to the user. The device handle should be used to identify the device on which the operation is to be performed.

Prototype
```
INT4 atlasAdd( ATLS_USR_CTXT usrCtxt, UINT4 maxIngressVCs, UINT4
maxEgressVCs, sATLS_DDB **patlasHandle)
```

Valid States `ATLS_EMPTY`

Inputs

`usrCtxt`	: user context information
`maxIngressVCs`	: the maximum Number of Ingress VCs
`maxEgressVCs`	: the maximum Number of Egress VCs

Outputs

`patlasHandle`	: device handle (to be used as an argument to most of the ATLAS APIs).

Returns
```
ATLS_SUCCESS,
ATLS_ERR_MOD_NOT_INIT
ATLS_ERR_EXCEED_MAX_DEVS
ATLS_ERR_INVALID_MAX_VCS
ATLS_ERR_DEV_NOT_DETECTED
ATLS_ERR_DEV_ALREADY_ADDED
ATLS_ERR_MEM_ALLOC
```

Side Effects Device is put in the `ATLS_PRESENT` state. A software reset is applied to the device.

Removing Devices: atlasDelete

This function is used to remove the specified device from the list of devices being controlled by the ATLAS driver. Deleting a device involves de-allocating the DDB for that device. It does not deallocate the user's context pointer. The user is responsible for deallocating the user's context. Once deleted, the device handle associated with this device is invalid and cannot be reused.

Prototype `INT4 atlasDelete(ATLAS atlas)`

Valid States `ATLS_PRESENT`

Inputs `atlas` : device handle (from `atlasAdd`)

Outputs None

Returns
`ATLS_SUCCESS`
`ATLS_ERR_INVALID_DEVICE`
`ATLS_ERR_INVALID_STATE`

Side Effects The DDB is deallocated and the device is no longer 'known' to the driver.

5.5 Device Initialization and Reset

Initializing Devices Based on Initialization Vectors: atlasInit

This function initializes the device based on an initialization vector passed by the user. This initialization vector is stored by the driver in the device's DDB. The device registers are then configured accordingly.

Prototype `INT4 atlasInit(ATLAS atlas, sATLS_INIT_VECT *psInitVector)`

Valid States `ATLS_PRESENT`

Inputs
`atlas` : device handle (from `atlasAdd`)
`psInitVector` : pointer to initialization vector that is used by the driver to program the device registers

Outputs None

Returns ATLS_SUCCESS
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_STATE

Side Effects The device is put in ATLS_INIT state.

Resetting Devices: atlasReset

This function applies a software reset to the ATLAS device; it also reinitializes all of the DDB's contents (except for the initialization vector, which is left unmodified). This function is typically called before initializing the device with a new initialization vector.

Prototype INT4 atlasReset(ATLAS atlas)

Valid States ATLS_PRESENT
ATLS_INIT
ATLS_ACTIVE

Inputs atlas : device handle (from atlasAdd)

Outputs None

Returns ATLS_SUCCESS
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE

Side Effects The device is put in the ATLS_PRESENT state. So the device has to be initialized after a reset.

5.6 Device Activation and Deactivation

Activating Devices: atlasActivate

This function activates the ATLAS device by preparing it for normal operation. This involves enabling device interrupts and enabling the Ingress and Egress Cell processing.

Prototypes `INT4 atlasActivate(ATLAS atlas)`

Valid States `ATLS_INIT`

Inputs `atlas` : device handle (from `atlasAdd`)

Outputs None

Returns
`ATLS_SUCCESS`
`ATLS_ERR_INVALID_DEVICE`
`ATLS_ERR_INVALID_STATE`

Side Effects Puts the device in `ATLS_ACTIVE` state.

De-activating Devices: atlasDeactivate

This function de-activates the ATLAS device and removes it from normal operation. This involves disabling device interrupts and disabling the Ingress and Egress Cell processing.

Prototype `INT4 atlasDeactivate(ATLAS atlas)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs `atlas` : device handle (from `atlasAdd`)

Outputs None

Returns
`ATLS_SUCCESS`
`ATLS_ERR_INVALID_DEVICE`
`ATLS_ERR_INVALID_STATE`

Side Effects Puts the device in `ATLS_INIT` state.

5.7 Device Diagnostics

Verifying the Access of Microprocessors: atlasTest

This function verifies the correctness of the microprocessor's access to the device registers by writing to and reading back values; it also verifies access to VC Table SRAM, PM Internal Ram, and Per-PHY Policing RAM.

Prototype `INT4 atlasTest(ATLAS atlas)`

Valid States `ATLS_PRESENT`

Inputs `atlas` : device handle (from `atlasAdd`)

Outputs None

Returns
`ATLS_SUCCESS`
`ATLS_ERR_INVALID_DEVICE`
`ATLS_ERR_SRAM_DIAG`
`ATLS_ERR_IPMRAM_DIAG`
`ATLS_ERR_EPMRAM_DIAG`
`ATLS_ERR_PER_PHY_POLICE_RAM_DIAG`

Side Effects Puts the device in the `ATLS_PRESENT` state after the test. So the device should be re-initialized after calling this function.

5.8 Connection Management

This section defines the functions available for creating and managing connections through the ATLAS device.

These functions access the Ingress/Egress VC Table registers to perform their tasks.

The driver will maintain a shadow copy of the Ingress and Egress VC Table structures.

Adding Ingress Connections: atlasAddIngressConnection

This function adds an ingress connection.

Prototype `INT4 atlasAddIngressConnection(ATLAS atlas, sATLS_INGRESS_RECORD *patlasIngressVCRecord, UINT2*pvcId,UINT1 forceAddr)`

Valid Sates `ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`patlasIngressVCRecord`	: connection information structure
`pvcId`	: pointer to VC Id to use, if forceAddr is non-zero
`forceAddr`	: if zero, the Driver allocates an unused VCid. If non-zero, allocate the VC Id as specified by `pvcId`.

Outputs

`pvcId`:	when `forceAddr` is non-zero, the user must specify the address at which the new VC should appear. When `forceAddr` is zero, this field acts as an output parameter and contains the address of the new VC allocated by the driver.

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_PHYID
ATLS_ERR_INVALID_FIELDA
ATLS_ERR_INVALID_FIELDB
ATLS_ERR_IVC_ADD_VC_EXISTS
ATLS_ERR_IVC_VC_ADDR_RANGE
ATLS_ERR_IVC_VC_ADDR_NONE_FREE
ATLS_ERR_IVC_VC_ADDR_NOT_FREE
ATLS_ERR_PRIMARY_KEY_UNEXPECTED
ATLS_ERR_PRIMARY_ADDR_RANGE
ATLS_ERR_SECONDARY_ADDR_RANGE
```

Deleting Ingress Connections: atlasDeleteIngressConnection

This function deletes an ingress connection.

Prototype `INT4 atlasDeleteIngressConnection(ATLAS atlas, sATLS_INGRESS_RECORD *patlasIngressVCRecord)`

Valid Sates ATLS_ACTIVE

Inputs

atlas	: device handle (from atlasAdd)
patlasIngressVCRecord	: only the fieldA, fieldB, inPhyId are used from this structure during the deletion. The structure is passed for consistency of interface.

Outputs None

Returns
ATLS_SUCCESS
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_PRIMARY_KEY_UNEXPECTED
ATLS_ERR_PRIMARY_ADDR_RANGE
ATLS_ERR_SECONDARY_ADDR_RANGE

Adding Egress Connections: atlasAddEgressConnection

This function adds an egress connection.

Prototype
```
INT4 atlasAddEgressConnection(ATLAS atlas,
sATLS_EGRESS_RECORD*patlasEgressVCRecord, UINT2*pvcId)
```

Valid Sates ATLS_ACTIVE

Inputs

atlas	: device handle (from atlasAdd)
patlasEgressVCRecord	: connection information structure
pvcId	: pointer to VC Id to use, if forceAddr is non-zero

Outputs

pvcId	: when forceAddr is non-zero the user must specify the address at which the new VC should appear. When forceAddr is zero this field acts as an output parameter and contains the address of the new VC allocated by the driver.

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_PHYID
ATLS_ERR_INVALID_FIELDA
ATLS_ERR_INVALID_FIELDB
ATLS_ERR_EVC_RECORD_ADDR_RANGE
ATLS_ERR_EVC_ADD_RECORD_EXISTS
```

Adding Egress Connections: atlasAddEgressDummyConnection

Add the Egress dummy connections required for the cell insertion software workaround. For the software workaround to function correctly, there must be one dummy connection per PHY (the number of PHYs is specified by the Egress Cell Processor Direct Lookup Index Configuration #1).

Prototype

```
INT4 atlasAddEgressDummyConnection(ATLAS atlas,
sATLS_EGRESS_RECORD*patlasEgressVCRecord, UINT2*pvcId)
```

Valid Sates `ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`patlasEgressVCRecord`	: connection information structure
`pvcId`	: pointer to VC Id to use, if forceAddr is non-zero

Outputs

`pvcId`	: contains the address of the new VC allocated by the driver.

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_PHYID
ATLS_ERR_INVALID_FIELDA
ATLS_ERR_INVALID_FIELDB
ATLS_ERR_EVC_RECORD_ADDR_RANGE
ATLS_ERR_EVC_ADD_RECORD_EXISTS
```

Deleting Egress Connections: atlasDeleteEgressConnection

This function deletes an egress connection.

Prototype `INT4 atlasDeleteEgressConnection(ATLAS atlas, sATLS_EGRESS_RECORD *patlasEgressVCRecord)`

Valid States `ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`patlasEgressVCRecord`	: only the fieldA, fieldB, inPhyId, are used from this structure during the deletion. The structure is passed for consistency of interface.

Outputs None

Returns
`ATLS_SUCCESS`
`ATLS_ERR_INVALID_STATE`
`ATLS_ERR_INVALID_DEVICE`
`ATLS_ERR_EVC_RECORD_ADDR_RANGE`
`ATLS_ERR_EVC_ADD_RECORD_NOT_EXIST`

Getting Ingress VC Status: atlasGetIngressVcStatus

This function gets ingress VC status. This function returns the full contents of the following rows of the ingress VC -1,2,3,4,7,8,10,11,12,13, and 14. All other fields are undefined

Prototype `INT4 atlasGetIngressVcStatus (ATLAS atlas, sATLS_INGRESS_RECORD *patlasIngressVCRecord)`

Valid States `ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)

Outputs

`patlasIngressVCRecord`	: the user must set the fieldA, fieldB, phyId, VPI, and VCI before calling this function. The driver finds the VC and fills in the other fields for the user.

Returns

```
ATLS_SUCCESS
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_PHYID
ATLS_ERR_INVALID_FIELDA
ATLS_ERR_INVALID_FIELDB
ATLS_ERR_PRIMARY_KEY_UNEXPECTED
ATLS_ERR_PRIMARY_ADDR_RANGE
ATLS_ERR_SECONDARY_ADDR_RANGE
ATLS_ERR_IVC_VC_NOT_EXIST
```

Getting Ingress VC Counters: atlasGetIngressVcCounts

This function gets ingress VC. This function returns the full contents of the following rows of the ingress VC-5, 6 and 9. All other fields are undefined

Prototype

```
INT4 atlasGetIngressVcCounts (ATLAS atlas, sATLS_INGRESS_RECORD
*patlasIngressVCRecord)
```

Valid States `ATLS_ACTIVE`

Inputs	`atlas`	: device handle (from `atlasAdd`)
Outputs	`patlasIngressVCRecord`	: the user must set the fieldA, fieldB, phyId, VPI, and VCI before calling this function.The driver finds the VC and fills in the other fields for the user.

Returns

```
ATLS_SUCCESS
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_PHYID
ATLS_ERR_INVALID_FIELDA
ATLS_ERR_INVALID_FIELDB
ATLS_ERR_PRIMARY_KEY_UNEXPECTED
ATLS_ERR_PRIMARY_ADDR_RANGE
ATLS_ERR_SECONDARY_ADDR_RANGE
ATLS_ERR_IVC_VC_NOT_EXIST
```

Getting Ingress VC Counters: atlasReadIngressCounts

This function gets ingress VC counters only. This function returns the full contents of the following rows of the ingress VC-5, 6 and 9. All other fields are undefined.

Prototype
```
INT4 atlasReadIngressCounts ( ATLAS atlas,UINT2
u2VcId,UINT1 clearOnRead,sATLS_INGRESS_COUNTS *pCounts)
```

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record
`clearOnRead`	: reset the counts if clearOnRead = 1, otherwise don't clear it.

Outputs

`pcounts`	: connection information structure

Returns
```
ATLS_SUCCESS
ATLS_ERR_MOD_NOT_INIT
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST ATLS_ERR_REG_POLL_TIMED_OUT
```

Enabling Ingress VC : atlasEnableIngressVC

This function enables an existing ingress VC connection. It modifies the Active bit in the configuration field of row 2 in the VC table.

Prototype
```
INT4 atlasEnableIngressVC(ATLAS atlas, UINT2 u2VcId)
```

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.

Outputs None

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
```

Disabling Ingress VC : atlasDisableIngressVC

This function disables an existing ingress VC connection. It modifies the Active bit in the configuration field of row 2 in the VC table.

Prototype `INT4 atlasEnableIngressVC(ATLAS atlas, UINT2 u2VcId)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.

Outputs None

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
```

Getting Full Ingress VC Records: atlasGetIngressVc

This function gets the full ingress VC record. This is the equivalent of getting both the Status and Counts.

Prototype

```
INT4 atlasGetIngressVc (ATLAS atlas, sATLS_INGRESS_RECORD
*patlasIngressVCRecord)
```

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)

Outputs `patlasIngressVCRecord` : the user must set the fieldA, fieldB, phyId, VPI, and VCI before calling this function.The driver finds the VC and fills in the other fields for the user.

Returns
```
ATLS_SUCCESS
ATLS_ERR_INVALID_STATE ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_PHYID
ATLS_ERR_INVALID_FIELDA
ATLS_ERR_INVALID_FIELDB
ATLS_ERR_PRIMARY_KEY_UNEXPECTED
ATLS_ERR_PRIMARY_ADDR_RANGE
ATLS_ERR_SECONDARY_ADDR_RANGE
ATLS_ERR_IVC_VC_NOT_EXIST
```

Getting Egress VC Status: atlasGetEgressVcStatus

This function gets Egress VC status; it returns the full contents of the following rows of the Egress VC-0, 1, 2,3,8,9,10,11,12,13,14 and 15. All other fields are undefined.

Prototype
```
INT4 atlasGetEgressVcStatus (ATLAS atlas,sATLS_EGRESS_RECORD
*patlasEgressVCRecord)
```

Valid States `ATLS_ACTIVE`

Inputs `atlas` : device handle (from `atlasAdd`)

Outputs `patlasEgressVCRecord` : the user must set the fieldA, fieldB, and phyId before calling this function. The driver finds the VC and fills in the other fields for the user.

Returns
```
ATLS_SUCCESS
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_PHYID
ATLS_ERR_INVALID_FIELDA
ATLS_ERR_INVALID_FIELDB
ATLS_ERR_EVC_RECORD_ADDR_RANGE
ATLS_ERR_EVC_RECORD_NOT_EXIST
```

Getting Ingress VC Counters: atlasGetEgressVcCounts

This function gets Egress VC Counters. This function returns the full contents of the following rows of the Egress VC-4,5,6, and 7. All other fields are undefined.

Prototype `INT4 atlasGetEgressVcStatus (ATLAS atlas, sATLS_EGRESS_RECORD *patlasEgressVCRecord)`

Valid States `ATLS_ACTIVE`

Inputs `atlas` : device handle (from `atlasAdd`)

Outputs `patlasEgressVCRecord` : the user must set the fieldA, fieldB, and phyId before calling this function. The driver finds the VC and fills in the other fields for the user.

Returns
`ATLS_SUCCESS`
`ATLS_ERR_INVALID_STATE`
`ATLS_ERR_INVALID_DEVICE`
`ATLS_ERR_INVALID_PHYID`
`ATLS_ERR_INVALID_FIELDA`
`ATLS_ERR_INVALID_FIELDB`
`ATLS_ERR_EVC_RECORD_ADDR_RANGE`
`ATLS_ERR_EVC_RECORD_NOT_EXIST`

Getting Egress VC Counters: atlasReadEgressCounts

This function gets Egress VC Counters. This function returns the full contents of the following rows of the Egress VC-4,5,6, and 7. All other fields are undefined

Prototype `INT4 atlasReadEgressVC(ATLAS atlas, UINT2 u2VcId, UINT1 clearOnRead, sATLS_EGRESS_COUNTS *pCounts)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs	`atlas`	: device handle (from `atlasAdd`)
	`u2VcId`	: index to the VC table record.
	`clearOnRead`	: Reset the counts if clearOnRead = 1, otherwise don't clear it.
Outputs	`pCounts`	: connection information structure.

Returns	`ATLS_SUCCESS,`
	`ATLS_ERR_INVALID_STATE`
	`ATLS_ERR_INVALID_DEVICE`
	`ATLS_ERR_IVC_VC_NOT_EXIST`
	`ATLS_ERR_REG_POLL_TIMED_OUT`
	`ATLS_ERR_MOD_NOT_INIT`	

Enabling Egress VC : atlasEnableEgressVC

This function enables an existing Egress VC connection. It modifies the Active bit of row 2 in the Egress VC table.

Prototype `INT4 atlasEnableEgressVC(ATLAS atlas, UINT2 u2VcId)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs	`atlas`	: device handle (from `atlasAdd`)
	`u2VcId`	: index to the VC table record.
Outputs	None	

Returns	`ATLS_SUCCESS,`
	`ATLS_ERR_INVALID_STATE`
	`ATLS_ERR_INVALID_DEVICE`
	`ATLS_ERR_IVC_VC_NOT_EXIST`
	`ATLS_ERR_REG_POLL_TIMED_OUT`	

Disabling Egress VC : atlasDisableEgressVC

This function disables an existing egress VC connection. It modifies the Active bit of row 2 in the Egress VC table.

Prototype `INT4 atlasEnableEgressVC(ATLAS atlas, UINT2 u2VcId)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.

Outputs None

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
```

Getting Full Egress VC Records: atlasGetEgressVc

This function gets the full Egress VC record. This is the equivalent of getting both the Status and Counts.

Prototype `INT4 atlasGetEgressVcStatus (ATLAS atlas, sATLS_EGRESS_RECORD *patlasEgressVCRecord)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)

Outputs

`patlasEgressVCRecord`	: the user must set the fieldA, fieldB, and phyId before calling this function. The driver finds the VC and fills in the other fields for the user.

Returns

```
ATLS_SUCCESS
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_PHYID
ATLS_ERR_INVALID_FIELDA
ATLS_ERR_INVALID_FIELDB
ATLS_ERR_EVC_RECORD_ADDR_RANGE
ATLS_ERR_EVC_RECORD_NOT_EXIST
```

Getting Ingress VC Key: atlasReadIngressKey

This function gets the Ingress VC Key parameters; it returns the following VC ingress table fields:

Row 1 : fieldA, fieldB, phyId, vci, vpi.

Prototype

```
INT4 atlasReadIngressKey(ATLAS atlas, UINT2 u2VcId, sATLS_INGRESS_KEY *pKey)
```

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.

Outputs

`pKey`	: translation information structure.

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT
```

Modifying Ingress VC Key: atlasWriteIngressKey

This function modifies the Ingress VC Key parameters; the following VC ingress table fields are updated:

Row 1 : fieldB, phyId, vci, vpi.

Prototype

```
INT4 atlasWriteIngressKey(ATLAS atlas, UINT2 u2VcId,
sATLS_INGRESS_KEY *pKey)
```

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.

Outputs

`pKey`	: translation information structure.

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT
```

Getting Ingress VC Configuration: atlasReadIngressConfig

This function gets Ingress VC Configuration. This function returns the following VC ingress table fields :

Row 1 : NNI
Row 2 : status, internal Status, configuration
Row 4 : gfr
Row 5 : gfr state
Row 10 : maximum Frame Length

Prototype

```
INT4 atlasReadIngressConfig(ATLAS atlas, UINT2 u2VcId,
sATLS_INGRESS_CONFIG *pConfig)
```

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.

Outputs

`pConfig`	: configuration information structure.

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT
```

Modifying Ingress VC configuration: atlasWriteIngressConfig

This function modifies the Ingress VC Configuration of an existing connection; the following VC ingress table fields are updated :

Row 1 : NNI
Row 2 : status, internal Status, configuration
Row 4 : gfr
Row 5 : gfr state
Row 10 : maximum Frame Length.

Prototype	`INT4 atlasWriteIngressConfig(ATLAS atlas, UINT2 u2VcId, sATLS_INGRESS_CONFIG *pConfig)`	
Valid States	`ATLS_INIT, ATLS_ACTIVE`	
Inputs	`atlas`	: device handle (from `atlasAdd`)
	`u2VcId`	: index to the VC table record.
	`pConfig`	: configuration information structure.
Outputs	None	

Returns	`ATLS_SUCCESS,`
	`ATLS_ERR_INVALID_STATE`
	`ATLS_ERR_INVALID_DEVICE`
	`ATLS_ERR_IVC_VC_NOT_EXIST`
	`ATLS_ERR_REG_POLL_TIMED_OUT`
	`ATLS_ERR_MOD_NOT_INIT`	

Modifying Connection Configurations: atlasModifyIngressVcConfiguration

This function modifies the configuration of an existing connection. Only configuration information is updated. All other fields are ignored.

Confguration information is contained in:

Row 1: F4toF5AIS, PMActive2, Pmaddr2, PNActive1, Pmaddr1, NNI
Row 2: Configuration, OAM Configuration, and VPC Pointer
Row 10: MaximumFramelength

Prototype `INT4 atlasModifyIngressVcConfiguration (ATLAS atlas, sATLS_INGRESS_RECORD *patlasIngressVCRecord)`

Valid States `ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`patlasIngressVCRecord`	: connection information structure

Outputs `None`

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_PHYID
ATLS_ERR_INVALID_FIELDA
ATLS_ERR_INVALID_FIELDB
ATLS_ERR_PRIMARY_KEY_UNEXPECTED
ATLS_ERR_PRIMARY_ADDR_RANGE
ATLS_ERR_SECONDARY_ADDR_RANGE
ATLS_ERR_IVC_VC_NOT_EXIST
```

Getting Ingress VC Policing Configuration: atlasReadIngressPolicing

This function gets Ingress VC Policing configuration. This function returns the following VC ingress table fields :

Row 1 : CLP CC enable
Row 3 : Cocup, tat2, tat1
Row 4 : action2, action1, i2, l2, i1, l1
Row 5 : phy Police, violate.

Prototype `INT4 atlasReadIngressPolicing(ATLAS atlas, UINT2 u2VcId, sATLS_INGRESS_POLICING *pPolicing)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.

Outputs

`pPolicing`	: policing information structure.

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT
```

Modifying Ingress VC Policing Configuration: atlasWriteIngressPolicing

This function modifies Ingress VC Policing configuration; the following VC ingress table fields are updated :

Row 1 : CLP CC enable
Row 3 : Cocup, tat2, tat1
Row 4 : action2, action1, i2, l2, i1, l1
Row 5 : phy Police, violate.

Prototype

```
INT4 atlasWriteIngressPolicing(ATLAS atlas, UINT2 u2VcId,
sATLS_INGRESS_POLICING *pPolicing)
```

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.
pPolicing	: policing information structure.

Outputs None

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT
```

Modifying Policing Parameters: atlasModifyIngressVcPolicing

This function modifies the policing parameters of an existing connection. Only policing parameters are updated. All other fields are ignored.

Policing information is contained in:

Row 1: cLPccEn
Row 3: Cocup
Row 4: PoliceConfiguration, Actio2, Action1
Row 5: PhyPolice, violate

Prototype `INT4 atlasModifyIngressVcPolicing (ATLAS atlas, sATLS_INGRESS_RECORD *patlasIngressVCRecord)`

Valid States `ATLS_ACTIVE`

Inputs
`atlas` : device handle (from `atlasAdd`)
`patlasIngressVCRecord` : connection information structure

Returns
`ATLS_SUCCESS,`
`ATLS_ERR_INVALID_STATE`
`ATLS_ERR_INVALID_DEVICE`
`ATLS_ERR_INVALID_PHYID`
`ATLS_ERR_INVALID_FIELDA`
`ATLS_ERR_INVALID_FIELDB`
`ATLS_ERR_PRIMARY_KEY_UNEXPECTED`
`ATLS_ERR_PRIMARY_ADDR_RANGE`
`ATLS_ERR_SECONDARY_ADDR_RANGE`
`ATLS_ERR_IVC_VC_NOT_EXIST`

Getting Ingress VC Translation Configuration: atlasReadIngressTranslation

This function gets the Ingress VC translation configuration. This function returns the following VC ingress table fields :

Row 7 : Header, UDF, prepo1, prepo2
Row 8 : prepo3 to prepo10.

Prototype `INT4 atlasReadIngressTranslation(ATLAS atlas, UINT2 u2VcId, sATLS_INGRESS_TRANSLATION *pTranslation)`

Valid States `ATLS_INIT,ATLS_ACTIVE`

Inputs
`atlas` : device handle (from `atlasAdd`)
`u2VcId` : index to the VC table record.

Outputs
`pTranslation` : translation information structure.

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT
```

Modifying Ingress VC Translation Configuration: atlasWriteIngressTranslation

This function modifies the Ingress VC translation configuration; the following VC ingress table fields are updated :

Row 7 : Header, UDF, prepo1, prepo2
Row 8 : prepo3 to prepo10.

Prototype

```
INT4 atlasWriteIngressTranslation(ATLAS atlas, UINT2 u2VcId,
sATLS_INGRESS_TRANSLATION *pTranslation)
```

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.
`pTranslation`	: translation information structure.

Outputs None

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT
```

Modifying Translation Parameters: atlasModifyIngressVcTranslation

This function modifies the Translation parameters of an existing connection. Only translation parameters are updated. All other fields are ignored.

Row 7: Header, udf, PrePo1, prePo2
Row 8: PrePo3, prePo4, PrePo5, prePo6, PrePo7, prePo8, PrePo9, prePo10

Prototype `INT4 atlasModifyIngressVcTranslation (ATLAS atlas,sATLS_INGRESS_RECORD *patlasIngressVCRecord)`

Valid States `ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`patlasIngressVCRecord`	: connection information structure

Returns

`ATLS_SUCCESS,`
`ATLS_ERR_INVALID_STATE`
`ATLS_ERR_INVALID_DEVICE`
`ATLS_ERR_INVALID_PHYID`
`ATLS_ERR_INVALID_FIELDA`
`ATLS_ERR_INVALID_FIELDB`
`ATLS_ERR_PRIMARY_KEY_UNEXPECTED`
`ATLS_ERR_PRIMARY_ADDR_RANGE`
`ATLS_ERR_SECONDARY_ADDR_RANGE`
`ATLS_ERR_IVC_VC_NOT_EXIST`

Modifying Configurations, Policings, and Translations: atlasModifyIngressVcConfiguration

This is equivalent to issuing a Modify Configuration, Policing, and Translation. All fields that would be updated by those functions will be updated by this function.

Prototype `INT4 atlasModifyIngressVcConfiguration (ATLAS atlas,sATLS_INGRESS_RECORD *patlasIngressVCRecord)`

Valid States `ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`patlasIngressVCRecord`	: connection information structure

Returns

`ATLS_SUCCESS,`
`ATLS_ERR_INVALID_STATE`
`ATLS_ERR_INVALID_DEVICE`
`ATLS_ERR_INVALID_PHYID`
`ATLS_ERR_INVALID_FIELDA`
`ATLS_ERR_INVALID_FIELDB`
`ATLS_ERR_PRIMARY_KEY_UNEXPECTED`
`ATLS_ERR_PRIMARY_ADDR_RANGE`
`ATLS_ERR_SECONDARY_ADDR_RANGE`
`ATLS_ERR_IVC_VC_NOT_EXIST`

Getting Ingress VC OAM Configuration: atlasReadIngressOAMConfig

This function gets the Ingress VC OAM configuration. This function returns the following VC ingress table fields :

Row 1 : f4ToF5Ais.
Row 2 : OAM configuration, vpcPointer

Prototype `INT4 atlasReadIngressOAMConfig(ATLAS atlas, UINT2 u2VcId, sATLS_INGRESS_OAMCONFIG *pOamConfig)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.

Outputs `pOamConfig` : OAM config information structure.

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT
```

Modifying Ingress VC OAM Configuration: atlasWriteIngressOAMConfig

This function modifies the Ingress VC OAM configuration; the following VC ingress table fields are updated:

Row 1 : f4ToF5Ais
Row 2 : OAM configuration, vpcPointer

Prototype `INT4 atlasWriteIngressOAMConfig(ATLAS atlas, UINT2 u2VcId, sATLS_INGRESS_OAMCONFIG *pOamConfig)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.
`pOamConfig`	: OAM config information structure.

Outputs None

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT
```

Modifying Ingress VC VPC Pointer: atlasWriteIngressVpcPointer

This function modifies the Ingress VC VPC pointer field in row 2.

Prototype

```
INT4 atlasWriteIngressVpcPointer(ATLAS atlas, UINT2 u2VcId,
sATLS_INGRESS_OAMCONFIG *pOamConfig)
```

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.
`pOamConfig`	: OAM config information structure.

Outputs None

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT
```

Getting Ingress VC OAM Defect: atlasReadIngressOAMDefect

This function gets the Ingress VC OAM Defect configuration; it returns the contents of row 11 to row 14.

Prototype `INT4 atlasReadIngressOAMDefect(ATLAS atlas, UINT2 u2VcId, sATLS_INGRESS_OAMDEFECT *pOamDefect)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.

Outputs

`pOamDefect`	: OAM defect config information structure

Returns

`ATLS_SUCCESS,`
`ATLS_ERR_INVALID_STATE`
`ATLS_ERR_INVALID_DEVICE`
`ATLS_ERR_IVC_VC_NOT_EXIST`
`ATLS_ERR_REG_POLL_TIMED_OUT`
`ATLS_ERR_MOD_NOT_INIT`

Activating/Deactivating Ingress VC CC support: atlasWriteIngressCCActivation

This function modifies the Ingress VC CC OAM support configuration. The contents of OAM configuration field in row 2 is updated.

Prototype `INT4 atlasWriteIngressCCActivation(ATLAS atlas, UINT2 u2VcId, UINT1 segment, UINT1 endToEnd)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs	`atlas`	: device handle (from `atlasAdd`)
	`u2VcId`	: index to the VC table record.
	`segment`	: Segment activation flag (1 = activate 0 = deactivate).
	endToEnd	: Segment activation flag (1 = activate, 0 = deactivate).
Outputs	None	

Returns	`ATLS_SUCCESS,`
	`ATLS_ERR_INVALID_STATE`
	`ATLS_ERR_INVALID_DEVICE`
	`ATLS_ERR_IVC_VC_NOT_EXIST`
	`ATLS_ERR_REG_POLL_TIMED_OUT`
	`ATLS_ERR_MOD_NOT_INIT`	

Activating/Deactivating Ingress VC AIS support: atlasWriteIngressAISActivation

This function modifies the Ingress VC AIS OAM support configuration. The contents of OAM configuration feld in row 2 is updated.

Prototype	`INT4 atlasWriteIngressAISActivation(ATLAS atlas, UINT2 u2VcId, UINT1 segment, UINT1 endToEnd)`	
Valid States	`ATLS_INIT, ATLS_ACTIVE`	
Inputs	`atlas`	: device handle (from `atlasAdd`)
	`u2VcId`	: index to the VC table record.
	`segment`	: Segment activation flag (1 = activate 0 = deactivate).
	`endToEnd`	: Segment activation flag (1 = activate, 0 = deactivate).
Outputs	None	

Returns	`ATLS_SUCCESS,`
	`ATLS_ERR_INVALID_STATE`
	`ATLS_ERR_INVALID_DEVICE`
	`ATLS_ERR_IVC_VC_NOT_EXIST`
	`ATLS_ERR_REG_POLL_TIMED_OUT`
	`ATLS_ERR_MOD_NOT_INIT`	

Activating/Deactivating Ingress VC RDI support: atlasWriteIngressRDIActivation

This function modifies the Ingress VC RDI OAM support configuration. The contents of OAM configuration field in row 2 is updated.

Prototype

```
INT4 atlasWriteIngressRDIActivation(ATLAS atlas, UINT2 u2VcId,
UINT1 segment, UINT1 endToEnd)
```

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.
`segment`	: Segment activation flag (1 = activate 0 = deactivate).
`endToEnd`	: Segment activation flag (1 = activate, 0 = deactivate).

Outputs None

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT
```

Modifying Existing Connections: atlasModifyEgressVc

This function modifies an existing connection. Only configuration information is updated. All other fields are ignored.

Row 0: Active, pmActive2, pmactive, nni
Row 1: CosEnable, vpcPointer, pmAddr2, pmAddr1
Row 2: OamConfiguration
Row 3: Configuration

Prototype

```
INT4 atlasModifyEgressVcConfiguration (ATLAS atlas,
sATLS_EGRESS_RECORD *patlasEgressVCRecord)
```

Valid States ATLS_ACTIVE

Inputs

atlas	: device handle (from atlasAdd)
patlasEgressVCRecord	: connection information structure

Returns

ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_PHYID
ATLS_ERR_INVALID_FIELDA
ATLS_ERR_INVALID_FIELDB
ATLS_ERR_EVC_RECORD_ADDR_RANGE
ATLS_ERR_EVC_RECORD_NOT_EXIST

Modifying PHY Id Fields: atlasModifyEgressVcPhyId

This function modifies just the PHY Id field of the VC Record.

Prototype INT4 atlasModifyEgressVcConfiguration (ATLAS atlas,sATLS_EGRESS_RECORD *patlasEgressVCRecord, UINT1 newPhyId)

Valid States ATLS_ACTIVE

Inputs

atlas	: device handle (from atlasAdd)
patlasEgressVCRecord	: connection information structure newPhyId: new PHY ID

Returns

ATLS_SUCCESS
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_PHYID
ATLS_ERR_INVALID_FIELDA
ATLS_ERR_INVALID_FIELDB
ATLS_ERR_EVC_RECORD_ADDR_RANGE
ATLS_ERR_EVC_RECORD_NOT_EXIST

Modifying VP/VCI Fields: atlasModifyEgressVcVpiVci

This function modifies just the VPI and VCI field of the VC Record.

Prototype `INT4 atlasModifyEgressVcConfiguration (ATLAS atlas,sATLS_EGRESS_RECORD *patlasEgressVCRecord, UINT2 newVpi, UINT2 newVci)`

Valid States `ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`patlasEgressVCRecord`	: connection information structure
`newVci`	: new VCI
`newVpi`	: new VPI

Returns

`ATLS_SUCCESS`
`ATLS_ERR_INVALID_STATE`
`ATLS_ERR_INVALID_DEVICE`
`ATLS_ERR_INVALID_PHYID`
`ATLS_ERR_INVALID_FIELDA`
`ATLS_ERR_INVALID_FIELDB`
`ATLS_ERR_EVC_RECORD_ADDR_RANGE`
`ATLS_ERR_EVC_RECORD_NOT_EXIST`

Getting Egress VC Configuration: atlasReadEgressConfig

This function gets Egress VC Configuration. This function returns the following VC egress table fields:

Row 0 : Active, NNI
Row 1 : Cos enable.
Row 2 : status, internal Status
Row 3 : configuration

Prototype `INT4 atlasReadEgressConfig(ATLAS atlas, UINT2 u2VcId, sATLS_EGRESS_CONFIG *pConfig)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.

Outputs

`pConfig`	: configuration information structure.

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT
```

Modifying Egress VC Configuration: atlasWriteEgressConfig

This function modifies the Egress VC Configuration of an existing connection; the following VC Egress table fields are updated:

Row 0 : Active, NNI
Row 1 : Cos enable
Row 2 : status, internal Status
Row 3 : configuration

Prototype

```
INT4 atlasWriteEgressConfig(ATLAS atlas, UINT2 u2VcId,
sATLS_EGRESS_CONFIG *pConfig)
```

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.
`pConfig`	: configuration information structure.

Outputs None

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT
```

Getting Egress VC OAM Configuration: atlasReadEgressOAMConfig

This function gets the Egress VC OAM configuration. This function returns the following VC egress table field :

Row 2 : OAM configuration.

Prototype `INT4 atlasReadEgressOAMConfig(ATLAS atlas, UINT2 u2VcId, sATLS_EGRESS_OAMCONFIG *pOamConfig)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.

Outputs

`pOamConfig`	: OAM config information structure.

Returns

`ATLS_SUCCESS,`
`ATLS_ERR_INVALID_STATE`
`ATLS_ERR_INVALID_DEVICE`
`ATLS_ERR_IVC_VC_NOT_EXIST`
`ATLS_ERR_REG_POLL_TIMED_OUT`
`ATLS_ERR_MOD_NOT_INIT`

Modifying Egress VC OAM Configuration: atlasWriteEgressOAMConfig

This function modifies the Egress VC OAM configuration; the following VC egress table field is updated:

Row 2 : OAM configuration

Prototype `INT4 atlasWriteEgressOAMConfig(ATLAS atlas, UINT2 u2VcId, sATLS_EGRESS_OAMCONFIG *pOamConfig)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.
`pOamConfig`	: OAM config information structure.

Outputs None

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT
```

Getting Egress VC OAM Defect: atlasReadEgressOAMDefect

This function gets the Egress VC OAM Defect configuration; it returns The contents of row 8 to row 15.

Prototype

```
INT4 atlasReadEgressOAMDefect(ATLAS atlas, UINT2 u2VcId,
sATLS_EGRESS_OAMDEFECT *pOamDefect)
```

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.

Outputs

`pOamDefect`	: OAM defect config information structure

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT
```

Modifying Egress VC OAM Defect: atlasWriteEgressOAMDefect

This function modifies the Egress VC OAM Defect configuration. The contents of row 8 to row 15 is updated.

Prototype

```
INT4 atlasWriteEgressOAMDefect(ATLAS atlas, UINT2 u2VcId,
sATLS_EGRESS_OAMDEFECT *pOamDefect)
```

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.
`pOamDefect`	: OAM defect config information structure

Outputs None

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT
```

Modifying Egress VC VPC Pointer: atlasWriteEgressVpcPointer

This function modifies the Egress VC VPC pointer field in row 1.

Prototype

```
INT4 atlasWriteEgressVpcPointer(ATLAS atlas, UINT2 u2VcId, UINT2 u2Vpc)
```

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.
`u2Vpc`	: VPC pointer value to be written in the egress VC table.

Outputs None

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT
```

Activating/Deactivating Egress VC CC support: atlasWriteEgressCCActivation

This function modifies the Egress VC CC OAM support configuration. The contents of OAM configuration in row 2 is updated.

Prototype `INT4 atlasWriteEgressCCActivation(ATLAS atlas, UINT2 u2VcId, UINT1 segment, UINT1 endToEnd)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.
`segment`	: Segment activation flag (1 = activate 0 = deactivate).
`endToEnd`	: Segment activation flag (1 = activate, 0 = deactivate).

Outputs None

Returns
`ATLS_SUCCESS,`
`ATLS_ERR_INVALID_STATE`
`ATLS_ERR_INVALID_DEVICE`
`ATLS_ERR_IVC_VC_NOT_EXIST`
`ATLS_ERR_REG_POLL_TIMED_OUT`
`ATLS_ERR_MOD_NOT_INIT`

Activating/Deactivating Egress VC AIS support: atlasWriteEgressAISActivation

This function modifies the Egress VC AIS OAM support configuration. The contents of OAM configuration field in rowe 2 is updated.

Prototype `INT4 atlasWriteEgressAISActivation(ATLAS atlas, UINT2 u2VcId, UINT1 segment, UINT1 endToEnd)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs	`atlas`	: device handle (from `atlasAdd`)
	`u2VcId`	: index to the VC table record.
	`segment`	: Segment activation flag (1 = activate 0 = deactivate).
	`endToEnd`	: Segment activation flag (1 = activate, 0 = deactivate).

Outputs None

Returns
```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT
```

Activating/Deactivating Egress VC RDI support: atlasWriteEgressRDIActivation

This function modifies the Egress VC RDI OAM support configuration. The contents of OAM configuration field in row 2 is updated.

Prototype
```
INT4 atlasWriteEgressRDIActivation(ATLAS atlas, UINT2 u2VcId, UINT1 segment, UINT1 endToEnd)
```

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs	`atlas`	: device handle (from `atlasAdd`)
	`u2VcId`	: index to the VC table record.
	`segment`	: Segment activation flag (1 = activate 0 = deactivate).
	`endToEnd`	: Segment activation flag (1 = activate, 0 = deactivate).

Outputs None

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT
```

5.9 Performance monitoring sessions

This section defines functions that can be used to setup and activate PM sessions.

PM Session Allocation

Performance Monitoring dynamically allocates PM Session Ids. To do this, it maintains a list of free PM Session Ids, one for each of the four Banks of PM Sessions. The driver implements four arrays of Boolean values that identify whether or not a PM Session is free. The PM Session Id is used to index into the array. The In/EgressBank1/2PMFree arrays contain 128 elements. The driver also maintains the current number of free PM Sessions and an index to the next free PM Session Id.

Allocating Free PM Record Ids: atlasGetIngressBank1PMId

This function allocates a free PM Record Id from the Ingress-Bank-1 free list.

Prototype `INT4 atlasGetIngressBank1PMId (ATLAS atlas,UINT1 *pmSessionId)`

Valid States `ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)

Outputs

`pmSessionId`	: address of where to put the allocated PM Session Id

Returns

```
ATLS_SUCCESS
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_PM_NOT_FREE
```

Allocating Free PM Record Ids: atlasGetIngressBank2PMId

This function allocates a free PM Record Id from the Ingress-Bank-2 free list.

Prototype `INT4 atlasGetIngressBank2PMId (ATLAS atlas,UINT1 *pmSessionId)`

Valid States `ATLS_ACTIVE`

Inputs `atlas` : device handle (from `atlasAdd`)

Outputs `pmSessionId` : address of where to put the allocated PM Session Id

Returns
`ATLS_SUCCESS`
`ATLS_ERR_INVALID_STATE`
`ATLS_ERR_INVALID_DEVICE`
`ATLS_ERR_PM_NOT_FREE`

Allocating Free PM Record Ids: atlasGetEgressBank1PMId

This function allocates a free PM Record Id from the Egress-Bank-1 free list.

Prototype `INT4 atlasGetEgressBank1PMId (ATLAS atlas,UINT1 *pmSessionId)`

Valid States `ATLS_ACTIVE`

Inputs `atlas` : device handle (from `atlasAdd`)

Outputs `pmSessionId` : address of where to put the allocated PM Session Id

Returns
`ATLS_SUCCESS`
`ATLS_ERR_INVALID_STATE`
`ATLS_ERR_INVALID_DEVICE`
`ATLS_ERR_PM_NOT_FREE`

Allocating Free PM Record Ids: atlasGetEgressBank2PMId

This function allocates a free PM Record Id from the Egress-Bank-2 free list.

Prototype `INT4 atlasGetEgressBank2PMId (ATLAS atlas,UINT1 *pmSessionId)`

Valid States `ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)

Outputs

`pmSessionId`	: address of where to put the allocated PM Session Id

Returns

`ATLS_SUCCESS`
`ATLS_ERR_INVALID_STATE`
`ATLS_ERR_INVALID_DEVICE`
`ATLS_ERR_PM_NOT_FREE`

Freeing PM Record Ids: atlasFreeIngressBank1PMId

This function frees a PM Record Id back to the Ingress-Bank-1 free list. There is no validation that this session Id is not in use, nor that it is not already on the free list.

Prototype `INT4 atlasFreeIngressBank1PMId (ATLAS atlas,UINT1 pmSessionId)`

Valid States `ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`pmSessionId`	: Id of PM session to free

Outputs None

Returns

`ATLS_SUCCESS`
`ATLS_ERR_INVALID_STATE`
`ATLS_ERR_INVALID_DEVICE`
`ATLS_ERR_PM_SESSION_INDEX_RANGE`

Freeing PM Record Ids: atlasFreeIngressBank2PMId

This function frees a PM Record Id back to the Ingress-Bank-2 free list. There is no validation that this session Id is not in use, nor that it is not already on the free list.

Prototype `INT4 atlasFreeIngressBank2PMId (ATLAS atlas,UINT1 pmSessionId)`

Valid States `ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`pmSessionId`	: Id of PM session to free

Outputs None

Returns

```
ATLS_SUCCESS
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_PM_SESSION_INDEX_RANGE
```

Freeing PM Record Ids: atlasFreeEgressBank1PMId

This function frees a PM Record Id back to the Egress-Bank-1 free list. There is no validation that this session Id is not in use, nor that it is not already on the free list.

Prototype `INT4 atlasFreeEgressBank2PMId (ATLAS atlas,UINT1 pmSessionId)`

Valid States `ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`pmSessionId`	: Id of PM session to free

Outputs None

Returns

```
ATLS_SUCCESS
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_PM_SESSION_INDEX_RANGE
```

Freeing PM Record Ids: atlasFreeEgressBank2PMId

This function frees a PM Record Id back to the Egress-Bank-2 free list. There is no validation that this session Id is not in use, nor that it is not already on the free list.

Prototype `INT4 atlasFreeEgressBank2PMId (ATLAS atlas,UINT1 pmSessionId)`

Valid States `ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`pmSessionId`	: Id of PM session to free

Outputs None

Returns

```
ATLS_SUCCESS
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_PM_SESSION_INDEX_RANGE
```

Writing PM Records: atlasWriteIngressBank1PMConfig

This function writes a PM record into the Internal Performance Monitoring Table.

Prototype `INT4 atlasWriteIngressBank1PMConfig (ATLAS atlas,UINT1 pmSessionId,UINT2 pmConfig)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`pmSessionId`	: determines session Id to access (0-127)
`pmConfig`	: new PM configuration word

Outputs None

Returns

```
ATLS_SUCCESS
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_PM_SESSION_ID
```

Writing PM Records: atlasWriteIngressBank2PMConfig

This function writes a PM record into the Internal Performance Monitoring Table.

Prototype `INT4 atlasWriteIngressBank2PMConfig (ATLAS atlas,UINT1 pmSessionId,UINT2 pmConfig)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`pmSessionId`	: determines session Id to access (0-127)
`pmConfig`	: new PM configuration word

Outputs None

Returns

`ATLS_SUCCESS`
`ATLS_ERR_INVALID_STATE`
`ATLS_ERR_INVALID_DEVICE`
`ATLS_ERR_INVALID_PM_SESSION_ID`

Writing PM Records: atlasWriteEgressBank1PMConfig

This function writes a PM record into the Internal Performance Monitoring Table.

Prototype `INT4 atlasWriteEgressBank1PMConfig (ATLAS atlas,UINT1 pmSessionId,UINT2 pmConfig)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`pmSessionId`	: determines session Id to access (0-127)
`pmConfig`	: new PM configuration word

Outputs None

Returns
```
ATLS_SUCCESS
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_PM_SESSION_ID
```

Writing PM Records: atlasWriteEgressBank2PMConfig

This function writes a PM record into the Internal Performance Monitoring Table.

Prototype INT4 atlasWriteEgressBank2PMConfig (ATLAS `atlas`,UINT1 pmSessionId,UINT2 pmConfig)

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`pmSessionId`	: determines session Id to access (0-127)
`pmConfig`	: new PM configuration word

Outputs None

Returns
```
ATLS_SUCCESS
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_PM_SESSION_ID
```

Reading PM Records: atlasReadIngressBank1PMRecord

This function reads a PM record from the Internal Ingress Bank 1 Performance Monitoring Table.

Prototype
```
INT4 atlasReadIngressBank1PMRecord ( ATLAS atlas,UINT1 pmSessionId,
UINT1 clearOnRead, sATLS_PM_RECORD *pmRecord)
```

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`clearOnRead`	: 0 – registers not cleared upon read : 1 – registers cleared upon read
`pmSessionId`	: determines session Id to access (0-127),

Outputs

`pmRecord`	: pointer to pmRecord to be filled in.

Returns

```
ATLS_SUCCESS
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_PM_SESSION_ID
```

Reading PM Records: atlasReadIngressBank2PMRecord

This function reads a PM record from the Internal Ingress Bank 2 Performance Monitoring Table.

Prototype

```
INT4 atlasReadIngressBank2PMRecord ( ATLAS atlas,UINT1 pmSessionId,
UINT1 clearOnRead, sATLS_PM_RECORD *pmRecord)
```

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device Handle (from `atlasAdd`)
`clearOnRead`	: 0 – registers not cleared upon read : 1 – registers cleared upon read
`pmSessionId`	: determines session Id to access (0-127),

Outputs

`pmRecord`	: pointer to pmRecord to be filled in.

Returns

```
ATLS_SUCCESS
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_PM_SESSION_ID
```

Reading PM Records: atlasReadEgressBank1PMRecord

This function reads a PM record from the Internal Egress Bank 1 Performance Monitoring Table.

Prototype `INT4 atlasReadEgressBank1PMRecord ( ATLAS atlas,UINT1 pmSessionId, UINT1 clearOnRead, sATLS_PM_RECORD *pmRecord)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`clearOnRead`	: 0 – registers not cleared upon read : 1 – registers cleared upon read
`pmSessionId`	: determines session Id to access (0-127),

Outputs

`pmRecord`	: pointer to pmRecord to be filled in.

Returns

`ATLS_SUCCESS`
`ATLS_ERR_INVALID_STATE`
`ATLS_ERR_INVALID_DEVICE`
`ATLS_ERR_INVALID_PM_SESSION_ID`

Reading PM Records: atlasReadEgressBank2PMRecord

This function reads a PM record from the Internal Egress Bank 2 Performance Monitoring Table.

Prototype `INT4 atlasReadEgressBank2PMRecord ( ATLAS atlas,UINT1 pmSessionId, UINT1 clearOnRead, sATLS_PM_RECORD *pmRecord)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`clearOnRead`	: 0 – registers not cleared upon read : 1 – registers cleared upon read
`pmSessionId`	: determines session Id to access (0-127),

Outputs

`pmRecord`	: pointer to pmRecord to be filled in.

Returns
```
ATLS_SUCCESS
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_PM_SESSION_ID
```

Getting Ingress VC PM parameters: atlasReadIngressPM

This function gets the Ingress VC PM configuration. This function returns the following VC ingress table fields :

Row 0 : pmActive2, pmActive1
Row 1 : pmAddr2, pmAddr1.

Prototype `INT4 atlasReadIngressPM(ATLAS atlas, UINT2 u2VcId, sATLS_INGRESS_PM *pPM)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs
`atlas` : device handle (from `atlasAdd`)
`u2VcId` : index to the VC table record.

Outputs `pPM` : PM config information structure.

Returns
```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT
```

Modifying Ingress VC PM parameters: atlasWriteIngressPM

This function modifies the Ingress VC PM configuration; the following VC ingress table fields are updated :

Row 0 : pmActive2, pmActive1
Row 1 : pmAddr2, pmAddr1.

Prototype `INT4 atlasReadIngressPM(ATLAS atlas, UINT2 u2VcId, sATLS_INGRESS_PM *pPM)`

Valid States ATLS_INIT, ATLS_ACTIVE

Inputs

atlas	: device handle (from atlasAdd)
u2VcId	: index to the VC table record.

Outputs

pPM	: PM config information structure.

Returns

ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT

Activating/Deactivating Ingress VC PM sessions: atlasWriteIngressPMActivation

This function activates/deactivates the Ingress VC PM session.

Prototype

```
INT4 atlasWriteIngressPMActivate(ATLAS atlas, UINT2 u2VcId,
UINT1 pm1, UINT1 pm2)
```

Valid States ATLS_INIT, ATLS_ACTIVE

Inputs

atlas	: device handle (from atlasAdd)
u2VcId	: index to the VC table record.
pm1	: PM1 activation flag (1 = activate, 0 = deactivate)
pm2	: PM2 activation flag (1 = activate, 0 = deactivate)

Outputs None

Returns

ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT

Modifiying Ingress PM Parameters: atlasModifyIngressVcPM

This function modifies the Performance Monitoring parameters of an existing Ingress VC. This allows the association of PM sessions with an Ingress VC; it also allows the enable/disable of Performance Monitoring for the connection. Only performance monitoring parameters are updated. All other fields are ignored.

Row 1: PmActive2, pmAddr2, pmActive1, pmAddr1

Prototype `INT4 atlasModifyIngressVcPM (ATLAS atlas, sATLS_INGRESS_RECORD *patlasIngressVCRecord)`

Valid States `ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`patlasIngressVCRecord`	: connection information structure.

Outputs None

Returns

```
ATLS_SUCCESS
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_PHYID
ATLS_ERR_INVALID_FIELDA
ATLS_ERR_INVALID_FIELDB
ATLS_ERR_PRIMARY_KEY_UNEXPECTED
ATLS_ERR_PRIMARY_ADDR_RANGE
ATLS_ERR_SECONDARY_ADDR_RANGE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_INVALID_VC_RECORD
```

Modifying Egresss PM Parameters: atlasModifyEgressVcPM

This function modifies the Performance Monitoring parameters of an existing connection. This allows the association of PM sessions with an Egress VC; it also allows the enable/disable of Performance Monitoring for the connection. Only performance monitoring parameters are updated. All other fields are ignored.

Row 0: pmActive2, pmactive,
Row 1: pmAddr2, pmAddr1

Prototype `INT4 atlasModifyEgressVcPM (ATLAS atlas,sATLS_EGRESS_RECORD *patlasEgressVCRecord)`

Valid States `ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`patlasEgressVCRecord`	: connection information structure

Outputs None

Returns

```
ATLS_SUCCESS
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_PHYID
ATLS_ERR_INVALID_FIELDA
ATLS_ERR_INVALID_FIELDB
ATLS_ERR_EVC_RECORD_ADDR_RANGE
ATLS_ERR_EVC_RECORD_NOT_EXIST
ATLS_ERR_INVALID_VC_RECORD
```

Getting Egress VC PM parameters: atlasReadEgressPM

This function gets the Egress VC PM configuration. This function returns the following VC egress table fields :

Row 0 : pmActive2, pmActive1
Row 1 : pmAddr2, pmAddr1

Prototype `INT4 atlasReadEgressPM(ATLAS atlas, UINT2 u2VcId, sATLS_EGRESS_PM *pPM)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.

Outputs

`pPM`	: PM config information structure.

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT
```

Modifying Egress VC PM parameters: atlasWriteEgressPM

This function modifies the Egress VC PM configuration; the following VC egress table fields are updated :

Row 0: pmActive2, pmActive1.

Row 1: pmAddr2, pmAddr1.

Prototype `INT4 atlasWriteEgressPM(ATLAS atlas, UINT2 u2VcId, sATLS_EGRESS_PM *pPM)`

Valid States `ATLS_INIT, ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`u2VcId`	: index to the VC table record.

Outputs

`pPM`	: PM config information structure.

Returns

```
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT
```

Activating/Deactivating Egress VC PM sessions: atlasWriteEgressPMActivation

This function activates/deactivates the Egress VC PM session.

Prototype `INT4 atlasWriteEgressPMActivate(ATLAS atlas, UINT2 u2VcId, UINT1 pm1, UINT1 pm2)`

Valid States ATLS_INIT, ATLS_ACTIVE

Inputs

atlas	: device handle (from atlasAdd)
u2VcId	: index to the VC table record.
pm1	: PM1 activation flag (1 = activate, 0 = deactivate)
pm2	: PM2 activation flag (1 = activate, 0 = deactivate)

Outputs None

Returns
ATLS_SUCCESS,
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_IVC_VC_NOT_EXIST
ATLS_ERR_REG_POLL_TIMED_OUT
ATLS_ERR_MOD_NOT_INIT

5.10 Cell Insertion/Extraction

This section defines functions that can be used to both insert cells into and remove cells from the MicroProcessor Cell interface.

As the Cell FIFOs are of limited depth, the user is responsible for processing received cells in a timely manner.

Writing Ingress Cell Buffer Structures: atlasInsertIngressCell

This function writes the cell buffer structure passed by the user into the ATLAS microprocessor ingress cell interface which is then transmitted by the device. This call will fail if the transmit FIFO is full.

Prototype
```
INT4 atlasInsertIngressCell(ATLAS atlas, UINT1 *psCellBuffer,
sATLS_CELL_INS_CTRL *psCtrl)
```

Valid States ATLS_ACTIVE

Inputs

`atlas`	: device handle (from `atlasAdd`)
`psCellBuffer`	: pointer to the buffer that contains the cell.
`psCtrl`	: controls the insertion of the cell into the ingress output

Outputs None

Returns

```
ATLS_SUCCESS
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_CELL_INS_CTRL
ATLS_ERR_ING_MUP_INS_BUSY
```

Reading Cells From Microprocessor Ingress: atlasExtractIngressCell

This function reads a cell from the microprocessor ingress FIFO for the specified device (if a cell is present). Users should keep reading cells until there are no more cells in the FIFO.

Prototype

```
INT4 atlasExtractIngressCell (ATLAS atlas,UINT1 *psCellBuffer,UINT1
extPhyId,UINT1 *plength, UINT1 *pphyId)
```

Valid States `ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`psCellBuffer`	: pointer to the buffer that will contain the cell.
`extPhyId`	: if non-zero, replace HEC with the PHYID; if zero, do not replace HEC with the PHYID

Outputs

`pLength`	: length of cell extracted
`pphyId`	: phyId

Returns

```
ATLS_SUCCESS
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_STATE
ATLS_NO_CELL
ATLS_ERR_INVALID_CELL_LENGTH
```

Reading Cells From Microprocessor Ingress (OAM cells only): atlasRxCell

This function reads a cell from the microprocessor ingress FIFO for the specified device (if a cell is present) and process it based on its OAM type.

Prototype `INT4 atlasRxCell(ATLAS atlas)`

Valid States `ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)

Outputs None

Returns

```
ATLS_SUCCESS
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_STATE
```

Writing Egress Cell Buffer Structures: atlasInsertEgressCell

This function writes the cell buffer structure passed by the user into the ATLAS that is then transmitted by the device. This call will fail if the transmit FIFO is full.

Prototype

```
INT4 atlasInsertEgressCell(ATLAS atlas,UINT1
*psCellBuffer,sATLS_CELL_INS_CTRL *psCtrl)
```

Valid States `ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`psCellBuffer`	: pointer to the buffer that contains the cell.
`psCtrl`	: controls the insertion of the cell into the egress output.

Outputs None

Returns

```
ATLS_SUCCESS
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_CELL_INS_CTRL
ATLS_ERR_EG_MUP_INS_BUS
```

Reading Cells from Microprocessor Egress: atlasExtractEgressCell

This function reads a cell from the microprocessor egress FIFO for the specified device (if a cell is present). Users should keep reading cells until no more cells are available.

Prototype `INT4 atlasExtractEgressCell (ATLAS atlas, UINT1 *psCellBuffer,UINT1 extPhyId, UINT1 *plength, UINT1 *pphyId)`

Valid States `ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`psCellBuffer`	: pointer to the buffer that will contain the cell.
`extPhyId`	: if non-zero, replace HEC with the PHYID; if zero, do not replace HEC with the PHYID

Outputs

`pLength`	: length of cell extracted.
`phyId`	: phyId

Returns

`ATLS_SUCCESS`
`ATLS_ERR_INVALID_DEVICE`
`ATLS_ERR_INVALID_STATE`
`ATLS_NO_CELL`
`ATLS_ERR_INVALID_CELL_LENGTH`

5.11 Per-PHY Policing

This set of functions provides access to the Internal Per-PHY Policing RAM.

Writing Internal Per-PHY Policing Contents: atlasWritePerPHYPolicing

This function writes the internal Per-PHY Policing RAM contents.

Prototype `INT4 atlasWritePerPHYPolicing(ATLAS atlas,UINT1 phyId, sATLS_PER_PHY_POLICING *pperPhyPolicing)`

Valid States `ATLS_ACTIVE`

Inputs

`altas`	: device handle (from `atlasAdd`)
`phyId`	: identifies the PHY device
`pperPhyPolicing`	: the new per PHY policing record

Outputs None

Returns

```
ATLS_SUCCESS
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_PHYID
```

Reading Internal Per-PHY Policing Contents: atlasReadPerPHYPolicing

This function reads the internal Per-PHY Policing RAM contents.

Prototype

```
INT4 atlasReadPerPHYPolicing(ATLAS atlas,UINT1 phyId,
sATLS_PER_PHY_POLICING *pperPhyPolicing)
```

Valid States `ATLS_ACTIVE`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`phyId`	: identifies the PHY device

Outputs

`pperPhyPolicing`	: the new per PHY policing record

Returns

```
ATLS_SUCCESS
ATLS_ERR_INVALID_STATE
ATLS_ERR_INVALID_DEVICE
ATLS_ERR_INVALID_PHYID
```

5.12 Statistics

This set of functions provides access to device counters and PHY counters, and to status information maintained by the device.

The Static configuration of the counters, latch-high or rollover, is performed via the Initialization vector.

The ATLAS device will indicate when counters are half full. It is the user's responsibility to ensure that they are read before any information is lost.

Reading Device Wide Counters: atlasGetDeviceCounts

This function reads the Device Wide counters and writes them into the Counter Structure specified.

Prototype `INT4 atlasGetDeviceCounts (ATLAS atlas, sATLS_AGGR_COUNTS *patlasAggrCounters, UINT1 clearOnRead)`

Valid States `ATLS_ACTIVE`

Inputs	`atlas`	: device handle (from `atlasAdd`)
	`clearOnRead`	: 0 – registers not cleared upon : 1 – registers cleared upon read
Outputs	`patlasAggrCounters`	: structure updated to reflect current device counters

Returns
`ATLS_SUCCESS`
`ATLS_ERR_INVALID_STATE`
`ATLS_ERR_INVALID_DEVICE`

5.13 Interrogation

This section defines the functions available to the user that can be used to request information from the ATLAS device.

These functions will typically be used after the ATLAS device has indicated, via the Indication Callback function and associated status, that this information is available.

Reading Ingress VC Table Changes: atlasGetIngressVCTableCOS

This function reads the Ingress VC Table Change of Connection Status FIFO Registers and returns their value. The user should call this function until the FIFO is empty.

Prototype `INT4 atlasGetIngressVCTableCOS (ATLAS atlas, UINT2 *pCOS)`

Valid States `ATLS_ACTIVE`

Inputs	`atlas`	: device handle (from `atlasAdd`)
Outputs	`pCOS`	: address to write Change Of State data

Returns	`ATLS_SUCCESS`
	`ATLS_ERR_INVALID_STATE`
	`ATLS_ERR_INVALID_DEVICE`
	`ATLS_COS_EMPTY`	

Reading Egress VC Table Changes: atlasGetEgressVCTableCOS

This function reads the Egress VC Table Change of Connection Status FIFO Registers and returns their values. The user should call this function until the FIFO is empty

Note: This function can only be called from the ACTIVE state.

Prototype	`INT4 atlasGetEgressVCTableCOS (ATLAS atlas,UINT2 *pCOS)`	
Valid States	`ATLS_ACTIVE`	
Inputs	`atlas`	: device handle (from `atlasAdd`)
Outputs	`pCOS`	: address to write Change Of State data

Returns	`ATLS_SUCCESS`
	`ATLS_ERR_INVALID_STATE`
	`ATLS_ERR_INVALID_DEVICE`
	`ATLS_COS_EMPTY`	

5.14 Interrupt Servicing

The ATLAS driver provides an ISR as well as a DPR for interrupt servicing.

Reading Interrupt Status Registers: atlasISR

This function reads the interrupt status registers of the interrupting device and outputs the status information for later interpretation and processing by `sysAtlasIntHandler`. `sysAtlasIntHandler` will typically request the Deferred Processing Routine (DPR) to process the interrupt at a later time.

If this function returns an `ATLS_INTERRUPTS_OCCURRED` value the `sysAtlasIntHandler` should send the associated Interrupt Status to the DPR Task using RTOS mechanisms.

Returning a value from this function enables it to be called directly by a system specific high priority, polling task. Only if an `ATLS_INTERRUPTS_OCCURRED` value is returned should the polling task send the status to the Deferred Processing routine.

This function may make a callback directly to the user's Exception Callback if an exception condition has occurred. Exception conditions indicate device problems that should not occur in normal operation. The user should decode the indicated exceptions and take appropriate, system-specific action as indicated.

The DPR Indication callback `indIndication` is intended to process expected, normal-case indications of device operation.

Prototype	`INT1 atlasISR(ATLAS atlas, sATLS_INT_STATUS *patlasIntStatus)`	
Valid States	`ALL STATES`	
Inputs	`atlas`	: device handle (from `atlasAdd`)
Outputs	`patlasIntStatus`	: interrupt status information
Returns	`ATLS_SUCCESS` `ATLS_INTERRUPTS_OCCURRED`	: no interrupts occurred : one or more interrupts occurred.

Processing Interrupt Status Information: atlasDPR

This function processes interrupt status information queued to it by the ISR. Processing involves ensuring an interrupt is indicated and calling the user's indication function `indIndication` with the indicated Interrupt Status.

The user should decode the interrupt status and take appropriate, system-specific action on the indicated conditions.

Prototype `void atlasDPR(ATLAS atlas,sATLS_INT_STATUS *patlasIntStatus)`

Valid States `ALL STATES`

Inputs

`atlas`	: device handle (from `atlasAdd`)
`patlasIntStatus`	: interrupt status information

Outputs None

Returns None

6 HARDWARE INTERFACE

6.1 Device I/O

Reading Specific Address Contents: sysAtlasRawRead

This function is a low-level, system-specific macro, written in the format of a function, that can be used to read the contents of a specific ATLAS register. This macro should be modified by the user to reflect the system addressing logic.

The following definition reflects the ATLAS registers presenting on 32-bit address boundaries, and the register value presenting in the bottom 16 bits.

Prototype `#define sysAtlasRawRead(baseAddr,regId,pval),sysAtlasRawRead(UINT1 *baseAddr, UINT2 regId, UINT2 *pval)`

Valid States `ALL STATES`

Inputs	`baseAddr`	: address of base register for ATLAS (Register 0)
	`regId`	: Id of Register to read
Outputs	`pval`	: value read

Writing Specific Address Contents: sysAtlasRawWrite

This function is a low-level, system-specific macro, written in the format of a function, that can be used to write the contents of a specific ATLAS register. This macro should be modified by the user to reflect the system addressing logic.

The following definition reflects the ATLAS registers presenting on 32-bit address boundaries, and the register value presenting in the bottom 16 bits.

Prototype `#define sysAtlasRawWrite(baseAddr,regId,val),`
`sysAtlasRawWrite(UINT1 *baseAddr, UINT2 regId, UINT2 val)`

Valid States `ALL STATES`

Inputs	`baseAddr`	: address of base register for ATLAS (Register 0)
	`regId`	: Id of Register to Write
	`val`	: value to write

Outputs None

Detecting New Devices: sysAtlasDeviceDetect

This function uses user context information to detect new ATLAS devices. The `atlasAdd` API function calls it. This function is system specific.

Prototype `void syAtlasDeviceDetect(ATLS_USR_CTXT userContext, void **pdevBaseAddr)`

Inputs `userContext` : user Information Context

Outputs `pdevBaseAddr` : updated to reflect Base Address of device

Returns `ATLS_SUCCESS`
`ATLS_NO_DEVICE`

6.2 Interrupt servicing

ISR Installation and removal

The following system specific routines install and remove the interrupt handlers and deferred processing routines for the ATLAS devices.

Installing Process Vector Tables: sysAtlasIntInstallHandler

This function installs sysAtlasIntHandler in the processor vector table, spawns the `sysAtlasDPRtask` routines as a task, and creates a communication channel (for example, a message queue) between the two. This function could also choose to create the high priority polling task rather than install the ISR if the polling model was to be used.

Prototype `void sysAtlasIntInstallHandler(void)`

Inputs None

Outputs None

Returns None

Deleting Message Queues: sysAtlasIntRemoveHandler

This function deletes the sysAtlasDPRtask and associated message queue. It also removes the `sysAtlasIntHandler` routine from the processor's interrupt vector table. If the polling model was in use, this task would delete the high priority polling task.

Prototype `void sysAtlasIntRemoveHandler(void)`

Inputs None

Outputs None

Returns None

System specific ISR and DPR routines

The system-specific ISR and DPR routines, `sysAtlasIntHandler` and `sysAtlasDPRtask`, are installed by the `sysAtlasIntInstallHandler` routine.

Enabling Interrupt Processing: sysAtlasIntHandler

This routine is invoked when one or more ATLAS devices indicate interrupts. This routine invokes atlasISR for each device for which interrupt processing is enabled. For each device indicating an interrupt, this routine queues the interrupt status information for later processing by `sysAtlasDPRtask`.

If there are several ATLAS device being managed there may be several messages queued from the atlasISR to the `sysAtlasDPRtask`.

Prototype `void sysAtlasIntHandler(void)`

Inputs None

Outputs None

Returns None

Retrieving Interrupt Status Information: sysAtlasDPRtask

This routine is spawned as a separate task within the RTOS. It retrieves interrupt status information saved for it by the `sysAtlasIntHandler` routine and invokes the `atlasDPR` routine indicating the appropriate device.

Prototype `void sysAtlasDPRtask(void)`

Inputs None

Outputs None

Returns None

7 RTOS INTERFACE

7.1 Service Calls

The ATLAS driver uses the following RTOS services.

Memory Allocation/De-allocation

Allocating Bytes: sysAtlasMemAlloc

This function allocates a specified number of bytes.

Prototype `void *sysAtlasMemAlloc(UINT2 alloc_size)`

Inputs `alloc_size` : number of bytes to be allocated

Outputs None

Returns Pointer to first byte of allocated memory
NULL pointer (memory allocation failed)

De-Allocating Memory: sysAtlasMemFree

This function de-allocates memory allocated using `ATLASMemAlloc`.

Prototype `void sysAtlasMemFree(void *addr)`

Inputs `addr` : pointer to memory to be freed

Outputs None

Returns None

Setting Memory to a specified value: sysAtlasMemSet

This function is used to zero memeory. The driver uses this to reset data-structures.

Prototype	`VOID sysAtlasMemSet(VOID *ptr, UINT1 byte, UINT4 byteCount)`	
Inputs	`ptr`	: pointer to start of memory to set
	`byte`	: byte value to set in memory
	`bytecount`	: the number of bytes, starting at `ptr`, to set.
Outputs	None	
Returns	None	

7.2 Indication Callbacks

The ATLAS driver uses the following indication routines to notify the applications of events within the device and driver. These routines need to be implemented by the user.

The callback functions will be passed a copy of the full Interrupt Status Registers.

The ATLAS device has many possible indications. These are categorized into two groups.

System Exceptions (SE). These indicate device failure conditions. This indication will be made in the context of the device ISR. Therefore no RTOS blocking calls should be made.

Application Indications (AI). These indicate device conditions that require application attention.

When the `atlasISR` is invoked it will examine the Interrupt status mask to decide whether a System Exception indication should be raised. If so, the ISR will call the `indException` callback directly.

If there is any interrupt status indicated, the `altasISR` function will indicate this to the user supplied function `sysAtlasIntHandler`. This function should then pass the interrupt status to the atlasDPR function.

If an interrupt notification has been passed from to the `atlasDPR` from the `atlasISR`, the `atlasDPR` will make the `indIndication` callback.

Each callback function will be called with the applications context ID and the full Interrupt Status Register contents.

The following Tables indicate which categories the Interrupt indications fall into.

Table 34: ATLAS Interrupt Reg #1 Categories

Interrupt Name	Category
ATLS_INT_1_I_INVALI	AI
ATLS_INT_1_I_PTIVCII	AI
ATLS_INT_1_I_OAMERRI	AI
ATLS_INT_1_I_SEG_CCI	AI
ATLS_INT_1_I_END_CCI	AI
ATLS_INT_1_I_SEG_AISI	AI
ATLS_INT_1_I_END_AISI	AI
ATLS_INT_1_I_SEG_RDII	AI
ATLS_INT_1_I_END_RDII	AI
ATLS_INT_1_I_POLI	AI
ATLS_INT_1_I_XPOLI	AI
ATLS_INT_1_I_XFERI	AI

Table 35: ATLAS Interrupt Reg #2 Categories

Interrupt Name	Category
ATLS_INT_2_I_FULLI	AI
ATLS_INT_2_I_UPFOVRI	AI
ATLS_INT_2_I_UPCAI	AI
ATLS_INT_2_I_INSRDYI	AI
ATLS_INT_2_I_RSOCI	SE
ATLS_INT_2_I_RPRTYI	SE
ATLS_INT_2_I_PCELLI	SE
ATLS_INT_2_I_SRCHERRI	AI
ATLS_INT_2_I_BCIFFULLI	AI
ATLS_INT_2_I_PHYPOLI	AI
ATLS_INT_2_I_PHYXPOLI	AI
ATLS_INT_2_I_COSFULLI	AI
ATLS_INT_2_I_XCOSI	AI

Table 36: ATLAS Interrupt Reg #3 Categories

Interrupt Name	Category
ATLS_INT_3_I_SPRTYI[0]	SE
ATLS_INT_3_I_SPRTYI[1]	SE
ATLS_INT_3_I_SPRTYI[2]	SE
ATLS_INT_3_I_SPRTYI[3]	SE
ATLS_INT_3_I_SPRTYI[4]	SE
ATLS_INT_3_I_SPRTYI[5]	SE
ATLS_INT_3_I_SPRTYI[6]	SE
ATLS_INT_3_I_SPRTYI[7]	SE
ATLS_INT_3_E_SPRTYI[0]	SE
ATLS_INT_3_E_SPRTYI[1]	SE
ATLS_INT_3_E_SPRTYI[2]	SE
ATLS_INT_3_E_SPRTYI[3]	SE

Table 37: ATLAS Interrupt Reg #4 Categories

Interrupt Name	Category
ATLS_INT_4_E_XFERI	AI
ATLS_INT_4_E_PTIVCII	AI
ATLS_INT_4_E_OAMERR	AI
ATLS_INT_4_E_SEG_CCI	AI
ATLS_INT_4_E_END_CCI	AI
ATLS_INT_4_E_Seg_AISI	AI
ATLS_INT_4_E_End_AISI	AI
ATLS_INT_4_E_Seg_RDII	AI
ATLS_INT_4_E_End_RDII	AI
ATLS_INT_4_E_FULLI	AI
ATLS_INT_4_E_UPFOVRI	AI
ATLS_INT_4_E_UPCAI	AI
ATLS_INT_4_E_INSRDYI	AI
ATLS_INT_4_E_IOVRI	AI

Interrupt Name	Category
ATLS_INT_4_E_PCELLI	AI

Table 38: ATLAS Interrupt Reg #5 Categories

Interrupt Name	Category
ATLS_INT_5_E_ISOCI	SE
ATLS_INT_5_E_IPRTYI	SE
ATLS_INT_5_E_IWRENBI	SE
ATLS_INT_5_E_BCIFFULLI	AI
ATLS_INT_5_E_COSFULLI	AI
ATLS_INT_5_E_XCOSI	AI
ATLS_INT_5_E_SEARCHEI	AI

Callback Functions

The following defines the callback functions that the user needs to provide.

Informing the User of OAM Cell Processing: indOAMRx

This function informs the user that a OAM cell has been processed (Loopback or Activation/Deactivation CC). This function will be called by the `altasRxCell` routine.

Prototype

```
void indOamRx(ATLS_USR_CTXT usrCtxt, INT4 u4OamType, UINT1 u1CmdFlag,
INT4 arg1, INT4 arg2, INT4 result)
```

Inputs

`usrCxt` : user's context information for this ATLAS device

`u4OamType` : Indicates the type of the OAM cell

`ATLS_OAM_LPBK` : Indicates a Loopback cell.

`ATLS_OAM_CC_ACT_DEACT` : Indicates a activation/deactivation CC cell

`u1CmdFlag` : Indicates the type of cell received

`ATLAS_OAM_RESPONSE` : An OAM response has been received (confirmation message)

`ATLAS_OAM_COMMAND` : An OAM command has been received (activation/decativation)

`arg1` : OAM function type.

`arg2` : The connection VC Id.

`result` : Indicates if the OAM processing has been succesful.

Outputs None

Returns None

Requesting a Backward Connection Information from the User : indBackEci

This function requests the backward connection information (needed for the OAM processing) to the user. This function will be called by the `altasRxCell` routine.

Prototype

```
void indBackEci(ATLS_USR_CTXT usrCtxt, UINT2 u2ICI, INT4 *pResult,
UINT1 *pu1PhyId, UINT2 *pu2ECI)
```

Inputs

`usrCxt` : user's context information for this ATLAS device

`u2ICI` : The connection ID of the forward OAM connection

`pResult` : The result of the request returned by the user.

`pu1PhyId` : The Phy ID of the backward connection returned by the user.

`pu2ECI` : The backward connection VC ID returned by the user.

Outputs None

Returns None

Informing the User of an Occurrence of a System Exception : indException

This function informs the user that a System Interrupt/Exception has occurred. The user should process all indicated exceptions in this category. This function will be called by the `altasISR` routine. Note that this function, since it is called in the context of an ISR, should not make any blocking calls to an RTOS.

This function should be used to process the interrupts that indicate a device exception. An exception is a condition that is not part of the standard operation of the device.

All available interrupts/exceptions will be indicated in the status.

Prototype
```
void indException(ATLS_USR_CTXT usrCtxt, sATLS_INT_STATUS *intStatus,
UINT1 arg1, UINT1 arg2, UINT1 arg3)
```

Inputs

`usrCxt`	: user's context information for this ATLAS device
`intStatus`	: structure indicating all the ATLAS interrupts indicated.
`arg1`	: not used.
`arg2`	: not used.
`arg3`	: not used.

Outputs None

Returns None

Informing the User of an Occurrence of a Normal Interrupt : indIndication

This function informs the user that a normal Interrupt indicating that user action is required has occurred. The user should process all indicated interrupts in this category. This function will be called by the `altasDPR` routine.

This function should be used to process the interrupts that indicate the normal operation of the device. This can be microprocessor cell availability, counter overflow, etc.

All available interrupts/exceptions will be indicated in the status.

Prototype `void indIndication(ATLS_USR_CTXT usrCtxt, sATLS_INT_STATUS *intStatus)`

Inputs

`usrCxt`	: user's context information for this ATLAS device
`intStatus`	: structure indicating all the ATLAS interrupts indicated.

Outputs None

Returns None

8 PORTING THE DRIVER

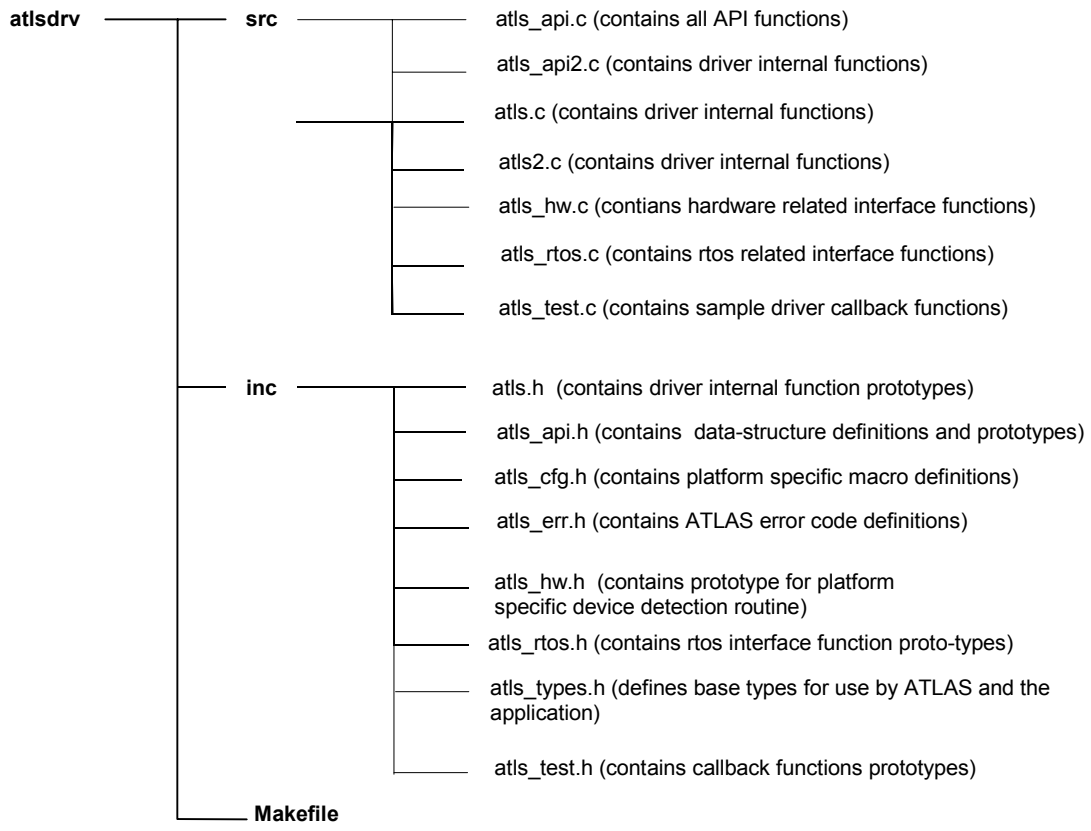
This section outlines how to port the ATLAS device driver to your hardware and RTOS platform.

Note: Because each platform and application is unique, this manual can only offer guidelines for porting the ATLAS driver.

8.1 Driver Source Files

The C source files listed in Figure 8 contain the code for the ATLAS driver. You may need to modify the code or develop additional code. The code is in the form of constants, macros, and functions. For the ease of porting, the code is grouped into source files (`src`) and include files (`inc`). The `src` files contain the functions and the `inc` files contain the constants and macros.

Figure 8: Driver Source Files



8.2 Driver Porting Procedures

The following steps summarize how to port the ATLAS driver to your platform. The following sections describe these steps in more detail.

Note: Because each platform and application is unique, this manual can only offer guidelines for porting the ATLAS driver.

8.3 Driver Porting Procedures

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

Procedure 1: Port the driver's RTOS extensions (page 132):

Procedure 2: Port the driver to your hardware platform (page 133):

Procedure 3: Port the driver's application-specific elements (page 134):

Procedure 4: Build the driver (page 135).

Procedure 1 : Porting the Driver's RTOS Extensions

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

- Task management
- Message queues
- Memory Management

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

To port the driver's RTOS extensions:

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

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

Service Type	Macro Name	Description
Memory	sysAtlasMemCopy	Copies the memory block from src to dest

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

Service Type	Function Name	Description
Memory	sysAtlasMemAlloc	Allocates the memory block
	sysAtlasMemFree	Frees the memory block
Interrupt	sysAtlasIntInstallHandler	Installs the interrupt handler for the RTOS
	sysAtlasIntRemoveHandler	Removes the interrupt handler from the RTOS
	sysAtlasIntHandler	Interrupt handler for the ATLAS device
	sysAtlasDPRTask	Deferred process routine for interrupts

Procedure 2: Porting the Driver to a Hardware Platform

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

To port the driver to your hardware platform:

1. Modify the device detection function in the atls_hw.c file. The function sysAtlasDeviceDetect is implemented for a PCI platform. Modify it to reflect your specific hardware interface. Its purpose is to detect a ATLAS device based on a UsrContext input parameter. It returns two output parameters:
    - The base address of the ATLAS device
    - A pointer to the system-specific configuration information
2. Modify the low-level device read/write macros in the atls_cfg.h file. You may need to modify the raw read/write access macros (sysAtlasRawRead and sysAtlasRawWrite) to reflect your system's addressing logic.

Procedure 3: Porting the Driver's Application-Specific Elements

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

Before you port the driver's application-specific elements, ensure that you:

1. Port the driver's RTOS extensions
2. Port the driver to your hardware platform

To port the driver's application-specific elements:

1. Define the following driver task-related constants for your RTOS-specific services in file `atls_rtos.h` and `atls_cfg.h`

#Define	Description	Default
`ATLS_DPR_TASK_PRIORITY`	Deferred Task (DPR) task priority	`85`
`ATLS_DPR_TASK_STACK_SIZE`	DPR task stack size, in bytes	`4096`
`ATLS_POLLING_DELAY`	Constant used in polling task mode, this constant defines the interval time in millisecond between each polling action	`50`
`ATLS_TASK_SHUTDOWN_DELAY`	Delay time in millisecond. When clearing the DPR loop active flag in the DPR task, this delay is used to gracefully shutdown the DPR task before deleting it.	`10`
`ATLS_MAX_DPR_MSGS`	The queue message depth of the queue used for pass interrupt context between the ISR task and DPR task	`10`
`ATLS_MAX_NUM_DEVS`	The maximum number of ATLAS devices in the system	`1`

2. Code the callback functions according to your application. There are four sample callback functions in the `atls_test.c` file. You can use these callback functions or you can customize them before using the driver. The driver will call these callback functions when an event occurs on the device. These functions must conform to the following prototypes:
    - `void indOamRx(ATLS_USR_CTXT usrCtxt, INT4 u4OamType, UINT1 u1CmdFlag, INT4 arg1, INT4 arg2, INT4 result)`
    - `void indBackEci(ATLS_USR_CTXT usrCtxt, UINT2 u2ICI, INT4 *pResult, UINT1 *pu1PhyId, UINT2 *pu2ECI)`

- `void indIndication(ATLS_USR_CTXT usrCtxt, sATLS_INT_STATUS *intStatus)`
- `void indException(ATLS_USR_CTXT usrCtxt, sATLS_INT_STATUS *intStatus, UINT1 arg1, UINT1 arg2, UINT1 arg3)`

3. The driver processes only two types of cells through the microprocessor port : OAM Loopback and CC Activation/Deactivation. If you want to process other types of cells through the microprocessor port, you should modify the atlasRXCell API in the atls_api2.c file.

Procedure 4: Building the Driver

This section describes how to build the ATLAS 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 ATLAS API driver library using your make utility.
4. Link the ATLAS API driver library to your application code.

APPENDIX: CODING CONVENTIONS

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

Variable Type Definitions

Table 39: Variable Type Definitions

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

Naming Conventions

Table 30 presents a summary of the naming conventions followed by all PMC driver software. A detailed description is then given in the following sub-sections.

The names used in the drivers are verbose enough to make their purpose fairly clear. This makes the code more readable. Generally, the device's name or abbreviation appears in prefix.

Table 40: Naming Conventions

Type	Case	Naming convention	Examples
Macros	Uppercase	prefix with "m" and device abbreviation	mAtlas_WRITE
Constants	Uppercase	prefix with device abbreviation	ATLS_REG
Structures	Hungarian Notation	prefix with "s" and device abbreviation	sATLS_INIT_VECTOR
API Functions	Hungarian Notation	prefix with device name	atlasAdd()

Type	Case	Naming convention	Examples
Porting Functions	Hungarian Notation	prefix with “sys” and device name	`sysAtlasRead()`
Other Functions	Hungarian Notation		`myOwnFunction()`
Variables	Hungarian Notation		`maxDevs`
Pointers to variables	Hungarian Notation	prefix variable name with “p”	`pmaxDevs`
Global variables	Hungarian Notation	prefix with device name	`atlasGDD`

Macros

The following list identifies the marcro 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: `mAtlas_WRITE` is a valid name for a macro.

Constants

The following list identifies the const ants 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: `ATLS_REG` is a valid name for a constant.

Structures

The following list identifies the structures 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: `sATLS_INIT_VECTOR` 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: `atlasAdd()` is a valid name for an API function.

Porting Functions

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

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: `sysAtlasRead()` 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: `myOwnFunction()` 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 `atlasBaseAddress` 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 41 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. `atls_api1.c` or `atls_api2.c`).

There are 4 different types of files:

The API file containing all the API functions

The hardware file containing the hardware dependant functions

The RTOS file containing the RTOS dependant functions

The other files containing all the remaining functions of the driver

Table 41: File Naming Conventions

File Type	File Name
API	`atls_api1.c, atls_api.h`
Hardware Dependant	`atls_hw.c, atls_hw.h`
RTOS Dependant	`atls_rtos.c, atls_rtos.h`
Other	`atls_isr.c, atls_defs.h`

API Files

The name of the API files must start with the device abbreviation followed by an underscore and ‘`api`’. Eventually a number might be added at the end of the name.

Examples: `atls_api1.c` is the only valid name for the file that contains the first part of the API functions, `atls_api.h` is the only valid name for the file that contains all of the API functions headers.

Hardware Dependent Files

The name of the hardware dependent files must start with the device abbreviation followed by an underscore and '`hw`'. Eventually a number might be added at the end of the file name.

Examples: `atls_hw.c` is the only valid name for the file that contains all of the hardware dependent functions, `atls_hw.h` is the only valid name for the file that contains all of the hardware dependent functions headers.

RTOS Dependant Files

The name of the RTOS dependant files must start with the device abbreviation followed by an underscore and '`rtos`'. Eventually a number might be added at the end of the file name.

Examples: `atls_rtos.c` is the only valid name for the file that contains all of the RTOS dependent functions, `atls_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: `atls_isr.c` is a valid name for a file that would deal with initialization of the device, `atls_defs.h` is a valid name for the corresponding header file.

ACRONYMS

API: Application Programming Interface

DDB: Device Data Block

DIV: Device Initialization Vector

DPR: Deferred Processing Routine

FIFO: First in, first out

GDD: Global driver database

GPIC: PCI controller

HCS: Header check sequence

HDLC: High-level data link control

ISR: Interrupt Service Routine

MIV: Module Initialization Vector

MVIP: Multi-vendor integration protocol

PCI: Processor connection interface

PHY: Physical layer

RAPI: Receive Any-PHY packet interface

RTOS: Real-Time operating system

INDEX