PM73122, PM73123, PM73124

AAL1GATOR-32/-8/-4

DRIVER USER'S MANUAL

PROPRIETARY AND CONFIDENTIAL

RELEASE

ISSUE 3: AUGUST, 2001

ABOUT THIS MANUAL AND AAL1GATOR-32/-8/-4

This manual describes the AAL1gator-32/-8/-4 device driver. It describes the driver's functions, data structures, and architecture. This manual focuses on the driver's interfaces to the application code, real-time operating system, and to the AAL1gator-32/-8/-4 device. It also describes in general terms how to modify and port the driver to your software and hardware platform.

The AAL1gator-32/-8/-4 Device Driver will support the AAL1gator-32 (PM73122), AAL1gator-8 (PM73123), and AAL1gator-4 (PM73124) devices. The Device Driver identifies which of the three Devices is installed and performs its functions accordingly. In systems with more than one Device, any combination of the three supported Devices is allowed.

The abbreviation used in this user's manual for the AAL1gator-32/-8/-4 is 'AAL1gator-32'. Constants are prefixed with '`AL3_`' and APIs are prefixed with '`al3`' (e.g. `al3ModuleOpen()`).

Audience

This manual is written for people who need to:

- Evaluate and test the AAL1gator-32/-8/-4 devices.
- Modify and add to the AAL1gator-32/-8/-4 driver functions.
- Port the AAL1gator-32/-8/-4 driver to a particular platform.

References

For more information about the AAL1gator-32 driver, see the driver's release notes. For more information about the AAL1gator-32, AAL1gator-8, and AAL1gator-4 devices, see the documents listed in the table below and any related errata documents.

Table 1: Related Documents

Document Number	Document Name
PMC-2000024	AAL1gator Product Family Technical Overview
PMC-2000088	AAL1gator White Paper (Network Convergence Of Voice, Data And Video)
PMC-1981419	AAL1gator-32 Data Sheet
PMC-1991271	AAL1gator-32 Short Form Data Sheet

Document Number	Document Name
PMC-1990887	AAL1gator-32 Reference Design
PMC-2000097	AAL1gator-8 Data Sheet
PMC-1991272	AAL1gator-8 Short Form Data Sheet
PMC-1991089	AAL1gator-8 Paper Reference Design
PMC-2000095	AAL1gator-8/-4 Designs Application Note
PMC-1991273	AAL1gator-4 Short Form Data Sheet
PMC-1991820	AAL1gator-32/8/4 Programmer's Guide

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	May, 2000	Document created
Issue 2	June 2001	Idle Channel Detection parameters changed for proper configuration. Interrupt and Deferred Processing vectors changed to reflect new Interrupt processing architecture. Added busMaster and twoC1FPEnable to SBI bus configuration. Added shiftCAS to Line Configuration. Fixed typographical errors.
Issue 3	August, 2001	Change Product Status from Preliminary to Release

Legal Issues

PMC-1991444 (P2), ref PMC-990901 (P1)

Contacting PMC-Sierra

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

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

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

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 DRIVER PORTING QUICK START

This section summarizes how to port the AAL1gator-32 device driver to your hardware and Real-time Operating System (RTOS) platform. For more information about porting the AAL1gator-32 driver, see Section 7.5 (page 115).

Note: Because each platform and application is unique, this manual can only offer guidelines for porting the AAL1gator-32 driver.

AAL1gator-32 driver code is organized into C source files. You may need to modify the code or develop additional code. The code is in the form of constants, macros, and functions. For ease of porting, the code is grouped into source files (`src`) and include files (`inc`). The source files contain the functions and the include files contain the constants and macros.

To port the AAL1gator-32 driver to your platform:

1. Port the driver's RTOS extensions (page 116):
   - Data types
   - RTOS-specific services
   - Utilities and interrupt services that use RTOS-specific services
2. Port the driver to your hardware platform (page 118):
   - Port the device detection function.
   - Port low-level device read-and-write macros.
   - Define hardware system-configuration constants.
3. Port the driver's application-specific elements (page 119):
   - Define the task-related constants.
   - Code the callback functions.
4. Build the driver (page 120).

2 DRIVER FUNCTIONS AND FEATURES

This section describes the main functions and features supported by the AAL1gator-32 driver.

Table 2: Driver Functions and Features

Function	Description
Device Initialization and Reset (page 64)	Initializes the AAL1gator-32 driver and its associated context structures. This involves reading in an initialization vector that contains various configuration parameters such as interface configuration. The driver validates this vector and the AAL1gator-32 device configures accordingly. The function also resets the AAL1gator-32 and the context information for that device.
Device Addition and Deletion (page 65)	Allocates and initializes memory to store context information for the device being added. De-allocates device context memory during device shutdown. You must locate the device on the Address Bus before you add the device.
Channel Provisioning (page 70)	Configures the channels of the AAL1gator-32 device by programming channel registers according to application parameters.
Statistics Collection and Status Monitoring (page 83)	Polls the various AAL1gator-32 counters so that they do not max out at 16 bits. Monitors device status (via interrupts or polling) and invokes application-defined callback functions when significant alarm/error events occur.
Interrupt Servicing (page 93)	Clears the interrupts raised by the AAL1gator-32 and stores the interrupt status for later processing by a deferred processing routine. The deferred processing routine runs in the context of a separate task within the RTOS and takes appropriate actions based on the interrupt status retrieved by the Interrupt Servicing Routine (ISR). This is true for both polled operation or interrupt operation. In polled mode, a separate task polls the interrupt status registers periodically. Once called the flows remain identical to the interrupt mode.
Device Diagnostics (page 98)	The driver will perform the following optional device diagnostics as part of a power-on self-test: • Tests register access • Tests RAM access

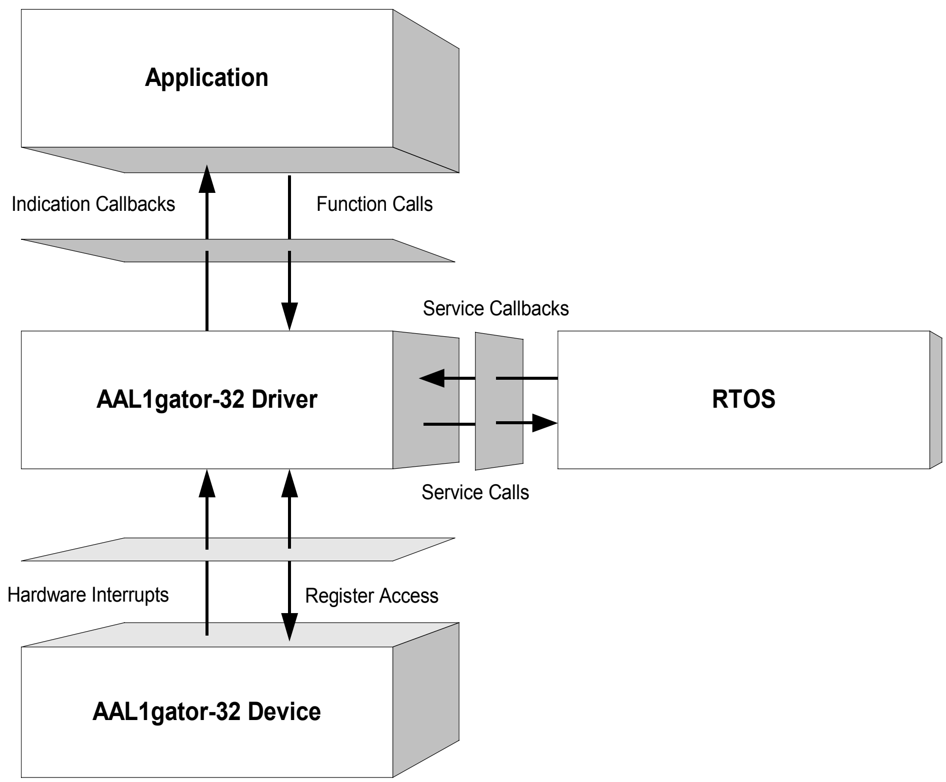
3 SOFTWARE ARCHITECTURE

This section describes the software architecture of the AAL1gator-32 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 AAL1gator-32 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 AAL1gator-32 device, such as:

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

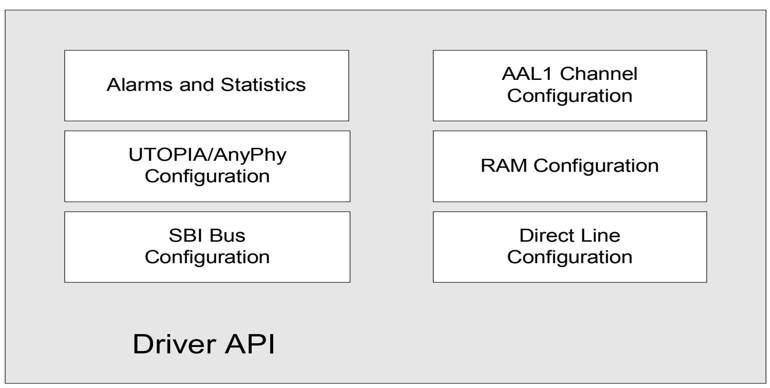
The driver API functions use the driver library functions as building blocks to provide this system level functionality (see below).

Driver API

The Driver Application Programming Interface (API) lists high-level functions that are invoked by application code to configure, control and monitor AAL1gator-32 devices. The API functions perform operations that are more meaningful from a system's perspective. The API includes functions that initialize the devices, perform diagnostic tests, validate configuration information to prevent incorrect configuration of the devices, and retrieve status and statistics information. The Driver API functions use the services of the other driver modules to provide this system-level functionality to the application programmer.

In addition, the Driver API consists of callback routines used to notify the application of significant events that take place within the device(s) and module.

Figure 2: Driver API Components



Alarms and Statistics

Alarms and Statistics functions are responsible for tracking devices status information and accumulating statistical counts for each device registered with (added to) the driver. This information is stored for later retrieval by the application software, and is also responsible for generating various alarms.

AAL1 Channel Configuration

AAL1 Channel Configuration functions are responsible for the provisioning and configuration of AAL1 Channels. This includes activating channels for structured and unstructured lines. For structured lines, timeslots are bundled to create AAL1 channels. These lines or bundles of timeslots then map to ATM VCs and in the process have several operating parameters configured.

AAL1 Channels are configured by using some or all of the available operating parameters. The "standard" channel configuration functions allow the user to easily configure an AAL1 channel by using defaults for most of the channel configuration parameters. The "enhanced" channel configuration functions open up all the configuration options to the user and are grouped so that a user can selectively configure a group or leave it in the default configuration.

The AAL1 channel configuration groups supported are: standard (the minimal parameters); enhanced; sequence number processing; conditioning; and idle detection. The user can configure AAL1 channels in any combination of the above

UTOPIA/Any-PHY Configuration

The UTOPIA/Any-PHY bus is the interface to the ATM side of the AAL1gator-32 devices. The source `(Tx)` and sink `(Rx)` sides of the bus are separately configurable.

RAM Configuration

The RAM interface is the interface between the AAL1gator-32 devices and their SRAMs, and it is here that configuration and statistics data structures are stored.

SBI Bus Configuration

The SBI bus is a parallel interface to TDM traffic that is only supported by the AAL1gator-32 (not the AAL1gator-8 and AAL1gator-4). This interface is capable of delivering combinations of T1/E1/DS3 to the AAL1gator-32 device. This section is responsible for configuring the SBI Bus Interface. SBI tributary types and mappings are configurable. The AAL1gator-32 device supports two pages of SBI Tributary mappings, one of which is configured as active by the application (the other is left inactive). This support enables the application to make changes to the inactive page before returning to active mode.

Direct Line Configuration

The Direct Line interface bypasses the SBI and H/MVIP blocks and brings clock & data signals out of the Device for connection to external framer(s). The Direct Line Interface supports DS3 & E3, E1 & T1 connections.

3.3 Real Time Operating System

The RTOS interface module provides functions that enable the driver to use RTOS services. The AAL1gator-32 driver requires memory, interrupt, and preemption services from the RTOS.

The RTOS interface functions perform the following tasks for the AAL1gator-32 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.

3.4 Driver Hardware Interface

The AAL1gator-32 hardware interface provides functions that read from and write to AAL1gator-32 device-registers. The hardware interface also provides a template for an ISR that the driver calls when the device raises a hardware interrupt. You must modify this function based on the interrupt configuration of your system.

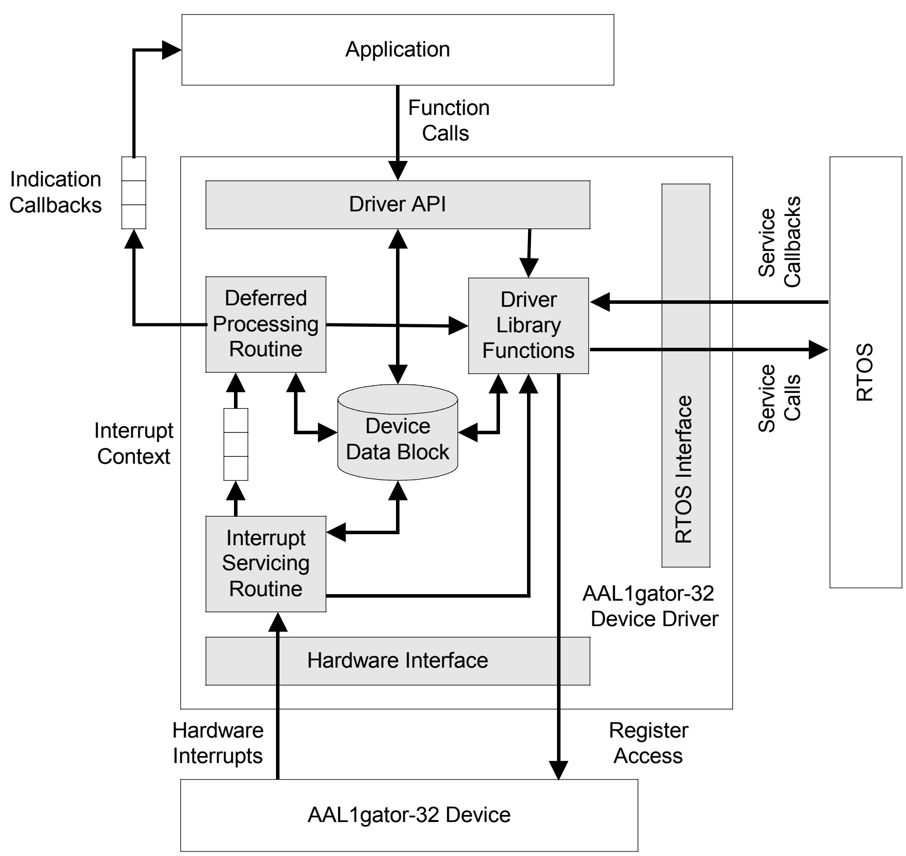
3.5 Main Components

Figure 3 illustrates the top level architectural components of the ALL1gator-32 device driver. This applies in both polled and interrupt driven operation. In interrupt driven mode, the Hardware interrupt is vectored to an application function that in turn calls the driver's ISR API `al3ISR(). The al3ISR` reads the device status, clears the cause(s) of the interrupt and creates a message that is sent to the DPR. In polled mode, the application makes a periodic call to `al3Poll()`, which in turn executes some of the functionality of the ISR (in order to read the Device status), and creates a message that is sent to the DPR.

The driver includes four main modules:

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

Figure 3: 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.

Device Data-Block Module

The Device Data-Block Module (DDB) stores context information about the AAL1gator-32 device, such as:

- Device state
- Control information
- Initialization vectors
- Callback function pointers
- Statistical counts

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

Module Data Block

The Module Data Block (MDB) and Module Status Block (MSB) are the top layer data structures. They are created by the AAL1gator-32 device driver to keep track of its initialization and operating parameters, modes and dynamic data. The MDB allocates via an RTOS call at the time the driver first initializes. The module also contains the MSB and all the Device Structures.

The Device Data Block (DDB) and Device Status Block (DSB) are contained in the MDB and initialized by the AAL1gator-32 Module for each Device that is registered. This keeps track of the Device's initialization and operating parameters, modes and dynamic data. There is a limit on the number of Device Blocks (Devices) available, and it is important to note that the USER sets that limit when the Module initializes.

Interrupt-Service Routine Module

The AAL1gator-32 driver provides an ISR called `al3ISR` 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 `al3ISR,` is system and RTOS dependent. Therefore, it is outside the scope of the driver.

See page 103 for a detailed explanation of the platform specific routines that must be supplied by the user.

Deferred-Processing Routine Module

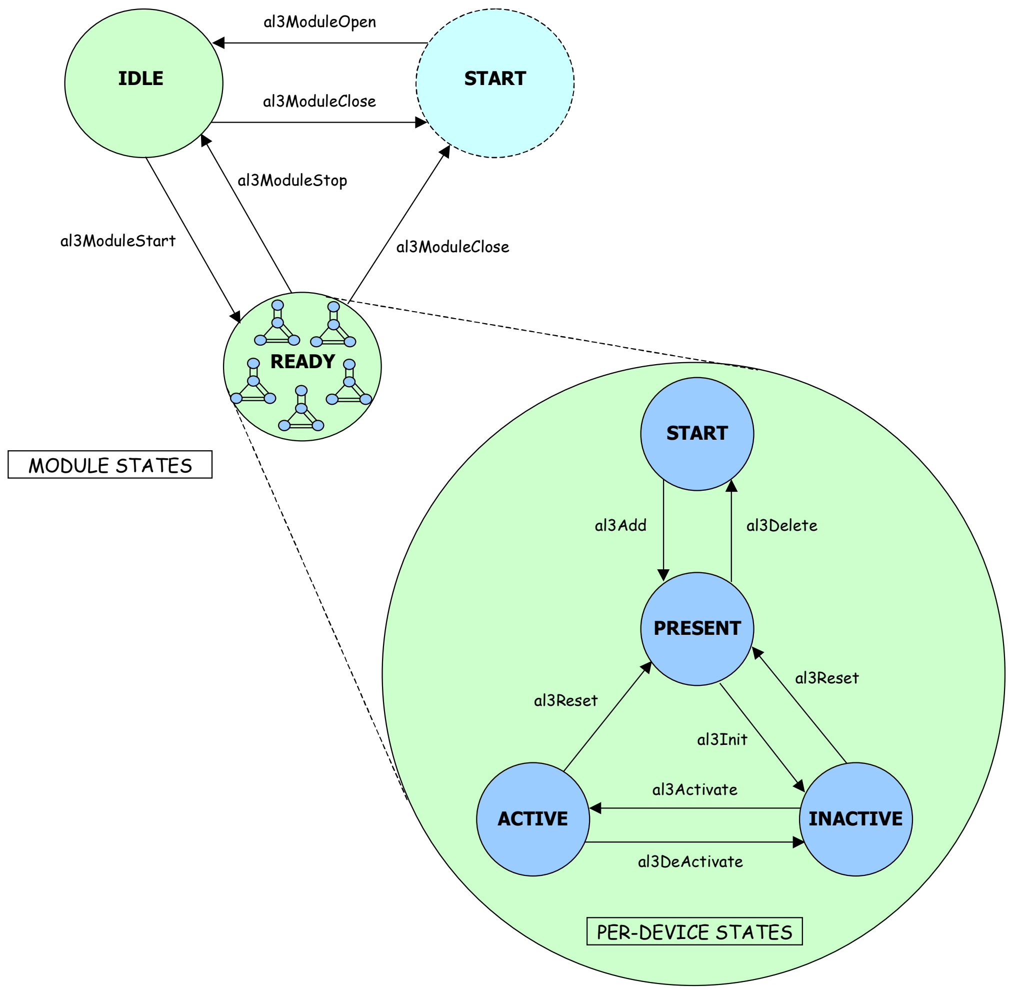
The Deferred-Processing Routine Module provided by the AAL1gator-32 driver (`al3DPR`) clears and processes interrupt conditions for the device. Typically a system specific function, which runs as a separate task within the RTOS, executes the DPR.

See page 104 for a detailed explanation of the DPR and interrupt-servicing model.

3.6 Software State Description

Figure 4 shows the software state diagrams for the AAL1gator-32 module and device(s) as maintained by the driver.

Figure 4: State Diagram



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

3.7 Module States

Start

The AAL1gator-32 driver Module is not initialized. The only API function accepted in this state is `al3ModuleOpen`. In this state the driver does not hold any RTOS resources (memory, timers, etc), has no running tasks, and performs no actions.

Idle

The AAL1gator-32 driver Module initializes successfully via the API function `al3ModuleOpen`. The Module Initialization Vector (MIV) has been validated, the Module Data Block (MDB) has been allocated and loaded with current data; the per-device data structures are allocated; and the RTOS has responded favorably to all the requests sent to it by the driver. The only API functions accepted in this state are `al3ModuleStart` and `al3ModuleClose`.

Ready

The normal operating state for the driver Module is "Ready" and can be entered by a call to `al3ModuleStart`. All RTOS resources allocate and the driver is ready for additional devices. The API functions accepted here for Module control are `al3ModuleStop`, and `al3ModuleClose`. The driver Module remains in this state while Devices are in operation. Add devices via `al3Add`.

3.8 Device States

The following is a description of the AAL1gator-32 per-device states.

Start

The AAL1gator-32 Device is not initialized. The only API function accepted in this state is `al3Add`. In this state the device is unknown by the driver and performs no actions.

Present

The AAL1gator-32 Device has been successfully added via the API function `al3Add`. 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 `al3Init` and `al3Delete`.

Active

The normal operating state for the Device(s) enters by a call to `al3Activate`. State changes initiate from the ACTIVE state via `al3DeActivate`, `al3Reset` and `al3Delete`.

Inactive

Enter "Inactive" via the `al3Init` or `al3DeActivate` function calls. In this state the Device remains configured but all data functions de-activate. This includes interrupts and Alarms, Status and Statistics functions. `al3Activate` will return the device to the ACTIVE state, while `al3Reset` or `al3Delete` will de-configure the Device. Queues are torn down.

3.9 Processing Flows

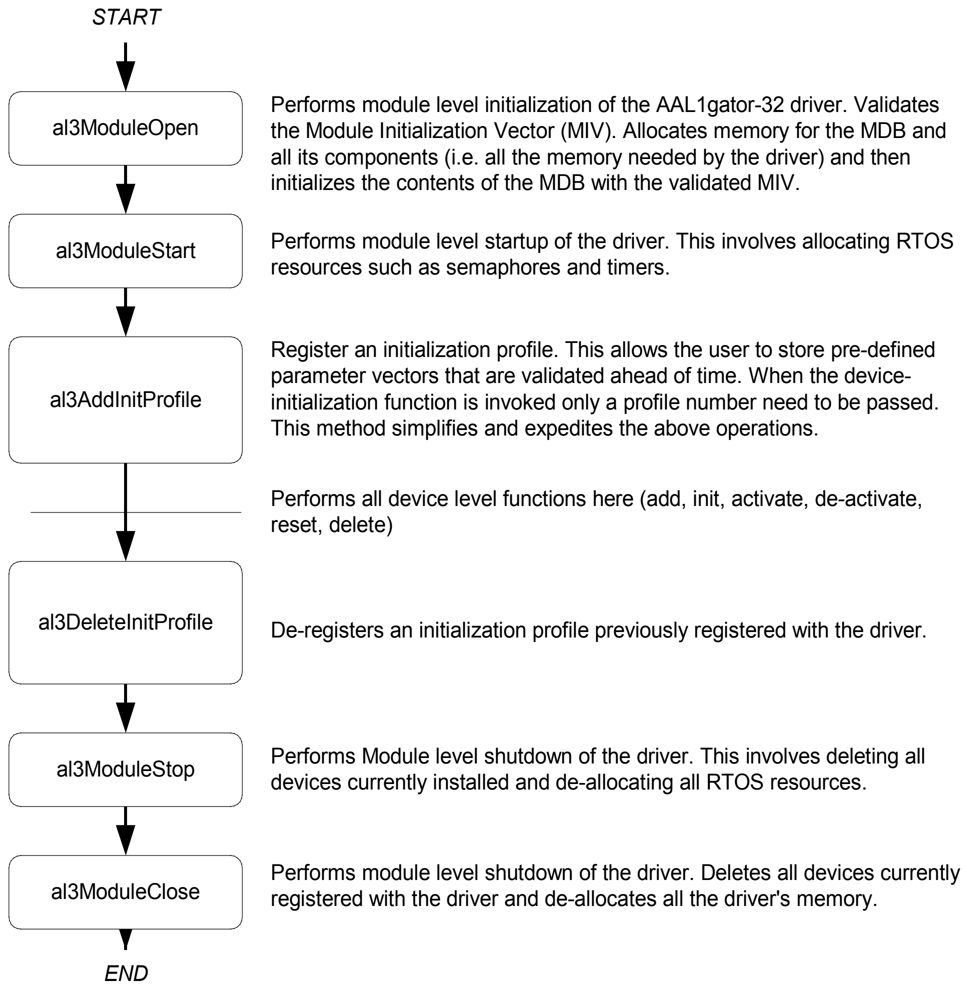
This section describes the main processing flows of the AAL1gator-32 driver modules.

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

Module Management

The following diagram illustrates the typical function call sequences that occur when initializing or shutting down the AAL1gator-32 driver module.

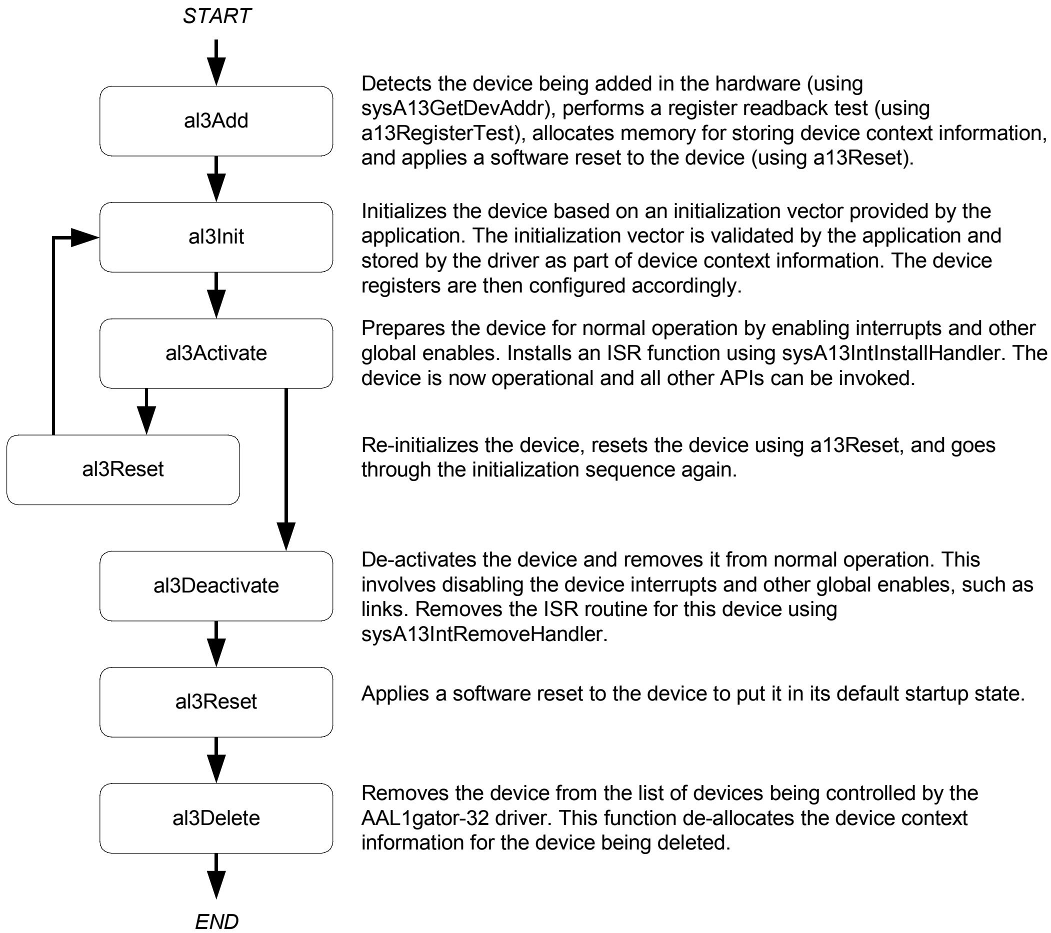
Figure 5: Module Management Flow Diagram



Device Management

The following figure shows the functions and process that the driver uses to add, initialize, re-initialize, and delete the AAL1gator-32 device.

Figure 6: Device Management Flow Diagram



3.10 Interrupt Servicing

The AAL1gator-32 driver services device interrupts using an interrupt service routine (ISR) that traps interrupts. It also uses a deferred interrupt-processing routine (DPR) that actually processes the interrupt conditions and clears them. This action lets the ISR execute quickly and exit. Most of the time-consuming processing of the interrupt conditions defers to the DPR by queuing the necessary interrupt-context information to the DPR task. The DPR function runs in the context of a separate task within the RTOS.

Note: Since the DPR task processes potentially serious interrupt conditions, you should set the DPR task's priority higher than the application task interacting with the AAL1gator-32 driver.

The driver provides system-independent functions, `al3ISR` and `al3DPR`. You must fill in the corresponding system-specific functions, `sysAl3ISR` and `sysAl3DPR`. The system-specific functions isolate the system-specific communication mechanism (between the ISR and DPR) from the system-independent functions, `al3ISR` and `al3DPR`.

Figure 7 illustrates the interrupt service model used in the AAL1gator-32 driver design.

Figure 7: Interrupt Service Model

Note: Instead of using an interrupt service model, you can use a polling service model in the AAL1gator-32 driver to process the device's event-indication registers (see page 31).

Calling al3ISR

An interrupt handler function, which is system dependent, calls `al3ISR`. 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 `(sysAl3IntHandler`) appears on page 103 .You can customize this example implementation to suit your needs.

The implemented interrupt handler (`sysAl3IntHandler`) installs in the interrupt vector table of the system processor. It calls when one or more AAL1gator-32 devices interrupt the processor. The interrupt handler subsequently calls `al3ISR` for each device in the active state.

The `al3ISR` function reads from the master interrupt-status register and the miscellaneous interrupt-status register of the AAL1gator-32. If a valid status bit is set, al3ISR then returns with the status information. Thereafter, `sysAl3IntHandler` function sends a message to the DPR task. The DPR task consists of the device handles of all the AAL1gator-32 devices that have had valid interrupt conditions.

Note: Normally you should store status information for deferred interrupt processing by implementing a message queue. The interrupt handler sends the status information to the queue by the `sysAl3IntHandler.`

Calling al3DPR

The `sysAl3DPRTask` function is a system specific function that runs as a separate task within the RTOS. You should set the DPR task's priority higher than the application task(s) interacting with the AAL1gator-32 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, `sysAl3DPRTask` calls the DPR (`al3DPR`).

The `al3DPR` then processes the status information and takes appropriate action based on the specific interrupt condition detected. The nature of this processing can differ from system to system. As a result, `al3DPR` calls different indication callbacks for different interrupt conditions.

Typically, you should implement these callback functions as simple message posting functions that post messages to an application task. However, you can implement the indication callback to perform processing within the DPR task context and return without sending any messages. In this case, ensure that the indication function does not call any API functions that change the driver's state, such as `al3Delete`. In addition, ensure that the indication function is non-blocking, as the DPR task executes while AAL1gator-32 interrupts are disabled. These callbacks can be customized to suit your system. See page 99 for example implementations of the callback functions.

Note: Since the al3`ISR` and `al3DPR` routines do not specify a communication mechanism, you have full flexibility in choosing a communication mechanism between the two. A convenient way to implement this communication mechanism is to use a message queue, which is a service that most RTOS' provide.

You must implement the two system specific functions, `sysAl3IntHandler` and `sysAl3DPRTask`. When the driver calls `sysAl3IntInstallHandler` for the first time, the driver installs `sysAl3IntHandler` in the interrupt vector table of the processor. The `sysAl3DPRTask` function is also spawned as a task during the first time invocation of `sysAl3IntInstallHandler`. The `sysAl3IntInstallHandler` function also creates the communication channel between `sysAl3IntHandler` and `sysAl3DPRTask`. This communication channel is most commonly a message queue associated with the `sysAl3DPRTask`.

Similarly, during removal of interrupts, the driver removes `sysAl3IntHandler` from the microprocessor's interrupt vector table and deletes the task associated with `sysAl3DPRTask.`

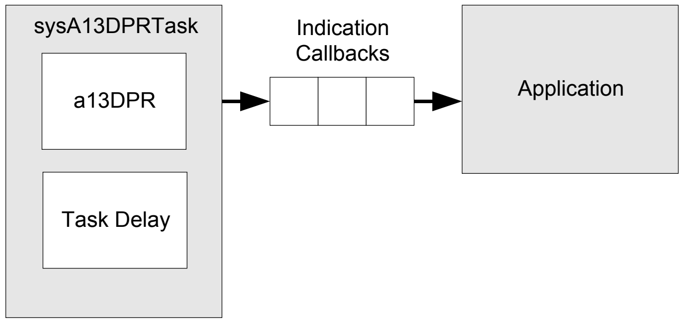
As a reference, this manual provides example implementations of the interrupt installation and removal functions on page 103. You can customize these prototypes to suit your specific needs.

3.11 Polling

Instead of using an interrupt service model, you can use a polling model in the AAL1gator-32 driver to process the device's event-indication registers.

The following figure illustrates the polling model used in the AAL1gator-32 driver design.

Figure 9: Polling Model



The polling code includes some system specific code (prefixed by "`sysAl3`"), which typically you must implement for your application. The polling code also includes some system independent code (prefixed by "`al3`") provided by the driver that does not change from system to system.

In polling mode, `sysAl3IntHandler` and `al3ISR` are not used. Instead, the application spawns a `sysAl3DPRTask` function as a task processor when the driver calls `sysAl3IntInstallHandler` for the first time.

In `sysAl3DPRTask`, the driver-supplied DPR (`al3DPR`) periodically calls active devices. The `al3DPR` reads from the master interrupt-status and miscellaneous interrupt-status registers of the AAL1gator-32. If some valid status bits are set, it processes the status information and takes appropriate action based on the specific interrupt condition detected.

The nature of this processing differs from system to system. Consequently, the DPR calls different indication callbacks for different interrupt conditions. You can customize these callbacks to fit your application's specific requirements. See page 99 for a description of these callback functions.

Similarly, during the removal of polling the driver removes the task associated with `sysAl3DPRTask` if the AAL1gator-32 devices do not activate.

3.12 Device Configuration

This section describes the various configuration operations performed by the driver.

AAL1 Channel Configuration

AAL1 channel configuration handles the provisioning and configuring of AAL1 channels inside the AAL1gator -32/-8/-4.

The API for this section of the driver consists of several functions in five groups.

The first is the channel provisioning group which consists of five functions. The first function, `al3ActivateChannelUnstr`, activates an AAL1 channel using a T1, E1, DS3, or E3 in unstructured mode. This AAL1 channel occupies that entire line. The next function, `al3ActivateChannelStr`, activates an AAL1 channel using one or more timeslots of a T1 or E1 line. Both `al3ActivateChannelStr` and `al3ActivateChannelUnstr` have enhanced versions. The enhanced versions offer extra configuration parameters such as max buffer size, cdvt, AAL0 mode, etc. The enhanced versions also allow the user to configure sequence number processing, conditioning and idle channel detection. If a NULL pointer passes for any of the channel configuration data structures, the `al3EnhancedActivateChannelStr` and `al3EnhancedActivateChannelUnstr` functions will use the default values for those data structures. These are the same defaults used when the non-enhanced Activate functions are invoked. The last function in this group, `al3DeActivateChannel`, deactivates an already provisioned AAL1 channel.

The second API group can add or remove timeslots of a T1 or E1 to or from an AAL1 channel. The function `al3AssociateChannel` adds timeslots to an AAL1 channel and the function `al3DisAssociateChannel` removes timeslots from an AAL1 channel.

The third API group is the SRTS (Synchronous Residual Time Stamp) group, which consists of two functions. The first function, `al3SRTSEnable`, enables SRTS and the second function, `al3SRTSDisable`, disables it.

Note: The AAL1gator-32/-8/-4 line level, not at the AAL1 channel level, controls the SRTS.

The fourth API group is the Conditioning group, and consists of four functions. The first function, `al3EnableTxCond`, enables conditioning in the Tx direction. The second, `al3DisableTxCond`, disables conditioning in the Tx direction. The third, `al3EnableRxCond`, enables conditioning in the Rx direction and the fourth, `al3DisableRxCond`, disables conditioning in the Rx direction.

The final API group is the Loopback group, which consists of two functions. The first function, `al3LpbkEnable`, puts an AAL1 channel in loopback mode and the second, `al3LpbkDisable`, takes the AAL1 channel out of loopback mode.

Finally, there is one function to configure clock generation for the lines on the AAL1gator-32/-8/-4device. The function, `al3GlobalClkConfig`, configures the adaptive filter size for the adaptive clock source method and the NCLK frequency for SRTS clock method.

Table 5 on page 41 shows the default values for global clock configuration

UTOPIA/Any-PHY Bus Configuration

UTOPIA/Any-PHY Bus configuration sets up the UTOPIA or Any-PHY bus on the AAL1gator-32.

The AAL1gator-32's UTOPIA/Any-PHY bus interface is capable of supporting an 8-bit or 16-bit wide bus, Level 1 or Level 2; as well as act as a Level 1 Bus Slave or Bus Master. On a Level 2 bus it can only act as a Bus Slave. Odd or Even Parity check can also be selected.

The UTOPIA/Any-PHY interface can be placed in remote loopback, so that all cells received by the AAL1gator-32/-8/-4 are looped back out the UTOPIA interface. Loopbacks are also possible on a per-VC basis towards the line.

The UTOPIA/Any-PHY interface has to identify which AAL1 Channel a particular VC is associated with. A mapping VPI:VCI to AAL1 Channel Queue method (Cell Header Interpretation) does this. The AAL1gator-32/-8/-4 devices support 3 methods for doing this.

Figure 8 illustrates the three methods.

Figure 8: Cell Header Interpretation

Method 1

11	10	9	8	7	6	5	4	3	2	1	0
Ignored											

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Ignored					A1SP		Data	Line			Queue MOD 32				

Method 2

11	10	9	8	7	6	5	4	3	2	1	0
Ignored											

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Ignored	A1SP		Data	Line			Queue MOD 32					Ignored			

Method 3
for UDF only

11	10	9	8	7	6	5	4	3	2	1	0
Ignored							A1SP		Line		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Ignored							Ignored					Ignored			

For Method 3, VCI is ignored, Queue Number 0 is assumed.

There is only one UTOPIA/Any-PHY related function in the API, `al3UtopiaConfig` configures the UTOPIA/Any-PHY interface according to the parameters passed to this function. There is a default initialization profile defined with the driver. The Initialization Profiles on page 41 include the UTOPIA/Any-PHY configuration.

RAM Interface Configuration

The AAL1gator-32/-8/-4 RAM interface supports one of either Synchronous SRAMs or ZBT RAMs. These RAMs store some AAL1gator-32/-8/-4 data structures. The AAL1gator-32/-8/-4 can also check Even or Odd parity on the RAMs' data buses and generate parity error interrupts to the microprocessor.

There are 2 RAM interfaces supported by the AAL1gator-32, and 1 RAM interface supported by both AAL1gator-8 and AAL1gator-4.

The API for this section of the AAL1gator-32 consists of only one function, `al3RAMConfig`, which passes the RAM configuration parameters and performs the necessary actions to configure the RAM interface according to the parameters.

RAM Initialization Profiles are included in Table 7, page 43

SBI Bus Configuration

The SBI (Scaleable Bandwidth Interconnect) Bus is a parallel bus used for transmitting TDM data between physical and data link layer devices. This interface is one of the 4 possible TDM side interfaces that the AAL1gator-32 supports. The other 3 are the Direct Line Low Speed, the H-MVIP bus and the Direct High Speed interface. The latter 2 require no software configuration. The AAL1gator-32's SBI interface allows a lot of flexibility in mapping SBI bus tributaries to AAL1gator-32 links. The SBI bus tributaries can be T1, E1, or DS3 payloads. The AAL1gator-32's SBI bus interface supports handling all these tributary types, there are however some limitations. All tributaries in an SPE (Synchronous Payload Envelope) must be of the same type and all AAL1gator-32 links in a link group must also be of the same type. There are 3 SPEs supported by the SBI bus, and there are 2 16-link link groups inside the AAL1gator-32. Other than these limitations, you are free to map the tributaries inside the SPEs on the SBI bus to any of the AAL1gator-32's thirty-two links.

The API for this section of the AAL1gator-32 consists of two functions, `al3SBIConfig` and `al3SBITribConfig`. The first configures the 3 SPEs and 2 Link Groups according to the parameters passed to it. The second configures the individual tributaries and maps them to one of the 32 AAL1gator-32 links.

SBI Bus configuration profiles are included in Table 8, page 43.

Note: The SBI bus is not supported by the AAL1gator-8 and AAL1gator-4 devices.

Direct Line Interface Configuration

The Direct Line Low Speed interface is a direct clock and data interface to a T1/E1 framer. This interface is one of the 4 possible TDM side interfaces that the AAL1gator-32 supports. The other 3 are the SBI bus, the H-MVIP bus and the Direct High Speed interface. The AAL1gator-8 and AAL1gator-4 do not support the SBI bus.

The AAL1gator-32 can support up to 16 direct low speed interfaces. The AAL1gator-8 can support up to 8 and the AAL1gator-4 can support up to 4.

The API for this section of the AAL1gator-32 consists of one function, `al3DirectConfig`, which configures the AAL1gator-32/-8/-4's low speed direct line interface based on the parameters passed to it.

Direct Line Interface configuration profiles are included in Table 11, page 44.

Alarms and Statistics

Most of the statistics for the AAL1gator-32 relate to the AAL1 channels provisioned through it. There are some statistics related to OAM cells that are per AAL1 SAR Processor (A1SP), although the Statistic Retrieval Functions for OAM statistics are per device. There are 4 A1SPs in the AAL1gator-32, and 1 A1SP in both the AAL1gator-8 and AAL1gator-4.

Software extends statistics to 32-bits from 16 bits. A periodic task achieves this as part of the Statistics Section. This task periodically polls all the hardware counters and updates their software counterparts respectively. The user adjusts the period of this task's execution. The task calls `sysAl3UpdateStats`.

Alarms and Statistics functions also generate alarms. SBI bus tributary alarms are enabled and disabled using `al3EnableSBIAlarm` and `al3DisableSBIAlarm`. Note: These functions are only valid for the AAL1gator-32 device.

This section also allows you to force a high-speed line configured for DS3 to generate cells with the AIS pattern using `al3EnableDS3AISCells` and `al3DisableDS3AISCells`.

3.13 Constants

The driver code uses the following Constants:

- `<AL3_ERROR_CODES>`: error codes used throughout the driver code, returned by the API functions and used in the global error number field of the MDB and DDB.
- `AL3_MAX_DEVICES`: defines the maximum number of devices supported by this driver. This constant must not be changed without a thorough analysis of the consequences to the driver code.
- `AL3_MAX_LINES`: defines the maximum number of lines per device. This constant must not be changed without a thorough analysis of the consequences to the driver code. (Limit should be 32 for AAL1gator-32, 8 for AAL1gator-8 and 4 for AAL1gator-4)
- `AL3_MAX_DIRECT`: defines the maximum number of direct low speed line interfaces per device. This constant must not be changed without a thorough analysis of the consequences to the driver code. (Limit should be 16 for AAL1gator-32, 8 for AAL1gator-8 and 4 for AAL1gator-4)

- `AL3_MAX_SPES`: defines the maximum number of synchronous payload envelopes (SPEs) on the SBI bus for each device. This constant must not be changed without a thorough analysis of the consequences to the driver code. (Limit should be 3).

- `AL3_MAX_TRIBS`: defines the maximum number of tributaries inside each SPE on the SBI bus for each device. This constant must not be changed without a thorough analysis of the consequences to the driver code. (The maximum allowed tribs within an SPE is 28 for T1, 21 for E1, and 1 for DS3).

- `AL3_MAX_LGRPS`: defines the maximum number of link groups (line groups) per device. This constant must not be changed without a thorough analysis of the consequences to the driver code. (Limit should be 2).

- `AL3_MAX_QUEUES`: defines the maximum number of AAL1 channel queues per device. This constant must not be changed without a thorough analysis of the consequences to the driver code. (Limit should be 1024 for AAL1gator-32, 256 for ALL1gator-8, and 128 for ALL1gator-4).

- `AL3_MDB_USER_SIZE`: defines the size in UINT4s of the User Defined field in the MDB.

- `AL3_DDB_USER_SIZE`: defines the size in UINT4s of the User Defined field in the DDB.

3.14 Variables

Although variables within the driver are not meant to be used by the application code, there are several that can used by the application code. They are to be considered read-only by the application.

- `al3MDB`: a global pointer to the Module Data Block (MDB). The MDB is only valid if the 'valid' flag is set.

- `errModule`: this MDB structure element is used to store an error code that specifies the reason for an API function's failure. The field is only valid when the function in question returns an `AL3_FAIL` value.

- `modState`: this MDB structure element stores the Module state.

- `modValid`: this MDB structure element indicates that the MDB contains valid data.

- `al3DDB[ ]`: An array of pointers to the individual Device Data Blocks. The DDB is only valid if the 'valid' flag is set and that the array of DDBs is in no particular order.

- `errDevice`: this MDB structure element stores an error code that specifies the reason for an API function's failure. The field is only valid when the function in question returns an `AL3_FAIL` value. The various Read/Write API functions store error codes here, as well as the Device Diagnostic functions.

- `devState`: this structure element stores the Device state.
- `devValid`: this structure element indicates that the DDB contains valid data.

4 DATA STRUCTURES

4.1 Data Structures

The following are the main data structures employed by the AAL1gator-32 driver.

AAL1 Channel Configuration Tables

The following tables detail the provisioning and configuring of AAL1 channels inside the AAL1gator-32.

Table 1: AAL1 Channel Enhanced Parameters Default Values: sAL3_CFG_CHAN_ENH

Field Name	Default Value	Field Type	Field Description
`partialFill`	0x00	UINT1	Partial Cell Fill Char
`rxMaxBuf`	Calculated Max Buffer Size	UINT2	Maximum Buffer Size
`rxCDVT`	0x10	UINT2	Cell Delay Variation Tolerance
`txSuppress`	Disabled	UINT1	Suppress TX (0-Disable, 1-Enable)
`maintnBitInteg`	Disable	UINT1	Maintain Bit Integrity through Underrun (0-Disable, 1-Enable)
`addQueOffset`	0x00	UINT1	Add Queue Scheduling Offset
`aal0Mode`	AAL1	UINT1	AAL0 Mode (0-AAL1, 1-AAL0)
`txGfc`	0x00	UINT1	GFC for TX VC
`txPti`	0x00	UINT1	PTI for TX VC
`txClp`	0x00	UINT1	CLP for TX VC
`txHec`	Calculated HEC	UINT1	HEC for TX VC

Table 2: AAL1 Channel Sequence Number Processing Default Values: sAL3_CFG_CHAN_SNP

Field Name	Default Value	Field Type	Field Description
`snpAlgorithm`	Fast	UINT1	RX SN Processing (0-Fast, 1-Robust, 2-Disabled)

Field Name	Default Value	Field Type	Field Description
insertDataMode	Insert AIS	UINT1	Format of Data Inserted for Lost Cells (0-Insert AIS, 1-Insert Conditioned Data, 2-Insert Old Data, 3-Insert Conditioned Data with MSB randomized)
insertCondCellData	0xFF	UINT1	Value of conditioned data inserted
maxInsert	6	UINT1	Maximum number of cells inserted [1-7 cells]
noStartDrop	Disabled	UINT1	Don't Drop First Cell (0-Disabled, 1-Enabled)

Table 3: AAL1 Channel Conditioning Default Values: sAL3_CFG_CHAN_COND

Field Name	Default Value	Field Type	Field Description
txCondMode	Both	UINT1	Conditioning Mode (0-Both, 1-Only Signaling, 2-Only Data)
txCondSig	0x0	UINT1	TX Side Conditioned Signaling Nibble
txCondData	0xFF	UINT1	TX Side Conditioned Data Byte
rxCondSig	0x0	UINT1	RX Side Conditioned Signaling Nibble
rxCondData	0xFF	UINT1	RX Side Conditioned Data Byte
rxCondMode	Conditioned Data	UINT1	RX Underrun Data (0-Conditioned Data, 1-Conditioned Data with MSB randomized, 2-Old Data)
rxSigMode	Freeze Signaling	UINT1	RX Underrun Signaling (0-Freeze Signaling, 1-Conditioned Signaling)

Table 4: AAL1 Channel Idle Channel Detection Default Values: sAL3_CFG_CHAN_IDET

Field Name	Default Value	Field Type	Field Description
idleDetEnable	Disable	UINT2	Enable Idle Channel Detection (0-Disable, 1-Enable [DBCES], 2-Enable [Non-DBCES])
rxCASPattern	0x00	UINT2	RX CAS Idle Pattern (CAS Matching)
txCASPattern	0x00	UINT2	TX CAS Idle Pattern (CAS Matching)
rxMask	0x00	UINT2	RX Mask (CAS or Processor Matching)
txMask	0x00	UINT2	TX Mask (CAS or Processor Matching)
idlePattern	0x00	UINT2	Idle Pattern (Pattern Matching)
patternMask	0x00	UINT2	Pattern Mask (Pattern Matching)

.

Table 5: Global Clock Default Initialization Profile Values: sAL3_DIV_CLK

Field Name	Default Value	Field Type	Field Description
adapFiltSize	0	UINT1	Adaptive Clock Filter Size (0->16)
nClkDivEnable	Disabled	UINT1	NCLK Division Enable (0-Disabled, 1-Enabled)
nClkDivFactor	0	UINT1	NCLK Division Factor [nClkDivFactor+2] (0->7)

UTOPIA/Any-PHY Bus Configuration Table

The following tables detail setting up the UTOPIA or Any-PHY bus on the AAL1gator -32/-8/-4.

Table 6: UTOPIA/Any-PHY Default Initialization Profile Values: sAL3_DIV_UTOPIA

Field Name	Default Value	Field Type	Field Description
enable	Enabled	UINT1	UTOPIA/Any-PHY bus enable (0-Disabled, 1-Enabled)

Field Name	Default Value	Field Type	Field Description
vpiVciMapping	Method 1	UINT1	VCI range used for mapping to AAL1 Channel Queue numbers (0-Method 1, 1-Method 2, 2-Method 3)
loopbk	None	UINT1	UTOPIA/Any-PHY loopback (0-None, 1-Remote, 2-VCI Remote)
lpbkVci	0x1111	UINT1	UTOPIA/Any-PHY loopback 16bit VCI
srcAnyPhyMode	UTOPIA	UINT1	Source Any-PHY Mode (0-UTOPIA, 1-Any-PHY)
srcBusWidth	16bit	UINT1	Source UTOPIA/Any-PHY bus width (0-8bit, 1-16bit)
srcUtopMode	PHY	UINT1	Source UTOPIA bus mode (0-ATM, 1-PHY)
srcSlaveAddr	0x0000	UINT1	Source UTOPIA/Any-PHY 16-bit slave address
srcParity	Odd	UINT1	Source UTOPIA/Any-PHY parity (0-Odd, 1-Even)
srcCSMode	Disabled	UINT1	Source Any-PHY Chip Select Mode (0-Disabled, 1-Enabled)
snkAnyPhyMode	UTOPIA	UINT1	Sink Any-PHY Mode (0-UTOPIA, 1-Any-PHY)
snkBusWidth	16bit	UINT1	Sink UTOPIA/Any-PHY bus width (0-8bit, 1-16bit)
snkUtopMode	PHY	UINT1	Sink UTOPIA bus mode (0-ATM, 1-PHY)
snkSlaveAddr	0x0000	UINT1	Sink UTOPIA/Any-PHY 16-bit slave address
snkParity	Odd	UINT1	Sink UTOPIA/Any-PHY parity (0-Odd, 1-Even)
snkCSMode	Disabled	UINT1	Sink Any-PHY Chip Select Mode (0-Disabled, 1-Enabled)

RAM Interface Configuration Table

The following table depicts the default RAM configuration stored in the default initialization profile.

Table 7: RAM Default Initialization Profile Values: sAL3_DIV_RAM

Field Name	Default Value	Field Type	Field Description
`protocol`	SSRAM	UINT1	SRAM protocol (0-SSRAM, 1-ZBT)
`parity`	Odd	UINT1	SRAM parity type (0-Odd, 1-Even)

SBI Bus Configuration Tables

The following tables depict the default SBI SPE and Link Group configuration stored in the default initialization profile.

Table 8: SBI Bus Default Initialization Profile Values: sAL3_DIV_SBI

Field Name	Default Value	Field Type	Field Description
`mapEnable`	Mapping Enabled	UINT1	Tributary mapping (0-Forced, 1-Forced on Extract Only, 2-Forced on Insert Only, 3-Mapping Enabled)
`clkMaster`	Use Trib Cfg Setting	UINT1	Force Clock Mastering (0-Use Trib Cfg setting, 1-Force)
`busMaster`	Disabled	UINT1	Bus Master (0-Disabled, 1-Enabled)
`twoC1FPEnable`	Disabled	UINT1	Separate C1FP for Insert and Extract bus (0-Disabled, 1-Enabled)
`insBusParity`	Odd	UINT1	Insert Bus parity (0-Even, 1-Odd)
`extBusParity`	Odd	UINT1	Extract Bus parity (0-Even, 1-Odd)
`page`	1	UINT1	Active Configuration Page (0-Page 1, 1-Page 2)
`speCfg` `[AL3_MAX_SPES]`	See below	sAL3_DIV_SBI_SPE	SPE (Synchronous Payload Envelope) configuration
`linkGrpCfg` `[AL3_MAX_LGRPS]`	See below	sAL3_DIV_SBI_LGRP	AAL1gator-32 Link Group configuration

Table 9: SBI Bus SPE Default Initialization Profile Values: sAL3_DIV_SBI_SPE

Field Name	Default Value	Field Type	Field Description
`speType`	DS1	UINT1	SPE type (0-DS1, 1-E1, 2-DS3)
`speEnable`	Enable	UINT1	SPE enable (0-Disable, 1-Enable)
`speSync`	Asynchronous	UINT1	SPE sync (0-Asynchronous, 1-Synchronous)

Table 10: SBI Bus Link Group Default Initialization Profile Values: sAL3_DIV_SBI_LGRP

Field Name	Default Value	Field Type	Field Description
`lgrpType`	DS1	UINT1	Link Group type (0-DS1, 1-E1, 2-DS3)
`clkKill`	Disable	UINT1	Clock Kill (0-Disable, 1-Enable)

Direct Line Interface Configuration Table

The following table depicts the default Direct Low Speed configuration stored in the default initialization profile.

Table 11: Direct Line Default Initialization Profile Values: sAL3_DIV_DIRECT

Field Name	Default Value	Field Type	Field Description
`syncMode`	Frame	UINT1	Direct Line Sync Mode (0-Frame, 1-MultiFrame)
`mvipMode`	Disable	UINT1	MVIP Mode (0-Disable, 1-Enable)

4.2 Structures Passed by the Application

The application defines these structures and passes them by reference to functions within the driver.

Module Initialization Vector

Passed via the `al3ModuleOpen` call, this structure contains all the information needed by the driver to initialize and connect to the RTOS.

- maxDevs informs the Driver how many Devices will be operating concurrently during this session. The number calculates the amount of memory allocated to the driver. Memory is allocated in the al3ModuleOpen call. The maximum value passed is AL3_MAX_DEVS.

- autoStart tells the Driver to automatically start connecting to the RTOS. If the flag is ZERO, the Module will be initialized only, and the application will have to call al3ModuleStart at a later time.

- diagOnInit is a flag that tells the Driver to run diagnostic routines when the device initializes. If the flag is ZERO, the Module will be initialized only, and the application will have to call the diagnostic routines directly.

Table 12: Module Initialization Vector: sAL3_MIV

Field Name	Field Type	Field Description
pMDB	INT4 *	Pointer to MDB
maxDevs	UINT2	Maximum number of devices supported during this session
maxInitProfs	UINT2	Maximum number of initialization profiles
autoStart	BOOLEAN	If non-zero, al3ModuleStart is called internally
diagOnInit	BOOLEAN	If non-zero, diagnostic routines will be executed when every device is initialized.

Initialization Profile

Initialization Profile Top-Level Structure

Passed via the al3SetInitProfile and or al3Init call, this structure contains all the information needed by the driver to initialize and activate an AAL1gator-32 device.

- autoActivate tells the Driver to activate the Device being initialized. If the flag is ZERO, the Device will be initialized but left inactive, and the application will have to call al3Activate at a later time.

Table 13: Initialization Profile: sAL3_DIV

Field Name	Field Type	Field Description
modeHS	BOOLEAN	High-Speed Mode
autoActivate	BOOLEAN	Indicates that the device should be initialized directly to the ACTIVE state by calling al3Activate internally

Field Name	Field Type	Field Description
cfgLINE[AL3_MAX_A1SPS][AL3_LINES_PER_A1SP]	sAL3_DIV_LINE	AAL1gator-32 Line configuration block
cfgCLK[AL3_MAX_A1SPS]	sAL3_DIV_CLK	AAL1gator-32 Global Clock configuration block
cfgUtopia	sAL3_DIV_UTOPIA	UTOPIA configuration block
cfgRam	sAL3_DIV_RAM	RAM configuration block
cfgSbi	sAL3_DIV_SBI	SBI Bus configuration block
cfgTRIB[AL3_SIZE_SPE][AL3_SIZE_TRIB]	sAL3_DIV_TRIB	SBI Bus Tributary configuration block
cfgDirect[AL3_SIZE_DIRECT]	sAL3_DIV_DIRECT	Direct Line configuration block
modeISR	AL3_ISR_MODE	Indicates the type of ISR/polling to do
cbackA1SP	sAL3_CBACK	Address for the callback function for A1SP Events
cbackUtopia	sAL3_CBACK	Address for the callback function for UTOPIA bus Events
cbackRAM	sAL3_CBACK	Address for the callback function for RAM Events
cbackSBI	sAL3_CBACK	Address for the callback function for SBI bus Events

Initialization Profile Sub-Structures

Initialization Profile Sub-Structures appear in the initialization profile tables below.

Table 14: AAL1 Line Configuration: sAL3_DIV_LINE

Field Name	Field Type	Field Description
lowCDV	UINT1	Low CDV (0-Disable [frame based scheduling], 1-Enable [byte based scheduling])
refValEnable	UINT1	Enable Reference Value generation (0-OFF, 1-ON)
t1Mode	UINT1	Mode (0-E1, 1-T1)
sigType	UINT1	Signaling (0-E1 with E1 signaling, 1-E1 with T1 signaling) [For E1 Line type only]
hiResClkSynth	UINT1	Hi Resolution Clock Synthesis (0-Disable, 1-Enable)
mfAlign	UINT1	Multiframe Align Enable (0-Disable, 1-Enable)

Field Name	Field Type	Field Description
genSync	UINT1	Generate TL_SYNC (0-Receive, 1-Generate)
txClkSrc	UINT1	Tx Clock Source (0-External, 1-Looped, 2-Nominal, 3-SRTS, 4-Adaptive, 5-Externally Controlled, 6-Common, 7-Common w/TL_SIG)
rxClkSrc	UINT1	Rx Clock Source (0-External, 1-Common)
frameType	UINT1	Frame Type (0-Unused, 1-SDF_FR, 2-UDF, 3-SDF-MF)
srtsEnable	UINT1	Enable SRTS (0-OFF, 1-ON)
srtsCDVT	UINT1	SRTS CDVT (if enabled)
shiftCAS	UINT1	CAS nibble shifting (0-coincident with the second nibble of data, 1-coincident with the first nibble of data)
iDetCfg	UINT1	Idle Channel Detection Configuration: 0 - Disabled, 1 - Processor, 2 - CAS Matching, 3 - Pattern Matching
iDetIntvlLen	UINT1	Interval Length

Table 15: Global Clock Configuration: sAL3_DIV_CLK

Field Name	Field Type	Field Description
adapFiltSize	UINT1	Adaptive Clock Filter Size (0->16)
nClkDivEnable	UINT1	NCLK Division Enable (0-Disabled, 1-Enabled)
nClkDivFactor	UINT1	NCLK Division Factor [nClkDivFactor+2] (0->7)

Table 16: UTOPIA/Any-PHY Configuration: sAL3_DIV_UTOPIA

Field Name	Field Type	Field Description
enable	UINT1	UTOPIA/Any-PHY bus enable (0-Disabled, 1-Enabled)
vpiVciMapping	UINT1	VPI:VCI mapping to AAL1 Channel Queue numbers method (0-Method 1, 1-Method 2, 2-Method 3 [for all UDF only]) [Please see Theory of Operations for Mapping method explanations]
loopbk	UINT1	UTOPIA/Any-PHY loopback (0-None, 1-Remote, 2-VCI Remote)
lpbkVci	UINT1	UTOPIA/Any-PHY loopback 16bit VCI
srcAnyPhyMode	UINT1	Source Any-PHY Mode (0-UTOPIA, 1-Any-PHY)
srcBusWidth	UINT1	Source UTOPIA/Any-PHY bus width (0-8bit, 1-16bit)

Field Name	Field Type	Field Description
srcUtopMode	UINT1	Source UTOPIA bus mode (0-L1 Master, 1-L1 Slave, 2-L2 Single Address Slave, 3-L2 Multiple Address Slave)
srcSlaveAddr	UINT1	Source UTOPIA/Any-PHY 16-bit slave address
srcParity	UINT1	Source UTOPIA/Any-PHY parity (0-Odd, 1-Even)
srcCSMode	UINT1	Source Any-PHY Chip Select Mode (0-Disabled, 1-Enabled)
snkAnyPhyMode	UINT1	Sink Any-PHY Mode (0-UTOPIA, 1-Any-PHY)
snkBusWidth	UINT1	Sink UTOPIA/Any-PHY bus width (0-8bit, 1-16bit)
snkUtopMode	UINT1	Sink UTOPIA bus mode (0-L1 Master, 1-L1 Slave, 2-L2 Single Address Slave, 3-L2 Multiple Address Slave)
snkSlaveAddr	UINT1	Sink UTOPIA/Any-PHY 16-bit slave address
snkParity	UINT1	Sink UTOPIA/Any-PHY parity (0-Odd, 1-Even)
snkCSMode	UINT1	Sink Any-PHY Chip Select Mode (0-Disabled, 1-Enabled)

Table 17: RAM Configuration: sAL3_DIV_RAM

Field Name	Field Type	Field Description
protocol	UINT1	SRAM protocol (0-SSRAM, 1-ZBT)
parity	UINT1	SRAM parity type (0-Odd, 1-Even)

Table 18: SBI Bus Configuration: sAL3_DIV_SBI

Field Name	Field Type	Field Description
mapEnable	UINT1	Tributary mapping (0-Forced, 1-Mapping Enabled)
clkMaster	UINT1	Force Clock Mastering (0-Use Trib Cfg setting, 1-Force)
busMaster	UINT1	Bus Master (0-Disabled, 1-Enabled)
twoC1FPEnable	UINT1	Separate C1FP for Insert and Extract bus (0-Disabled, 1-Enabled)
insBusParity	UINT1	Insert Bus parity (0-Even, 1-Odd)
extBusParity	UINT1	Extract Bus parity (0-Even, 1-Odd)
page	UINT1	Active Configuration Page (0-Page 1, 1-Page 2)

Field Name	Field Type	Field Description
speCfg [AL3_MAX_SPES]	sAL3_CFG_SPE	SPE (Synchronous Payload Envelope) configuration
linkGrpCfg [AL3_MAX_LGRPS]	sAL3_CFG_LGRP	AAL1gator-32 Link Group configuration

Table 19: SBI Bus SPE Configuration: sAL3_DIV_SPE

Field Name	Field Type	Field Description
speType	UINT1	SPE type (0-DS1, 1-E1, 2-DS3)
speEnable	UINT1	SPE enable (0-Disable, 1-Enable)
speSync	UINT1	SPE sync (0-Asynchronous, 1-Synchronous)

Table 20: SBI Bus Link Group Configuration: sAL3_DIV_LGRP

Field Name	Field Type	Field Description
lgrpType	UINT1	Link Group type (0-DS1, 1-E1, 2-DS3)
clkKill	UINT1	Clock Kill (0-Disable, 1-Enable)

Table 21: SBI Bus Tributary Configuration: sAL3_DIV_TRIB

Field Name	Field Type	Field Description
link	UINT1	Link (line number) associated with this trib
enable	UINT1	Tributary Enable (0-Disabled, 1-Enabled, 2-Only Insert Enabled, 3-Only Extract Enabled)
type	UINT1	Tributary type (0-Structured w/CAS, 1-Structured w/o CAS, 2-Unstructured)
insClkMaster	UINT1	Tributary Clock Master on Insert Bus (0-Clock slave, 1-Clock master)
extClkMaster	UINT1	Tributary Clock Master on Extract Bus (0-Clock slave, 1-Clock master)
extClkMode	UINT1	Tributary Clock Mode for Extract Bus (0-EXT_CKCTL, 1-ClkRate, 2-Phase)
insSynchMode	UINT1	Tributary Synch for Insert Bus (0-Float, 1-Locked)

Table 22: Direct Line Configuration: sAL3_DIV_DIRECT

Field Name	Field Type	Field Description
syncMode	UINT1	Direct Line Sync Mode (0-Frame, 1-MultiFrame)
mvipMode	UINT1	MVIP Mode (0-Disable, 1-Enable)

AAL1 Channel Configration Parameters

Table 23: AAL1 Standard Channel Configuration: sAL3_CFG_CHAN

Field Name	Field Type	Field Description
txVpi	UINT2	VPI for TX VC
txVci	UINT2	VCI for TX VC
rxVpi	UINT2	VPI for RX VC
rxVci	UINT2	VCI for RX VC
rxCheckParity	UINT1	Parity Check (0-Off, 1-On)
suppressSignaling	UINT1	Suppress Signaling (0-Off, 1-On) [for SDF-MF only]

Table 24: AAL1 Enhanced Channel Configuration: sAL3_CFG_CHAN_ENH

Field Name	Field Type	Field Description
partialFill	UINT1	Partial Cell Fill Char
rxMaxBuf	UINT2	Maximum Buffer Size
rxCDVT	UINT2	Cell Delay Variation Tolerance
txSuppress	UINT1	Suppress TX (0-Disable, 1-Enable)
maintnBitInteg	UINT1	Maintain Bit Integrity through Underrun condition (0-Disable, 1-Enable)
addQueOffset	UINT1	Add Queue Scheduling Offset
aal0Mode	UINT1	AAL0 Mode (0-AAL1, 1-AAL0)
txGfc	UINT1	GFC for TX VC
txPti	UINT1	PTI for TX VC
txClp	UINT1	CLP for TX VC

Field Name	Field Type	Field Description
txHec	UINT1	HEC for TX VC

Table 25: AAL1 Channel Sequence Number Processing Configuration: sAL3_CFG_CHAN_SNP

Field Name	Field Type	Field Description
snpAlgorithm	UINT1	RX SN Processing (0-Fast, 1-Robust, 2-Disabled)
insertDataMode	UINT1	Format of Data Inserted for Lost Cells (0-Insert AIS, 1-Insert Conditioned Data, 2-Insert Old Data, 3-Insert Conditioned Data with MSB randomized)
insertCondCellData	UINT1	Value of conditioned data inserted
maxInsert	UINT1	Maximum number of cells inserted [1-7 cells]
noStartDrop	UINT1	Don't Drop First Cell (0-Disabled, 1-Enabled)

Table 26: AAL1 Channel Conditioning Configuration: sAL3_CFG_CHAN_COND

Field Name	Field Type	Field Description
txCondMode	UINT1	Conditioning Mode (0-Both, 1-Only Signaling, 2-Only Data)
txCondSig	UINT1	TX Side Conditioned Signaling Nibble
txCondData	UINT1	TX Side Conditioned Data Byte
rxCondSig	UINT1	RX Side Conditioned Signaling Nibble
rxCondData	UINT1	RX Side Conditioned Data Byte
rxCondMode	UINT1	RX Underrun Data (0- Conditioned Data, 1-Conditioned Data with MSB randomized, 2-Old Data)
rxSigMode	UINT1	RX Underrun Signaling (0-Freeze Signaling, 1-Conditioned Signaling)

Table 27: AAL1 Channel Idle Channel Detection Configuration: sAL3_CFG_CHAN_IDET

Field Name	Field Type	Field Description
idleDetEnable	UINT2	Enable Idle Channel Detection (0-Disable, 1-Enable [DBCES], 2-Enable [Non-DBCES])
rxCASPattern	UINT2	RX CAS Idle Pattern (CAS Matching)
txCASPattern	UINT2	TX CAS Idle Pattern (CAS Matching)
rxMask	UINT2	RX Mask (CAS or Processor Matching)
txMask	UINT2	TX Mask (CAS or Processor Matching)
idlePattern	UINT2	Idle Pattern (Pattern Matching)
patternMask	UINT2	Pattern Mask (Pattern Matching)

Counter Specification

Table 28: Counter Specification: sAL3_CNTR_SPEC

Field Name	Field Type	Field Description
rdata	UINT4	Read Data
wdata	UINT4	Write Data
aspNum	UINT2	A1SP Number (Not required if queId is specified)
lineNum	UINT2	Line Number (Not required if queId is specified)
queNum	UINT2	Queue Number (Not required if queId is specified)
queId	sAL3_QID	Queue Id
type	AL3_CNTR_TYPE	Counter Type To Return

Sticky Bit Error Word

Table 29: Sticky Bit Error Word: sAL3_STICKY

Field Name	Field Type	Field Description
transfer	BOOLEAN	Transferring data to the sticky bits
cellRcvd	BOOLEAN	A Cell was received
dbcesBitMaskErr	BOOLEAN	There was a parity error in the DBCES Bit Mask

Field Name	Field Type	Field Description
transfer	BOOLEAN	Transferring data to the sticky bits
ptrRuleErr	BOOLEAN	There was a violation of a pointer generation rule
allocTblBlank	BOOLEAN	A cell was dropped because of a blank allocation table
ptrSearch	BOOLEAN	A cell was dropped because a valid pointer has not yet been found
forcedUndr	BOOLEAN	A cell was dropped because a forced underrun condition exists
snCellDrop	BOOLEAN	A cell was dropped in accordance with the "SN Algorithm"
ptrRcvd	BOOLEAN	A pointer was received
ptrParErr	BOOLEAN	A cell was received with a pointer parity error
srtsResume	BOOLEAN	An SRTS resume has occurred
srtsUndrn	BOOLEAN	A cell was received while the SRTS queue was in underrun
resume	BOOLEAN	A resume has occurred; a valid cell was received and stored into the buffer
ptrMismatch	BOOLEAN	A cell was dropped because of a pointer mismatch
overrun	BOOLEAN	A cell was dropped due to overrun
underrun	BOOLEAN	A cell was received while this queue was in underrun

ISR Enable/Disable Mask

Passed via the `al3MaskSet`, `al3MaskGet` and `al3MaskClear` calls, ISR Enable/Disable Mask contains all the information needed by the driver to enable and disable any of the interrupts in the AAL1gator-32.

Note: For all interrupts in the ISR mask, there are masks that allow you to mask out a whole group of interrupts. If you specify "Enable Some" for these you can mask the interrupts individually.

Table 30: ISR Mask: sAL3_MASK

Field Name		Field Type	Field Description
ram	ram1	UINT1	RAM 1 parity
	ram2	UINT1	RAM 2 parity

Field Name			Field Type	Field Description
alarmSBI			UINT1	SBI Alarm
exSBI	sync		UINT1	Extract bus DC, SBIIP or C1FP Error
	fifoOvr		UINT1	Extract FIFO Overrun
	fifoUdr		UINT1	Extract FIFO Underrun
	parity		UINT1	Extract Bus Parity Error
inSBI	sync		UINT1	Insert bus DC, SBIIP or C1FP Error
	fifoOvr		UINT1	Insert FIFO Overrun
	fifoUdr		UINT1	Insert FIFO Underrun
a1sp [AL3_MAX_A1SPS]	master		UINT1	A1SP in MASTER register
	oam		UINT1	A1SP OAM
	talpFifoFull		UINT1	A1SP TALP FIFO Full
	frmAdvFifoFull		UINT1	A1SP Frame Advance FIFO Full
	rxStatFifoFull		UINT1	A1SP RX Status FIFO Full
	rxStatFifoNotEmpty		UINT1	A1SP RX Status FIFO Not Empty
	txIdleFifoFull		UINT1	A1SP TX Idle State FIFO Full
	txIdleFifoNotEmpty		UINT1	A1SP TX Idle State FIFO Not Empty
	sticky	cellRx	UINT1	Cell Received Sticky Bit
		dbces	UINT1	DBCES Bit Mask Error Sticky Bit
		ptrRule	UINT1	Pointer Rule Error Sticky Bit
		allocTbl	UINT1	Allocation Table Blank Sticky Bit
		ptrSrch	UINT1	Pointer Search Sticky Bit
		fRedUndr	UINT1	Forced Underrun Sticky Bit
		snCellDrop	UINT1	SN Cell Drop Sticky Bit
		ptrRx	UINT1	Pointer Received Sticky Bit
		ptrParity	UINT1	Pointer Parity Error Sticky Bit
		srtsRes	UINT1	SRTS Resume Sticky Bit

Field Name			Field Type	Field Description
		srtsUndr	UINT1	SRTS Underrun Sticky Bit
		res	UINT1	Resume Sticky Bit
		ptrMis	UINT1	Pointer Mismatch Sticky Bit
		ovr	UINT1	Overrun Sticky Bit
		undr	UINT1	Underrun Sticky Bit
	rxStatResync		UINT1	Rx Line entered a resync state
	txStatResync		UINT1	Tx Line entered a resync state
	rxStatBitmask		UINT1	DBCES exited Underrun
	rxStatUdrExit		UINT1	QUEUE exited Underrun
	rxStatUdrEnter		UINT1	QUEUE entered Underrun
	rxStatQueError		UINT1	QUEUE Error (Sticky Bits)
utopia	parity		UINT1	UTOPIA Parity
	runtCell		UINT1	UTOPIA Runt cell
	transErr		UINT1	UTOPIA Cell Transfer Error
	fifo		UINT1	UTOPIA FIFO Full
	lpbkFifo		UINT1	UTOPIA Loopback FIFO Full

4.3 Structures in the Driver's Allocated Memory

Structures located in the Driver's Allocated Memory are used by the driver, and are part of the context memory allocated when the driver is opened.

Module Data Block

The MDB is the top-level structure for the Module, containing configuration data about the Module level code, and pointers to configuration data about Device level codes.

Table 31: Module Data Block: sAL3_MDB

Field Name	Field Type	Field Description
errModule	INT4	Module based error code

Field Name	Field Type	Field Description
maxDevs	UINT2	Maximum number of devices that can be added
maxDIVs	UINT2	Maximum number of DIVs (profiles)
autoStart	BOOLEAN	Automatic start on Open
diagOnInit	BOOLEAN	Run diagnostics during the al3Init()
modState	UINT2	Current module state
modValid	UINT2	This structure is valid
numDevs	UINT2	Current number of added devices
numDIVs	UINT2	Current number of Added Profiles (DIVs)
timerModule	void *	(pointer to) Timer ID variable
semModule	void *	(pointer to) Semaphore ID variable
bufOK	BOOLEAN	sysAl3BufferStart succeeded
isrOK	BOOLEAN	sysAl3ISRHandlerInstall succeeded
appMDB	BOOLEAN	MDB memory was passed by the application
updActive	BOOLEAN	Statistics are being gathered.
vpiModeOK [AL3_MAX_DEVICES]	BOOLEAN	Accumulation of LINE modes
user [AL3_MDB_USER_SIZE]	UINT4	Extra space for use by the Application
modMSB	sAL3_MSB	Module status block
divAddr	sAL3_DIV *	Address of the DIVs in the MDB
pDIV [AL3_MAX_DIVS]	sAL3_DIV *	DIV pointer array
ddbAddr	sAL3_DDB *	Address of the DDBs in the MDB
pDDB [AL3_MAX_DEVICES]	sAL3_DDB *	DDB pointer array

Device Data Block

The DDB is the top-level structure for each Device, containing configuration data about the Device level code, and pointers to configuration data about Device level sub-blocks.

Table 32: Device Data Block: sAL3_DDB

Field Name		Field Type	Field Description
errDevice		INT4	Global return code for Device functions
baseAddr		UINT2*	Base address of the Device
usrCtxt		void *	Application-specific use
autoInit		BOOLEAN	Copy of flag from profile
divNum		UINT2	Profile Number to be used for Initialization
modeISR		AL3_ISR_MODE	Indicates the current type of ISR/Polling
cbackRAM		sAL3_CBACK	RAM Events
cbackSBI		sAL3_CBACK	SBI Bus Events
cbackA1SP		sAL3_CBACK	A1SP Events
cbackUtopia		sAL3_CBACK	UTOPIA Bus Events
numQUE		UINT2	Maximum Number of Queues for the Device
numA1SP		UINT2	Maximum Number of A1SPs for the Device
numLINE		UINT2	Maximum Number of Lines for the Device
numDIRECT		UINT2	Maximum Number of Low Speed Lines for the Device
ramEndAddr		UINT4	SRAM ending address for the device
devState		UINT2	Current state of the Device
devValid		UINT2	Structure is Valid
devNum		UINT2	Index into al3DDB[]
revision		UINT2	Device Revision Data
lineMode		UINT2	Current Line Mode
hwFail		BOOLEAN	HW Failure Flag
activePageEXSBI		UINT2	Current 'in-use' page for EXSBI Block
activePageINSBI		UINT2	Current 'in-use' page for INSBI Block
sbiLinkMap [AL3_MAX_LINES]	speNum	UINT1	SPE number
	tribNum	UINT1	Tributary number
	insPage	UINT1	INSBI page
	extPage	UINT1	EXSBI page
statUpdateTime		UINT4	Tracks STATS updates
statUpdatePeriod		UINT4	Tracks STATS updates
txOAMCount [AL3_MAX_A1SPS]		UINT2	per A1SP TX OAM Cell Counter

Field Name	Field Type	Field Description
user [AL3_DDB_USER_SIZE]	UINT4	USER data area
a1sp [AL3_MAX_A1SP]	sAL3_ADB	A1SP Structures (above)
mask	sAL3_MASK	ISR Mask
devDSB, devCntr	sAL3_DSB	Current Device Status Block (counters)

Module Status Block

The Module Status Block holds Alarm, Status and Statistics information for the Module, as well as dynamic configuration information that can be modified by the USER.

Table 33: Module Status Block: sAL3_MSB

Field Name	Field Type	Field Description
valid	UINT2	Indicates that this structure is valid
moduleOK	UINT2	General health of the Module

Device Status Block

The Device Status Block holds Alarm, Status and Statistics information for the Device, as well as dynamic configuration information that can be modified by the USER.

Table 34: Device Status Block: sAL3_DSB

Field Name				Field Type	Field Description
counter				UINT4	Counter Return Value
a1sp [AL3_MAX _A1SP]	rxOAMCellCnt			UINT4	RX OAM cell count
	rxDroppedOAMCellCnt			UINT4	RX dropped OAM cell count
	txOAMCellCnt			UINT4	TX OAM cell count
	line[[AL3_ LINES_PER_	rxQue[AL3_ QUEUES_PER	seqErrCnt	UINT4	RX sequence error count

Field Name				Field Type	Field Description
	A1SP]	_LINE]	badSNPCnt	UINT4	RX bad SNP count
			cellCnt	UINT4	RX cell count
			stickyBits	UINT4	RX sticky bits
			droppedCellCnt	UINT4	RX dropped cell count
			underrunCnt	UINT4	RX underrun count
			lostCellCnt	UINT4	RX lost cell count
			overrunCnt	UINT4	RX Overrun count
			ptrReFrameCnt	UINT4	RX pointer reframe count
			ptrPerrCnt	UINT4	RX pointer parity error count
			misInsertedCellCnt	UINT4	RX misinserted cell count
		rxQue[AL3_QUEUES_PER_LINE]	condCellCnt	UINT4	TX conditioned cell count
			supCellCnt	UINT4	TX suppressed cell count
			cellCnt	UINT4	TX cell count

4.4 Structures Passed Through RTOS Buffers

Interrupt Service Vector

The Interrupt Service Vector is used in two ways. First, it determines the size of buffer required by the RTOS for use in the driver. Second, it is the template for data captured during ISR processing and thereafter sending it to the Deferred Processing Routine (DPR).

Table 35: Interrupt Service Vector: sAL3_ISV

Field Name	Field Type	Field Description
devId	sDEV_HNDL	Device Handle
master	UINT2	Master Interrupt

Deferred Processing Vector

The Deferred Processing Vector is used in two ways. First, it determines the size of buffer required by the RTOS for use in the driver. Second, it also acts as a template for data assembled by the DPR and sent to the application code.

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

Table 36: Deferred Processing Vector: sAL3_DPV

Field Name	Field Type	Field Description
data	UINT2	Additional information describing the event
index	UINT2	Additional information describing the event – only used for A1SP event.

5 APPLICATION PROGRAMMING INTERFACE

This section provides a detailed description of each function that is a member of the AAL1gator-32 driver Application Programming Interface (API).

5.1 Module Initialization

Opening Modules: al3ModuleOpen

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

Prototype `INT4 al3ModuleOpen(sAL3_MIV *pMIV)`

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

Outputs pointer to MDB passed through the MIV

Returns Success = `AL3_OK`

Failure = `<AL3 ERROR CODES>`

Valid States MOD_START

Side Effects Changes the STATE of the MODULE to MOD_IDLE

Closing Modules: al3ModuleClose

This function performs module level shutdowns of the driver. This involves deleting all devices controlled by the driver (by calling `al3Delete` for each device) and de-allocating the MDB.

Prototype `INT4 al3ModuleClose(void)`

Inputs None

Outputs None

Returns Success = `AL3_OK`

Failure = `<AL3 ERROR CODES>`

Valid States ALL STATES except MOD_START

Side Effects Changes the STATE of the MODULE to MOD_START

5.2 Module Activation

Starting Modules: al3ModuleStart

This function performs module level startup of the driver. This involves allocating RTOS resources such as buffers, semaphores and timers AND installing the ISR handler and DPR task.

Prototype	`INT4 al3ModuleStart(void)`
Inputs	None
Outputs	None
Returns	Success = `AL3_OK`
	Failure = `<AL3 ERROR CODES>`
Valid States	MOD_IDLE
Side Effects	Changes MODULE state to MOD_READY

Stopping Modules: al3ModuleStop

This function performs module level shutdown of the driver. This involves deleting all devices controlled by the driver and de-allocating all RTOS resources.

Prototype	`INT4 al3ModuleStop(void)`
Inputs	None
Outputs	None
Returns	Success = `AL3_OK`
	Failure = `<AL3 ERROR CODES>`
Valid States	READY and ALL DEVICE STATES
Side Effects	Changes MODULE state to MOD_IDLE

5.3 Profile Management

Creating Initialization Profiles: al3AddInitProfile

This function creates an initialization profile stored by the driver. Passing the initialization profile number can initialize devices simply.

Prototype `INT4 al3AddInitProfile(sAL3_DIV *pDIV, UINT2 *pDIVNum)`

Inputs

`pDIV` : pointer to initialization profile to be added

`pDIVNum`: (pointer to) a variable that holds the profile number

Outputs the resulting profile number

Returns Success = `AL3_OK`

Failure = `<AL3 ERROR CODES>`

Valid States ALL MODULE STATES except MOD_START

Side Effects None

Getting Initialization Profiles: al3GetInitProfile

This function Gets the contents of an initialization profile given its profile number.

Prototype `INT4 al3GetInitProfile(UINT2 profNum, sAL3_DIV *pDIV)`

Inputs

`profNum`: profile number

`pDIV`: pointer to profile

Outputs the resulting profile data

Returns Success = `AL3_OK`

Failure = `<AL3 ERROR CODES>`

Valid States ALL MODULE STATES except MOD_START

Side Effects None

Deleting Initialization Profiles: al3DeleteInitProfile

This function deletes an initialization profile given its profile number.

Prototype `INT4 al3DeleteInitProfile(UINT2 profNum)`

Inputs `profNum`: initialization profile number

Outputs None

Returns Success = `AL3_OK`

Failure = <`AL3 ERROR CODES`>

Valid States ALL MODULE STATES except MOD_START

Side Effects None

5.4 Device Initialization

Initializing Devices: al3Init

This function initializes the Device Data Block (DDB) associated to that device during `al3Add`, applies a reset to the device itself, and configures it according to the profile number passed by the Application. This function also calls the `al3Activate` function directly when the `autoActivate` flag is set in the profile. This function can also automatically run some diagnostics on the device before configuring it. This occurs if the `diagOnInit` flag was set in the MIV used in the `al3ModuleOpen` function call.

Prototype `INT4 al3Init(sDEV_HNDL devId, sAL3_DIV *pDIV, UINT2 profileNum)`

Inputs `devId`: device Handle (from `al3Add`)
`pDIV`: (pointer to) the profile for this Device - OR -
`profileNum`: profile number

Outputs None

Returns Success = `AL3_OK`

Failure = <`AL3 ERROR CODES`>

Valid States (MOD_READY) PRESENT

Side Effects Change DEVICE state to INACTIVE

Resetting Devices: al3Reset

This function applies a software reset to the AAL1gator-32 device. The function also resets all the DDB contents (except for the user context). This function is typically called before re-initializing the device.

Prototype	`INT4 al3Reset(sDEV_HNDL devId)`	
Inputs	`devId`:	device Handle (from `al3Add`)
Outputs	None	
Returns	Success = `AL3_OK`	
	Failure = `<AL3 ERROR CODES>`	
Valid States	(MOD_READY) INACTIVE ACTIVE	
Side Effects	Changes DEVICE state to PRESENT	

5.5 Device Addition and Deletion

Adding Devices: al3Add

This function verifies the presence of a new device in the hardware; configures a Device Data block (DDB); stores the contents of the passed Device Initialization Vector (DIV), and passes a pointer to the DDB.

Prototype	`sDEV_HNDL al3Add(void *usrCtxt, UINT2 *baseAddr, INT4 **pperrDevice)`	
Inputs	`usrCtxt`:	pointer to user context
	`baseAddr`:	pointer to base address
	`pperrDevice`: pointer to the location for the pointer of the device error to be stored	
Outputs	Places a pointer to the DDB into the DIV passed by the Application.	
Returns	Success = Device handle	
	Failure = NULL	
Valid States	(MOD_READY) START	
Side Effects	Changes the DEVICE state to PRESENT	

Deleting Devices: al3Delete

This function removes the specified device from the list of devices controlled by the AAL1gator-32 driver. Deleting a device involves clearing the DDB for that device and releasing its associated device handle.

Prototype	`INT4 al3Delete(sDEV_HNDL devId)`	
Inputs	`devId`:	device Handle (from `al3Add`)

Outputs None

Returns Success = AL3_OK

Failure = <AL3 ERROR CODES>

Valid States (MOD_READY) PRESENT INACTIVE ACTIVE

Side Effects Device state changed to START

5.6 Device Activation and De-Activation

Activating Devices: al3Activate

This function restores the state of a device after a de-activate. Interrupts may be re-enabled; queues are not restored.

Prototype `INT4 al3Activate(sDEV_HNDL devId)`

Inputs `devId`: device Handle (from `al3Add`)

Outputs None

Returns Success = AL3_OK

Failure = <AL3 ERROR CODES>

Valid States (MOD_READY) INACTIVE

Side Effects Change the DEVICE state to ACTIVE

Deactivating Devices: al3DeActivate

This function de-activates the device from operation. Interrupts are masked and the device is put into the soft reset state.

Prototype `INT4 al3DeActivate(sDEV_HNDL devId)`

Inputs `devId`: device Handle (from `al3Add`)

Outputs None

Returns Success = AL3_OK

Failure = <AL3 ERROR CODES>

Valid States (MOD_READY) ACTIVE

Side Effects Changes the DEVICE state to INACTIVE

5.7 Device Reading and Writing

Reading from Devices: al3Read

This function reads a register of a specific AAL1gator-32 device by providing the register number. This function derives the actual address location based on the device handle and register number inputs. It then reads the contents of this address location using the system specific macro, `sysAl3ReadReg`.

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

Prototype	`UINT2 al3Read(sDEV_HNDL devId, UINT4 regNum)`	
Inputs	`devId`	: device Handle (from `al3Add`)
	`regNum`	: register number
Outputs	ERROR code written to the DDB	
Returns	Success = the register value	
	Failure = 0x00	
Valid States	(MOD READY) PRESENT, ACTIVE, INACTIVE	
Side Effects	May affect registers that change after a read operation	

Writing to Devices: al3Write

This function writes to a register of a specific AAL1gator-32 device by providing the register number. This function derives the actual address location based on the device handle and register number inputs. It then writes the contents of this address location using the system specific macro, `sysAl3WriteReg`.

Note: A failure to write returns a zero and any error indication writes to the DDB

Prototype	`UINT2 al3Write(sDEV_HNDL devId, UINT4 regNum, UINT2 wdata)`	
Inputs	`devId`	: device Handle (from `al3Add`)
	`regNum`	: register number value: value to be written
	`wdata`	: data to write
Outputs	ERROR code written to the DDB	
Returns	Success = pre-READ register value	
	Failure = 0x00	
Valid States	(MOD_READY) PRESENT, ACTIVE, INACTIVE	
Side Effects	May change the configuration of the Device; some registers require unused bits to be '0'	

Reading from Register Blocks: al3ReadBlock

This function reads from a block of Device Registers. It can be used to read a contiguous register block of a specified Aal1gator 32/8/4 device by providing the starting register number, and the number of registers to read. This function derives the actual start address location based on the device handle and starting register number inputs. It then reads the contents of the associated register data block using multiple calls to the system specific macro, sysAl3ReadReg.

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

Prototype	`UINT2 al3ReadBlock(sDEV_HNDL devId, UINT4 regNum, UINT4 length, UINT2 *pBlock)`	
Inputs	`devId`:	device Handle (from `al3Add`)
	`regNum`:	register number
	`length`:	number of registers to read
	`pBlock`:	(pointer to) block read area
Outputs	ERROR code written to the DDB pBlock is filled with the register data	
Returns	Success = last value read	
	Failure = 0x00	
Valid States	(MOD READY) PRESENT, ACTIVE, INACTIVE	
Side Effects	May affect registers that change after a read operation	

Writing to Register Blocks: al3WriteBlock

This function writes to a block of Device Registers. It can be used to write a contiguous register block of a specified Aal1gator 32/8/4 device by providing the starting register number, and the number of registers to write. This function derives the actual start address location based on the device handle and starting register number inputs. It then writes the contents of the associated register data block using multiple calls to the system specific macro, sysAl3WriteReg.

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

Prototype	`UINT2 al3WriteBlock(sDEV_HNDL devId, UINT4 regNum, UINT4 length, UINT2 *pBlock)`	
Inputs	`devId`:	device Handle (from `al3Add`)
	`regNum`:	start of block register
	`length`:	number of registers in the block
	`pBlock`:	(pointer to) block of write data

Outputs	ERROR code written to the DDB
Returns	Success = last previous value found
	Failure = 0x00
Valid States	(MOD READY) PRESENT, ACTIVE, INACTIVE
Side Effects	May change the configuration of the device

Reading from Indirect Registers: al3ReadInd

This function reads an Indirect Device register. It can be used to Write an Indirect control or mapping register of the SBI block of a specified Aal1gator 32/8/4 device by providing the Page, SPE & Tributary numbers to read. This function derives the actual start address location based on the device handle and input parameters. It then reads the contents of the associated register data block using the system specific macro, sysAl3ReadReg

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

Prototype

```
UINT2 al3ReadInd(sDEV_HNDL devId, AL3_SECTION section,
BOOLEAN map, UINT2 pageNum, UINT2 speNum, UINT2 tribNum)
```

Inputs	`devId`:	device Handle (from `al3Add`)
	`section`:	INSBI or EXSBI
	`map`:	read from control registers or map registers
	`pageNum`:	SBI memory page
	`speNum`:	SBI SPE
	`tribNum`:	SBI tributary
Outputs	ERROR code written to the DDB	
Returns	Success = last value read	
	Failure = 0x00	
Valid States	(MOD READY) PRESENT, ACTIVE, INACTIVE	
Side Effects	May affect registers that change after a read operation	

Writing to Indirect Registers: al3WriteInd

This function writes to an Indirect Device register. It can be used to Write an Indirect control or mapping register of the SBI block of a specified Aal1gator 32/8/4 device by providing the Page, SPE & Tributary numbers to read. This function derives the actual start address location based on the device handle and input parameters. It then reads the contents of the associated register data block using the system specific macro, sysAl3WriteReg.

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

Prototype	`UINT2 al3WriteInd(sDEV_HNDL devId, AL3_SECTION section, BOOLEAN map, UINT2 pageNum, UINT2 speNum, UINT2 tribNum, UINT2 wdata)`	
Inputs	`devId`:	device Handle (from `al3Add`)
	`section`:	INSBI or EXSBI
	`map`:	read from control registers or map registers
	`pageNum`:	SBI memory page
	`speNum`:	SBI SPE
	`tribNum`:	SBI tributary
	`wdata`:	write data
Outputs	ERROR code written to the DDB	
Returns	Success = last previous value found	
	Failure = 0x00	
Valid States	(MOD READY) PRESENT, ACTIVE, INACTIVE	
Side Effects	May change the configuration of the device	

5.8 AAL1 Channel Provisioning

Setting Line Modes: al3SetLineMode

This function sets the line mode for one of the AAL1gator-32 lines.

Prototype	`INT4 al3SetLineMode(sDEV_HNDL devId, UINT2 linkNum, sAL3_DIV_LINE *pParms)`	
Inputs	`devId`:	device Handle (from `al3Add`)
	`linkNum`:	A1SP, Line number
	`pParms`:	points to LINE parameters structure
Outputs	None	
Returns	Success = `AL3_OK`	
	Failure = `<AL3 ERROR CODES>`	
Valid States	(MOD_READY) ACTIVE	
Side Effects	None	

Configuring Underrun Data: al3SetUnderrun

This function configures Underrun Data and Signaling on a timeslot basis. Only use this function, if you want to specify separate underrun parameters for each timeslot in a given queue.

Prototype	`INT4 al3SetUnderrun(sDEV_HNDL devId, UINT2 linkNum, UINT2 timeSlot, UINT2 rxData, UINT2 rxSig)`	
Inputs	`devId`:	device Handle (from `al3Add`)
	`linkNum`:	specifies the line number to configure
	`timeSlot`:	specifies the timeslot to configure
	`rxData`:	new default Rx Conditioned Data
	`rxSig`:	new default Rx Conditioned Signaling Data
Outputs	None	
Returns	Success = `AL3_OK`	
	Failure = `<AL3 ERROR CODES>`	
Valid States	(MOD_READY) ACTIVE	
Side Effects	None	

Setting Global Clock Configuration: al3GlobalClkConfig

This function sets the Global clock configuration for the AAL1gator-32 device.

Prototype	`INT4 al3GlobalClkConfig(sDEV_HNDL devId, sAL3_DIV_CLK *pParms)`	
Inputs	`devId`:	device Handle (from `al3Add`)
	`pParms`:	points to Config params
Outputs	None	
Returns	Success = `AL3_OK`	
	Failure = `<AL3 ERROR CODES>`	
Valid States	(MOD_READY) ACTIVE	
Side Effects	None	

Activating Channels: al3ActivateChannel

This function maps the channels of a T1 or an E1 line in Structured Data Format (SDF) or the entire line in Unstructured Data Format (UDF) to a VP/VC. `al3ActivateChannel` returns a queue handle for future mapping operations.

Prototype	`INT4 al3ActivateChannel(sDEV_HNDL devId, sAL3_QID *queId, UINT2 txLink, UINT4 channels, sAL3_CFG_CHAN *pParms)`	
Inputs	`devId`:	device Handle (from `al3Add`)
	`queId`:	pointer to queue handle
	`txLink`:	A1SP, Line & Queue Number
	`channels`:	bitmap of channels to activate
	`pParms`:	(pointer to) configuration structure
Outputs	Queue Id via the parameter '*queId'	
Returns	Success = `AL3_OK`	
	Failure = <`AL3 ERROR CODES`>	
Valid States	(MOD READY) ACTIVE	
Side Effects	None	

Deactivating Channels: al3DeActivateChannel

This function deactivates the line that is in use, and frees the queue handle.

Prototype	`INT4 al3DeActivateChannel (sDEV_HNDL devId, sAL3_QID queId)`	
Inputs	`devId`:	device Handle (from `al3Add`)
	`queId`:	specifies the queue handle for the line
Outputs	None	
Returns	Success = `AL3_OK`	
	Failure = <`AL3 ERROR CODES`>	
Valid States	(MOD READY) ACTIVE	
Side Effects	None	

Activating Channels with Enhanced Parameters: al3EnhancedActivateChannel

This function maps the channels of a T1 or an E1 line in Structured Data Format (SDF) or the entire line in Unstructured Data Format (UDF) to a VP/VC. It also returns a queue handle used for future operations on the mapping. In addition to the abilities of the `aal1ActivateChannel` function, this function also enables the extend parameters used in configuring the mapping, as well as parameters for configuring Sequence Number Processing, Conditioning, and Idle Channel Detection. Passing a NULL Pointer in place of a pointer to any of the configuration parameter data structures results in the function using the default parameters for that data structure.

Note: Passing a NULL Pointer in place of a pointer to any of the configuration parameter data structures results in the function using the default parameters for that data structure.

Prototype

```
INT4 al3EnhancedActivateChannel(sDEV_HNDL devId, sAL3_QID
*queId, UINT2  txLink, UINT4  channels, sAL3_CFG_CHAN
*pParms, sAL3_CFG_CHAN_ENH *pEnhParms, sAL3_CFG_CHAN_SNP
*pSNPParms, sAL3_CFG_CHAN_COND *pCondParms,
sAL3_CFG_CHAN_IDET *pIDetParms)
```

Inputs

devId:	device Handle (from `al3Add`)
queId:	pointer to queue handle
`txLink`:	A1SP, Line & Queue Number
`channels`:	bitmap of channels to activate
`pParms`:	(pointer to) configuration structure
`pEnhParms`:	(pointer to) Enhanced parameters
`pSNPParms`:	(pointer to) Sequence Number Processing parameters
`pCondParms`:	(pointer to) Conditioning parameters
`pIDetParms`:	(pointer to) Idle Detection parameters

Outputs Queue Id via the parameter '*queId'

Returns Success = `AL3_OK`

Failure = <`AL3 ERROR CODES`>

Valid States (MOD READY) ACTIVE

Side Effects None

Activating Unstructured Channels: al3ActivateChannelUnstr

This function activates a line of the device in Unstructured Data Format (UDF) mode. Returns a queue handle for future operations on the queue.

Prototype

```
INT4 al3ActivateChannelUnstr(sDEV_HNDL devId, sAL3_QID
*queId, UINT2 txLink, sAL3_CFG_CHAN *pParms)
```

Inputs

`devId`:	device Handle (from `al3Add`)
`queId`:	pointer to queue handle
`txLink`:	A1SP, Line & Queue Number
`pParms`:	(pointer to) configuration structure

Outputs Queue Id via the parameter '*queId'

Returns Success = `AL3_OK`

Failure = <`AL3 ERROR CODES`>

Valid States (MOD READY) ACTIVE

Side Effects None

Activating Unstructured Channels with Enhanced Parameters: al3EnhancedActivateChannelUnstr

This function activates a line of the device in Unstructured Data Format (UDF) mode. `al3EnhancedActivateChannelUnstr` returns a queue handle enabling future operations on the line. In addition to the abilities of the `aal1ActivateLine` function, this function also provides the user the ability to provide extended parameters used in configuring the line, as well as parameters for configuring Sequence Number Processing, Conditioning, and Idle Channel Detection. Passing a NULL Pointer in place of a pointer to any of the configuration parameter data structures results in the function using the default parameters for that data structure.

Note: Passing a NULL Pointer in place of a pointer to any of the configuration parameter data structures results in the function using the default parameters for that data structure.

Prototype

```
INT4 al3EnhancedActivateChannelUnstr(sDEV_HNDL devId,
sAL3_QID *queId, UINT2 txLink, sAL3_CFG_CHAN *pParms,
sAL3_CFG_CHAN_ENH *pEnhParms, sAL3_CFG_CHAN_SNP pSNPParms,
sAL3_CFG_CHAN_COND *pCondParms, sAL3_CFG_CHAN_IDET
*pIDetParms)
```

Inputs

`devId`:	device Handle (from `al3Add`)
`queId`:	pointer to queue handle
`txLink` :	A1SP, Line & Queue Number
`pParms` :	(pointer to) configuration structure
`pEnhParms` :	(pointer to) Enhanced parameters
`pSNPParms` :	(pointer to) Sequence Number Processing parameters
`pCondParms`:	(pointer to) Conditioning parameters
`pIDetParms`:	(pointer to) Idle Detection parameters

Outputs Queue Id via the parameter '*queId'

Returns Success = `AL3_OK`

Failure = <`AL3 ERROR CODES`>

Valid States (MOD READY) ACTIVE

Side Effects None

Deactivating Unstructured Channels: al3DeActivateChannelUnstr

This function deactivates the line that is in use, and frees the queue handle.

Prototype

```
INT4 al3DeActivateChannelUnstr(sDEV_HNDL devId,
sAL3_QID queId)
```

Inputs

`devId`:	device Handle (from `al3Add`)
`queId`:	specifies the queue handle for the line

Outputs None

Returns Success = AL3_OK

Failure = <AL3 ERROR CODES>

Valid States (MOD READY) ACTIVE

Side Effects None

Activating Structured Channels : al3ActivateChannelStr

This function maps the channels of a T1 or an E1 line in Structured Data Format (SDF) to a VP/VC. Returns a queue handle that will be used for future operations on the mapping.

Prototype

```
INT4 al3ActivateChannelStr(sDEV_HNDL devId,
sAL3_QID *queId, UINT2 txLink, UINT4 channels,
sAL3_CFG_CHAN *pParms)
```

Inputs

devId:	device Handle (from al3Add)
queId:	pointer to queue handle
txLink:	A1SP, Line & Queue Number
channels:	bitmap of channels to activate
pParms:	(pointer to) configuration structure

Outputs Queue Id via the parameter '*queId'

Returns Success = AL3_OK

Failure = <AL3 ERROR CODES>

Valid States (MOD READY) ACTIVE

Side Effects None

Activating Structured Channels With Enhanced Parameters: al3EnhancedActivateChannelStr

This function maps the channels of a T1 or an E1 line in Structured Data Format (SDF) to a VP/VC. Returns a queue handle used for future operations on the mapping. In addition to the abilities of the aal1ActivateChannel function, this function provides the user the ability to provide extended parameters used in configuring the mapping, as well as parameters for configuring Sequence Number Processing, Conditioning, and Idle Channel Detection. Passing a NULL Pointer in place of a pointer to any of the configuration parameter data structures results in the function using the default parameters for that data structure.

Note: Passing a NULL Pointer in place of a pointer to any of the configuration parameter data structures results in the function using the default parameters for that data structure.

Prototype

```
INT4 al3EnhancedActivateChannelStr( sDEV_HNDL
devId, sAL3_QID *queId, UINT2  txLink, UINT4
channels, sAL3_CFG_CHAN *pParms,
sAL3_CFG_CHAN_ENH *pEnhParms, sAL3_CFG_CHAN_SNP
```

`*pSNPParms, sAL3_CFG_CHAN_COND *pCondParms, sAL3_CFG_CHAN_IDET *pIDetParms)`

Inputs

`devId`:	device Handle (from `al3Add`)
`queId`:	pointer to queue handle
`txLink`:	A1SP, Line & Queue Number
`channels`:	bitmap of channels to activate
`pParms`:	(pointer to) configuration structure
`pEnhParms`:	(pointer to) Enhanced parameters
`pSNPParms`:	(pointer to) Sequence Number Processing parameters
`pCondParms`:	(pointer to) Conditioning parameters
`pIDetParms`:	(pointer to) Idle Detection parameters

Outputs Queue Id via the parameter '*queId'

Returns Success = `AL3_OK`

Failure = <AL3 ERROR CODES>

Valid States (MOD READY) ACTIVE

Side Effects None

Deactivating Structured Channels: al3DeActivateChannelStr

This function deactivates the channels on a line that is (are) in use, and frees the queue handle.

Prototype `INT4 al3DeActivateChannelStr(sDEV_HNDL devId, sAL3_QID queId)`

Inputs

`devId`:	device Handle (from `al3Add`)
`queId`:	specifies the queue handle for the channels

Outputs None

Returns Success = `AL3_OK`

Failure = <AL3 ERROR CODES>

Valid States (MOD READY) ACTIVE

Side Effects None

Associating Channels With An Existing Mapping: al3AssociateChannel

This function associates more T1/E1 timeslots to an existing mapping. After configuring the mapping, it enables it.

Prototype `INT4 al3AssociateChannel(sDEV_HNDL devId, sAL3_QID queId, UINT4 chanMap)` `sAL3_QID queId, UINT4 chanMap)`

Inputs	`devId`:	device Handle (from `al3Add`)
	`queId` :	specifies the queue handle for the channels
	`chanMap`:	bitmap of the channels to add
Outputs	None	
Returns	Success = `AL3_OK`	
	Failure = `<AL3 ERROR CODES>`	
Valid States	(MOD READY) ACTIVE	
Side Effects	None	

Disassociating Channels With An Existing Mapping: al3DisAssociateChannel

This function disassociates already mapped T1/E1 timeslots from an existing mapping. After reconfiguring the mapping, the function enables it.

Prototype

```
INT4 al3DisAssociateChannel (sDEV_HNDL devId,
sAL3_QID queId, UINT4 chanMap)
```

Inputs	`devId`:	device Handle (from `al3Add`)
	`queId` :	specifies the queue handle for the channels
	`chanMap`:	bitmap of the channels to remove
Outputs	None	
Returns	Success = `AL3_OK`	
	Failure = `<AL3 ERROR CODES>`	
Valid States	(MOD READY) ACTIVE	
Side Effects	None	

5.9 Channel Conditioning

Enabling Transmit Conditioning: al3EnableTxCond

This function enables transmit conditioning for an existing channel(s) to VP/VC mapping.

Prototype

```
INT4 al3EnableTxCond(sDEV_HNDL devId, sAL3_QID
queId)
```

Inputs	`devId`:	device Handle (from `al3Add`)

	`queId`:	specifies the queue handle for the channels
Outputs	None	
Returns	Success = `AL3_OK`	
	Failure = <`AL3 ERROR CODES`>	
Valid States	(MOD_READY) ACTIVE	
Side Effects	None	

Disabling Transmit Conditioning: al3DisableTxCond

This function disables transmit conditioning for an existing channel(s) to VP/VC mapping.

Prototype	`INT4 al3DisableTxCond(sDEV_HNDL devId, sAL3_QID queId)`	
Inputs	`devId`:	device Handle (from `al3Add`)
	`queId`:	specifies the queue handle for the channels
Outputs	None	
Returns	Success = `AL3_OK`	
	Failure = <`AL3 ERROR CODES`>	
Valid States	(MOD_READY) ACTIVE	
Side Effects	None	

Enabling Receive Conditioning: al3EnableRxCond

This function enables receive conditioning for an existing channel(s) to VP/VC mapping.

Prototype	`INT4 al3EnableRxCond(sDEV_HNDL devId, sAL3_QID queId)`	
Inputs	`devId`:	device Handle (from `al3Add`)
	`queId`:	specifies the queue handle for the channels
Outputs	None	
Returns	Success = `AL3_OK`	
	Failure = <`AL3 ERROR CODES`>	
Valid States	(MOD_READY) ACTIVE	
Side Effects	None	

Disabling Receive Conditioning: al3DisableRxCond

This function disables receive conditioning for an existing channel(s) to VP/VC mapping.

Prototype `INT4 al3DisableRxCond(sDEV_HNDL devId, sAL3_QID queId)`

Inputs `devId`: device Handle (from `al3Add`)

`queId`: specifies the queue handle for the channels

Outputs None

Returns Success = `AL3_OK`

Failure = <`AL3 ERROR CODES`>

Valid States (MOD_READY) ACTIVE

Side Effects None

5.10 SRTS Functions

Enabling SRTS: al3EnableSRTS

This function enables SRTS for the given T1 or E1 line. SRTS can only be enabled if the line is in UDF mode.

Prototype `INT4 al3EnableSRTS(sDEV_HNDL devId, UINT2 linkNum)`

Inputs `devId`: device Handle (from `al3Add`)
`linkNum`: A1SP, Line numbers

Outputs None

Returns Success = AL3_OK

Failure = <AL3 ERROR CODES>

Valid States linkNum - A1SP, Line numbers

Side Effects None

Disabling SRTS: al3DisableSRTS

This function disables SRTS for the given T1 or E1 line.

Prototype `INT4 al3DisableSRTS(sDEV_HNDL devId, UINT2 linkNum)`

Inputs `devId`: device Handle (from `al3Add`)
`linkNum`: A1SP, Line numbers

Outputs None

Returns Success = AL3_OK

	Failure = <AL3 ERROR CODES>
Valid States	(MOD_READY) ACTIVE
Side Effects	None

5.11 Loopback Functions

Enabling Loopbacks: al3EnableLpbk

This function enables loopback for the specified AAL1 channel Q. The loopback is performed before the AAL1 cells that are coming from the Line Interface reach the UTOPIA interface.

Prototype	`INT4 al3EnableLpbk(sDEV_HNDL devId, sAL3_QID queId)`	
Inputs	`devId`:	device Handle (from `al3Add`)
	`queId`:	AAL1 channel queue Handle (from `al3ActivateChannel`)
Outputs	None	
Returns	Success = `AL3_OK`	
	Failure = <`AL3 ERROR CODES`>	
Valid States	(MOD_READY) ACTIVE INACTIVE	
Side Effects	None	

Disabling Loopbacks: al3DisableLpbk

This function disables loopback for the specified AAL1 channel Q. The loopback is performed before the AAL1 cells that are coming from the Line Interface reach the UTOPIA interface.

Prototype	`INT4 al3DisableLpbk(sDEV_HNDL devId, sAL3_QID queId)`	
Inputs	`devId`:	device Handle (from `al3Add`)
	`queId`:	AAL1 channel queue Handle (from `al3ActivateChannel`)
Outputs	None	
Returns	Success = `AL3_OK`	
	Failure = <`AL3 ERROR CODES`>	
Valid States	(MOD_READY) ACTIVE INACTIVE	
Side Effects	None	

Enabling Utopia Loopbacks: al3UtopiaLpbkEnable

This function enables a loopback at the Utopia interface.

Prototype `INT4 al3UtopiaLpbkEnable (sDEV_HNDL devId, BOOLEAN vciMode, UINT2 lpbkVci)`

Inputs

`devId`:	device Handle (from `al3Add`)
`vciMode`:	flag that enables VCI checking
`lpbkVci`:	vci of the looped data

Outputs None

Returns Success = `AL3_OK`

Failure = <`AL3 ERROR CODES`>

Valid States (MOD_READY) ACTIVE INACTIVE

Side Effects May Set / Clear any register in the Device

Disabling Utopia Loopbacks: al3UtopiaLpbkDisable

This function disables a loopback at the Utopia interface.

Prototype `INT4 al3UtopiaLpbkDisable (sDEV_HNDL devId)`

Inputs `devId`: device Handle (from `al3Add`)

Outputs None

Returns Success = `AL3_OK`

Failure = <`AL3 ERROR CODES`>

Valid States (MOD_READY) ACTIVE INACTIVE

Side Effects May Set / Clear any register in the Device

5.12 Idle Detection Functions

Setting Activate Timeslots: al3SetTimeslotActive

This function uses with processor-based Idle Channel Detection. This function sets a timeslot as active.

Prototype `INT4 al3SetTimeslotActive(sDEV_HNDL devId, UINT2 linkNum, UINT2 timeSlot)`

Inputs

`devId`:	device Handle (from `al3Add`)
`linkNum`:	specifies the line number to set

	`timeSlot`:	specifies the timeslot to set Active

Outputs None

Returns Success = `AL3_OK`

Failure = <`AL3 ERROR CODES`>

Valid States (MOD_READY) ACTIVE

Side Effects None

Setting Idle Timeslots: al3SetTimeslotIdle

This function uses processor-based Idle Channel Detection. This function sets a timeslot as idle.

Prototype `INT4 al3SetTimeslotIdle(sDEV_HNDL devId, UINT2 linkNum, UINT2 timeSlot)`

Inputs

`devId`:	device Handle (from `al3Add`)
`linkNum`:	specifies the line number to set
`timeSlot`:	specifies the timeslot to set Idle

Outputs None

Returns Success = `AL3_OK`

Failure = <`AL3 ERROR CODES`>

Valid States (MOD_READY) ACTIVE

Side Effects None

5.13 OAM Functions

Transmitting OAM Cells: al3TxOAMcell

This function transmits an OAM cell.

Prototype `INT4 al3TxOAMcell(sDEV_HNDL devId, void *pOAMCell, BOOLEAN crcON)`

Inputs

`devId`:	device Handle (from `al3Add`)
`pOAMCell`:	(pointer to) the OAM Cell to send
`crcOn`:	flag to indicate if CRC Check should be run

Outputs None

Returns Success = `AL3_OK`

Failure = <`AL3 ERROR CODES`>

Valid States (MOD_READY) ACTIVE INACTIVE

Side Effects None

Receiving OAM Cells: al3RxOAMcell

This function receives an OAM cell by placing it in a buffer. Typically called by the ISR or the DPR.

Prototype `INT4 al3RxOAMcell(sDEV_HNDL devId, void *pOAMCell, BOOLEAN *pCRCPass)`

Inputs
`devId`: device Handle (from `al3Add`)
`pOAMCell`: (pointer to) space to hold the OAM Cell
`pCRCPass`: (pointer to) the variable indicating CRC Passed

Outputs the Cell contents via pOAMCell
the state of the CRC check via pCRCPass

Returns Success = `AL3_OK`

Failure = `<AL3 ERROR CODES>`

Valid States (MOD_READY) ACTIVE INACTIVE

Side Effects None

5.14 Alarms and Statistics

Enabling DS3 AIS Cells: al3EnableDS3AISCells

This function enables DS3 AIS cells to be sent on a particular high-speed line.

Prototype `INT4 al3EnableDS3AISCells(sDEV_HNDL deviceHandle, UINT2 lineNo)`

Inputs
`devId`: device Handle (from `al3Add`)
`lineNum`: LINE number (0, 16) (Line 16 only for AAL1GATOR-32)

Outputs None

Returns Success = `AL3_OK`

Failure = `<AL3 ERROR CODES>`

Valid States (MOD_READY) ACTIVE INACTIVE

Side Effects None

Disabling DS3 AIS Cells: al3DisableDS3AISCells

This function disables DS3 AIS cells being sent on a particular high-speed line.

Prototype `INT4 al3DisableDS3AISCells(sDEV_HNDL devId, UINT2 lineNum)`

Inputs `devId`: device Handle (from `al3Add`)
`lineNum`: LINE number (0, 16) (Line 16 only for AAL1GATOR-32)

Outputs None

Returns Success = `AL3_OK`

Failure = `<AL3 ERROR CODES>`

Valid States (MOD_READY) ACTIVE INACTIVE

Side Effects None

Enabling SBI Alarms: al3EnableSBIAlarm

This function enables alarm generation in a tributary on the SBI bus.

Note: This function is not supported by the AAL1gator-4 or AAL1gator-8.

Prototype `INT4 al3EnableSBIAlarm(sDEV_HNDL devId, UINT2 lineNum)`

Inputs `devId`: device Handle (from `al3Add`)
`lineNum`: LINE number (0-31)

Outputs None

Returns Success = AL3_OK

Failure = <AL3 ERROR CODES>

Valid States (MOD_READY) ACTIVE INACTIVE

Side Effects None

Disabling SBI Alarms: al3DisableSBIAlarm

This function disables alarm generation in a tributary on the SBI bus.

Note: This function is not supported by the AAL1gator-8 or AAL1gator-4.

Prototype `INT4 al3DisableSBIAlarm(sDEV_HNDL devId, UINT2 lineNum)`

Inputs `devId`: device Handle (from `al3Add`)
`lineNum`: LINE number (0-31)

Outputs None

Returns Success = AL3_OK

Failure = <AL3 ERROR CODES>

Valid States	(MOD_READY) ACTIVE INACTIVE
Side Effects	None

Returning Conditional Cell Count: al3GetTCondCellCount

This function returns the Tx Conditioned Cell count for the specified device and queue.

Prototype	`UINT4 al3GetTCondCellCount (sDEV_HNDL devId, sAL3_QID queId)`
Inputs	`devId`: device Handle (from `al3Add`) `queId`: QUEUE Handle
Outputs	None
Returns	The current counter value extended to 32 bits
Valid States	(MOD_READY) ACTIVE INACTIVE
Side Effects	None

Returning Suppressed Cell Count: al3GetTSupprCellCount

This function returns the Tx Suppressed Cell count for the specified device and queue.

Prototype	`UINT4 al3GetTSupprCellCount (sDEV_HNDL devId, sAL3_QID queId)`
Inputs	`devId`: device Handle (from `al3Add`) `queId`: QUEUE Handle
Outputs	None
Returns	The current counter value extended to 32 bits
Valid States	(MOD_READY) ACTIVE INACTIVE
Side Effects	None

Returning Tx Cell Count: al3GetTCellCount

This function returns the Tx Cell count for the specified device and queue.

Prototype	`UINT4 al3GetTCellCount(sDEV_HNDL devId, sAL3_QID queId)`
Inputs	`devId`: device Handle (from `al3Add`) `queId`: QUEUE Handle
Outputs	None
Returns	The current counter value extended to 32 bits
Valid States	(MOD_READY) ACTIVE INACTIVE

Side Effects None

Returning Rx OAM Cell Count: al3GetROAMCellCount

This function returns the Rx OAM Cell count for the specified device.

Prototype `UINT4 al3GetROAMCellCount (sDEV_HNDL devId , UINT2 lineNum)`

Inputs
`devId`: device Handle (from `al3Add`)
`lineNum`: LINE number (0,8,16,24 for AAL1GATOR-32) (0 for AAL1GATOR-8/4)

Outputs None

Returns The current counter value extended to 32 bits

Valid States (MOD_READY) ACTIVE INACTIVE

Side Effects None

Returning Tx OAM Cell Count: al3GetTOAMCellCount

This function retrieves the Tx OAM Cell count for the specified device.

Prototype `UINT4 al3GetTOAMCellCount (sDEV_HNDL devId, UINT2 lineNum)`

Inputs
`devId`: device Handle (from `al3Add`)
`lineNum`: LINE number (0,8,16,24 for AAL1GATOR-32) (0 for AAL1GATOR-8/4)

Outputs None

Returns The current counter value extended to 32 bits

Valid States (MOD_READY) ACTIVE INACTIVE

Side Effects None

Returning Dropped Rx OAM Cell Count: al3GetRDroppedOAMCellCount

This function returns the Dropped Rx OAM Cell count for the specified device.

Prototype `UINT4 al3GetRDroppedOAMCellCount (sDEV_HNDL devId, UINT2 lineNum)`

Inputs
`devId`: device Handle (from `al3Add`)
`lineNum`: LINE number (0,8,16,24 for AAL1GATOR-32) (0 for AAL1GATOR-8/4)

Outputs None

Returns The current counter value extended to 32 bits

Valid States (MOD_READY) ACTIVE INACTIVE

Side Effects None

Returning SN Error Count: al3GetRIncorrectSn

This function returns the Rx Cells with SN errors for the specified device and queue.

Prototype `UINT4 al3GetRIncorrectSn(sDEV_HNDL devId, sAL3_QID queId)`

Inputs
`devId`: device Handle (from `al3Add`)
`queId`: QUEUE Handle

Outputs None

Returns The current counter value extended to 32 bits

Valid States (MOD_READY) ACTIVE INACTIVE

Side Effects None

Returning Rx Cell Count With Incorrect SNP: al3GetRIncorrectSnp

This function returns the Rx Cell Count with the incorrect SNP, for the specified device and queue.

Prototype `UINT4 al3GetRIncorrectSnp(sDEV_HNDL devId, sAL3_QID queId)`

Inputs
`devId`: device Handle (from `al3Add`)
`queId`: QUEUE Handle

Outputs None

Returns The current counter value extended to 32 bits

Valid States (MOD_READY) ACTIVE INACTIVE

Side Effects None

Returning Cell Count: al3GetRCellCount

This function returns the Rx Cell count for the specified device and queue.

Prototype `UINT4 al3GetRCellCount(sDEV_HNDL devId, sAL3_QID queId)`

Inputs
`devId`: device Handle (from `al3Add`)
`queId`: QUEUE Handle

Outputs None

Returns The current counter value extended to 32 bits

Valid States (MOD_READY) ACTIVE INACTIVE

Side Effects None

Returning Dropped Rx Cell Count: al3GetRDroppedCellCount

This function returns the Dropped Rx Cells count for the specified device and queue.

Prototype	`UINT4 al3GetRDroppedCellCount(sDEV_HNDL devId, sAL3_QID queId)`	
Inputs	`devId`:	device Handle (from `al3Add`)
	`queId`:	QUEUE Handle
Outputs	None	
Returns	The current counter value extended to 32 bits	
Valid States	(MOD_READY) ACTIVE INACTIVE	
Side Effects	None	

Returning Rx Underrun Count: al3GetRecvUnderrun

This function returns the Receiver Underrun count for the specified device and queue.

Prototype	`UINT4 al3GetRecvUnderrun(sDEV_HNDL devId, sAL3_QID queId)`	
Inputs	`devId`:	device Handle (from `al3Add`)
	`queId`:	QUEUE Handle
Outputs	None	
Returns	The current counter value extended to 32 bits	
Valid States	(MOD_READY) ACTIVE INACTIVE	
Side Effects	None	

Returning Rx Overrun Count: al3GetRecvOverrun

This function returns the Receiver Overrun count for the specified device and queue.

Prototype	`UINT4 al3GetRecvOverrun (sDEV_HNDL devId, sAL3_QID queId)`	
Inputs	`devId`:	device Handle (from `al3Add`)
	`queId`:	QUEUE Handle
Outputs	None	
Returns	The current counter value extended to 32 bits	
Valid States	(MOD_READY) ACTIVE INACTIVE	
Side Effects	None	

Returning Rx Pointer Reframe Count: al3GetRPtrReframeCount

This function returns the Rx Pointer Reframe count for the specified device and queue.

Prototype	`UINT4 al3GetRPtrReframeCount(sDEV_HNDL devId, sAL3_QID queId)`

Inputs	`devId`:	device Handle (from `al3Add`)
	`queId`:	QUEUE Handle
Outputs	None	
Returns	The current counter value extended to 32 bits	
Valid States	(MOD_READY) ACTIVE INACTIVE	
Side Effects	None	

Returning Rx Pointer Parity Error Count: al3GetRPtrParErrorCount

This function returns the Rx Pointer Parity Error count for the specified device and queue.

Prototype `UINT4 al3GetRPtrParErrorCount(sDEV_HNDL devId, sAL3_QID queId)`

Inputs	`devId`:	device Handle (from `al3Add`)
	`queId`:	QUEUE Handle
Outputs	None	
Returns	The current counter value extended to 32 bits	
Valid States	(MOD_READY) ACTIVE INACTIVE	
Side Effects	None	

Returning Lost Cell Count: al3GetRLostCellCount

This function returns the Lost Cell count for the specified device and queue.

Prototype `UINT4 al3GetRLostCellCount(sDEV_HNDL devId, sAL3_QID queId)`

Inputs	`devId`:	device Handle (from `al3Add`)
	`queId`:	QUEUE Handle
Outputs	None	
Returns	The current counter value extended to 32 bits	
Valid States	(MOD_READY) ACTIVE INACTIVE	
Side Effects	None	

Returning Misinserted Cell Count: al3GetRMisInsertedCellCount

This function returns the Misinserted Cell count for the specified device and queue.

Prototype `UINT4 al3GetRMisInsertedCellCount (sDEV_HNDL devId, sAL3_QID queId)`

Inputs	`devId`:	device Handle (from `al3Add`)
	`queId`:	QUEUE Handle

Outputs None

Returns The current counter value extended to 32 bits

Valid States (MOD_READY) ACTIVE INACTIVE

Side Effects None

Returning Sticky Bits: al3GetStickyBits

This function returns the Sticky Bit Word for the specified device and queue.

Note: Sticky Bits automatically clear after they have been read.

Prototype `UINT4 al3GetStickyBits (sDEV_HNDL devId, sAL3_QID queId, sAL3_STICKY *pSticky)`

Inputs

`devId`:	device Handle (from `al3Add`)
`queId`:	QUEUE Handle
`pSticky`:	(pointer to) space to return the Sticky Bits

Outputs None

Returns The current data value extended to 32 bits

Valid States (MOD_READY) ACTIVE INACTIVE

Side Effects None

5.15 UTOPIA Bus Configuration Functions

Configuring Utopia Bus: al3UtopiaConfig

This function configures the device's UTOPIA/Any-PHY bus.

Prototype `INT4 al3UtopiaConfig(sDEV_HNDL devId, sAL3_DIV_UTOPIA *pParms)`

Inputs

`devId`:	device Handle (from `al3Add`)
`pParms`:	(pointer to) utopia parameters structure

Outputs None

Returns Success = AL3_OK

Failure = <AL3 ERROR CODES>

Valid States (MOD_READY) ACTIVE

Side Effects None

5.16 RAM Interface Configuration Functions

Configuring RAM Interface: al3RamConfig

This function configures the device's two SRAM interfaces.

Prototype `INT4 al3RamConfig(sDEV_HNDL devId, sAL3_DIV_RAM *pParms)`

Inputs `devId`: device Handle (from `al3Add`)
`pParms`: points to RAM config params

Outputs None

Returns Success = AL3_OK
Failure = <AL3 ERROR CODES>

Valid States (MOD_READY) ACTIVE

Side Effects None

5.17 SBI Bus Configuration Functions

Configuring SBI Bus: al3SBIConfig

This function configures the device's SBI bus.

Note: This function is not supported by the AAL1gator-8 or AAL1gator-4.

Prototype `INT4 al3SBIConfig(sDEV_HNDL devId, sAL3_DIV_SBI *pParms)`

Inputs
`devId`: device Handle (from `al3Add`)
`pParms`: points to SBI bus config params

Outputs None

Returns Success = AL3_OK

Failure = <AL3 ERROR CODES>

Valid States (MOD_READY) ACTIVE

Side Effects None

Configuring SBI Bus Tributarys: al3SBITribConfig

This function configures a tributary on the SBI bus.

Note: The AAL1gator-4 or AAL1gator-8 does not support this function.

Prototype `INT4 al3SBITribConfig(sDEV_HNDL devId, UINT2 speNum, UINT2 tribNum, sAL3_DIV_TRIB *pParms)`

Inputs
`devId`: device Handle (from `al3Add`)
`speNum`: SPE number (1-3)
`tribNum`: Tributary number (1-28)
`pParms`: (pointer to) TRIB parameters structure

Outputs None

Returns Success = AL3_OK

Failure = <AL3 ERROR CODES>

Valid States (MOD_READY) ACTIVE

Side Effects None

5.18 Direct Line Configuration Functions

Configuring Direct Lines: al3DirectConfig

This function configures the device's direct low speed (T1/E1) line interface.

Prototype `INT4 al3DirectConfig(sDEV_HNDL devId, UINT2 linkNum, sAL3_DIV_DIRECT *pParms)`

Inputs

`devId`: device Handle (from `al3Add`)
`linkNum`: Link Number (0-15 for AAL1GATOR-32)
(0-7 for AAL1GATOR-8)
(0-3 for AAL1GATOR-4)
`pParms`: (pointer to) direct parameters structure

Outputs None

Returns Success = AL3_OK

Failure = <AL3 ERROR CODES>

Valid States (MOD_READY) ACTIVE

Side Effects None

5.19 Interrupt Service Functions

Getting ISR Mask Registers: al3GetMask

This function returns the contents of the interrupt mask registers of the AAL1gator-32 device.

Prototype `INT4 al3GetMask(sDEV_HNDL devId, sAL3_MASK *pMASK)`

Inputs

`devId`: device Handle (from `al3Add`)
`pMASK`: (pointer to) mask structure

Outputs None

Returns Success = `AL3_OK`

Failure = <`AL3 ERROR CODES`>

Valid States INACTIVE, ACTIVE

Side Effects None

Setting ISR Mask Registers: al3SetMask

This function sets the contents of the interrupt mask registers of the AAL1gator-32 device.

Prototype `INT4 al3SetMask(sDEV_HNDL devId, sAL3_MASK *pMASK)`

Inputs

`devId`: device Handle (from `al3Add`)

`pMASK`: (pointer to) mask structure

Outputs None

Returns Success = `AL3_OK`

Failure = <`AL3 ERROR CODES`>

Valid States (MOD_READY) ACTIVE INACTIVE

Side Effects None

Clearing ISR Mask Registers: al3ClearMask

This function clears individual interrupt bits and registers in the AAL1gator-32 device. Any bits that are set in the passed structure clear in the associated AAL1gator-32 registers.

Prototype `INT4 al3ClearMask(sDEV_HNDL devId, sAL3_MASK *pMASK)`

Inputs

`devId`: device Handle (from `al3Add`)
`pMASK`: (pointer to) mask structure

Outputs None

Returns Success = `AL3_OK`

Failure = <`AL3 ERROR CODES`>

Valid States (MOD_READY) ACTIVE INACTIVE

Side Effects None

Polling ISR Registers: al3Poll

This function commands the Driver to poll the interrupt registers in the Device. The call will fail unless the device is initialized into polling mode. The output of the poll is the same as when interrupts are enabled: the data gathered passes to the DPR for disposition.

Prototype `INT4 al3Poll (sDEV_HNDL devId, void *pBuf)`

Inputs

`devId`: device Handle (from `al3Add`)

pBuf: (pointer to) a preallocated ISV

Outputs None

Returns SUCCESS -> AL3_OK

FAILURE -> <AAL1GATOR-3 ERROR CODES>

Valid States (MOD_READY) ACTIVE INACTIVE

Side Effects None

ISR Config: al3ISRConfig

This function configures the driver to be in either polled or interrupt mode.

Prototype `INT4 al3ISRConfig(sDEV_HNDL devId, AL3_ISR_MODE mode)`

Inputs

`devId`: device Handle (from `al3Add`)
`mode`: polled or interrupt mode (AL3_ISR_MANUAL, AL3_ISR_HDWR)

Outputs None

Returns Success = AL3_OK

Failure = <AL3 ERROR CODES>

Valid States (MOD_READY) PRESENT ACTIVE INACTIVE

Side Effects None

Reading Interrupt Status Registers: al3ISR

This function reads the state of the interrupt registers in the AAL1gator-32 and stores them into an ISV. Performs functions needed to clear the interrupt, from simply clearing bits to complex functions. It then sends this ISV via a message queue or other USER defined method to the DPR task. This routine is called by the application code, from within `al3ISRHandler`.

Prototype `void *al3ISR (sDEV_HNDL devId, void *pBuf)`

Inputs

`devId`: device Handle (from `al3Add`)
`pBuf`: (pointer to) a preallocated ISV

Outputs ISR state via 'pBuf'

Returns pBuf

Valid States (MOD_READY) ACTIVE INACTIVE

Side Effects None

Device Processing Routine: al3DPR

This function acts on data contained in an ISV, creates a DPV, invoking application code callbacks (if defined and enabled) and possibly performing linked actions. The `al3DPR` calls from within the application function `al3DPRTask`.

Prototype `sAL3_DPV *al3DPR(void *pBuf)`

Inputs `pBuf`: ISV buffer (from al3ISR())

Outputs None

Returns If `pBuf` pointed to a user allocated buffer then, a pointer to the buffer

Else, a NULL pointer

Valid States (MOD_READY) PRESENT ACTIVE INACTIVE

Side Effects None

5.20 Counter Functions

Retrieving Statistical Counts: al3GetCounter

This function retrieves all the statistical counts that are kept in the Device Status Block (DSB).

Prototype `INT4 al3GetCounter (sDEV_HNDL devId, sAL3_CNTR_SPEC *pSpec, sAL3_DSB *pDSB, BOOLEAN update)`

Inputs

`devId`: device Handle (from `al3Add`)
`pSpec`: (pointer to) parameter block
`pDSB`: (pointer to) space to return DSB
`update`: if set, update from hardware

Outputs current DSB via pDSB

Returns Success = `AL3_OK`

Failure = `<AL3 ERROR CODES>`

Valid States (MOD_READY) ACTIVE INACTIVE

Side Effects None

Retrieving Statistical Counts: al3GetStats

This function retrieves all statistical counts kept in the Device Status Block (DSB). It is the USER's responsibility to ensure that the pointer points to an area of memory large enough to hold a copy of the DSB.

Prototype `INT4 al3GetStats (sDEV_HNDL devId, sAL3_DSB *pDSB)`

Inputs

`devId`: device Handle (from `al3Add`)
`pDSB`: (pointer to) device status block

Outputs None

Returns Success = `AL3_OK`

Failure = <`AL3 ERROR CODES`>

Valid States (MOD_READY) ACTIVE INACTIVE

Side Effects None

Clearing Statistical Counts: al3ClearStats

This function clears the statistical counts inside the Device Status Block (DSB). Passed structure non-zero fields correspond to the cleared counters.

Prototype `INT4 al3ClearStats (sDEV_HNDL devId, sAL3_DSB* pBuf)`

Inputs

`devId`: device Handle (from `al3Add`)
`pBuf`: DSB structure used as a key

Outputs None

Returns Success = `AL3_OK`

Failure = <`AL3 ERROR CODES`>

Valid States (MOD_READY) ACTIVE INACTIVE

Side Effects None

5.21 Device Diagnostics

Testing A Single Device Register: al3TestReg

This function verifies the hardware access to a device register by writing and reading back values as well as detecting parity errors.

Prototype `INT4 al3TestReg (sDEV_HNDL devId, UINT4 regNum)`

Inputs

`devId`: device Handle (from `al3Add`)
`regNum`: register number to test

Outputs None

Returns Success = `AL3_OK`

Failure = <`AL3 ERROR CODES`>

Valid States (MOD_READY) PRESENT

Side Effects May Set / Clear any register in the Device

Testing Device Registers: al3TestRegs

This function verifies the hardware access to device registers by writing and reading back values as well as detecting parity errors.

Prototype `INT4 al3TestRegs (sDEV_HNDL devId)`

Inputs `devId`: device Handle (from `al3Add`)

Outputs None

Returns Success = `AL3_OK`

Failure = <`AL3 ERROR CODES`>

Valid States (MOD_READY) PRESENT

Side Effects May Set / Clear any register in the Device

Testing Data Bus Wiring: al3TestDataBus

This function tests the data bus wiring between the AAL1gator-32 CPU, and SRAMs by performing a walking 1's test on every location in the AAL1gator-32 device's memory space.

Prototype `INT4 al3TestDataBus(sDEV_HNDL devId, UINT4 firstAddr, UINT4 lastAddr)`

Inputs
`devId`: device Handle (from `al3Add`)
`firstAddr`: starting Address for test
`lastAddr`: ending Address for test

Outputs None

Returns Success = `AL3_OK`

Failure = `<AL3 ERROR CODES>`

Valid States (MOD_READY) PRESENT

Side Effects Clears RAM and any Device configuration

Testing Address Bus Wiring: al3TestAddrBus

This function tests the address bus wiring between the AAL1gator-32 CPU, and SRAMs by performing a walking 1's test on the relevant bits of the address and checking for aliasing.

Prototype
```
INT4 al3TestAddrBus(sDEV_HNDL devId, UINT4 firstAddr,
UINT4 lastAddr, UINT2 testConst)
```

Inputs
devId: device Handle (from `al3Add`)
`firstAddr`: first address
`lastAddr`: last address
`testConst`: data value to use for testing

Outputs None

Returns Success = `AL3_OK`

Failure = `<AL3 ERROR CODES>`

Valid States (MOD_READY) PRESENT

Side Effects Clears RAM and any Device configuration

5.22 Callback Functions

The AAL1gator-32 driver has the capability to callback functions within the USER code when certain events occur. These events and their associated callback routine declarations are detailed below. There is no USER code action that is required by the driver for these callbacks; the USER is free to implement these callbacks in any manner or else they can be deleted from the driver.

A1SP Callbacks: cbackA1SP

This callback function is provided by the USER and is used by the DPR to report A1SP events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements. The USER should free the DSB buffer.

Prototype	`void cbackA1SP(sAL3_DPV *pcurrDPV)`
Inputs	`pcurrDPV`: pointer to current DPV received from DPR
Outputs	None
Returns	None
Valid States	ACTIVE
Side Effects	None

Utopia Callbacks: cbackUtopia

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

Prototype	`INT4 cbackUtopia(sAL3_DPV *pcurrDPV)`
Inputs	`pcurrDPV`: pointer to current DPV received from DPR
Outputs	None
Returns	None
Valid States	ACTIVE
Side Effects	None

RAM Callbacks: cbackRam

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

Prototype	`INT4 cbackRAM(sAL3_DPV *pcurrDPV)`
Inputs	`pcurrDPV`: pointer to current DPV received from DPR

Outputs None

Returns None

Valid States ACTIVE

Side Effects None

SBI Callbacks: cbackSBI

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

Prototype `void cbackSBI(sAL3_DPV *pcurrDPV)`

Inputs `pcurrDPV`: pointer to current DPV received from DPR

Outputs None

Returns None

Valid States ACTIVE

Side Effects None

6 HARDWARE INTERFACE

The AAL1gator-32 driver interfaces directly with the USER's hardware. In this section, a listing of each point of interface is shown, along with a declaration and any specific porting instructions. It is the responsibility of the USER to connect these requirements into the hardware, either by defining a macro or by writing a function for each item listed. Take care when matching parameters and return values.

6.1 Device I/O

Safe Reading from Registers: sysAl3SafeReadReg

This function reads the contents of a specific register location. This macro/function should be UINT2 oriented and should be defined by the user to reflect the target system's addressing logic. This function is expected to have error recovery since this function is used to access the device first.

Prototype `#define sysAl3SafeReadReg(baseAddr, offset),UINT2 sysAl3SafeReadReg(UINT2 * baseAddr, UINT4 offset)`

Inputs
`baseAddr`: base Address of the Device
`offset`: offset from 'baseAdd' for this read

Outputs
`pData`: data read placed into this (pointed to) variable

Returns
Success = data read
Failure = <no convention yet set>

Reading from Registers: sysAl3ReadReg

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

Prototype `#define sysAl3ReadReg(baseAddr, offset), UINT2 sysAl3ReadReg(UINT2 * baseAddr, UINT4 offset)`

Inputs
`baseAddr`: base Address of the Device
`offset`: offset from 'baseAdd' for this read

Outputs None

Returns Always = data read

Writing to Registers: sysAl3WriteReg

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

Prototype `#define sysAl3WriteReg(baseAddr, offset, data), void sysAl3WriteReg(UINT2 * baseAddr, UINT4 offset, UINT2 data)`

Inputs
`baseAddr`: base Address of the Device
`offset`: offset from 'baseAdd' for this read
`data`: data to be written

Outputs None

Returns Always = data written

6.2 Interrupt Servicing

This section describes the platform specific routines that are required by the AAL1gator-32 driver AND provided by the USER. Details are given with each routine.

Installing Handlers: sysAl3ISRHandlerInstall

This function installs the USER-supplied Interrupt Service Routine (ISR), `sysAl3ISRHandler`, into the processor's interrupt vector table.

Prototype `INT4 sysAl3ISRHandlerInstall(void )`

Inputs None

Outputs None

Returns Success = `AL3_OK`

Failure = `<AL3 ERROR CODES>`

Invoking Handlers: sysAl3ISRHandler

This function is invoked when one or more AAL1gator-32 devices raise the interrupt line to the microprocessor. This routine invokes the driver-provided routine, `al3ISR`, for each device registered with the driver.

Prototype `void sysAl3ISRHandler (INT4 irq)`

Inputs None

Outputs None

Returns Success = `AL3_OK`

Failure = <AL3 ERROR CODES>

Removing Handlers: sysAl3ISRHandlerRemove

This function disables the Interrupt processing for this device. Removes the USER-supplied Interrupt Service routine (ISR), `sysAl3ISRHandler`, from the processor's interrupt vector table.

Prototype `INT4 sysAl3ISRHandlerRemove(void)`

Inputs None

Outputs None

Returns Success = `AL3_OK`

Failure = <AL3 ERROR CODES>

Invoking DPR Routines: sysAl3DPRTask

This routine is spawned as a separate task within the RTOS. It runs periodically and retrieves interrupt status information saved for it by the `al3ISRHandler` routine and then invokes the `al3DPR` routine for the appropriate device.

Prototype `void sysAl3DPRTask (void)`

Inputs None

Outputs None

Returns None

Starting the DPR Tasks: sysAl3DPRTaskStart

This routine invokes the DPR task. This routine is called in `al3ModuleStart`.

Prototype `INT4 sysAl3DPRTaskStart (void *dprFuncAddr)`

Inputs None

Outputs None

Returns Success = 0x00

Failure = non-zero

Stopping the DPR Tasks: sysAl3DPRTaskStop

This routine deletes the DPR task. This routine is called in `al3ModuleStop`.

Prototype `void sysAl3DPRTaskStop (void)`

Inputs None

Outputs None

Returns None

Starting Statistics Task: sysAl3StatTask

This routine is spawned as a separate task within the RTOS. It runs periodically and retrieves hardware statistics and updates software statistics in DSB accordingly. The period of this task is defined by `statUpdatePeriod` in the DDB.

Prototype `void sysAl3StatTask (void)`

Inputs None

Outputs None

Returns None

Starting Statistics Task: sysAl3StatTaskStart

This routine spawns the Stats task. This routine is called in `al3ModuleStart`.

Prototype `INT4 sysAl3StatTaskStart (void *statFuncAddr)`

Inputs None

Outputs None

Returns Success = `AL3_OK`

Failure = `<AL3 ERROR CODES>`

Stopping Statistic Updates: sysAl3StatTaskStop

This routine deletes the Stats task. This routine is called in `al3ModuleStop`.

Prototype `void sysAl3StatTaskStop (void)`

Inputs None

Outputs None

Returns Success = `AL3_OK`

Failure = `<AL3 ERROR CODES>`

7 RTOS INTERFACE

The AAL1gator-32 driver requires the use of some RTOS resources. In this section, a listing of each required resource is shown, along with a declaration and any specific porting instructions. It is the responsibility of the USER to connect these requirements into the RTOS, either by defining a macro or writing a function for each item listed. Care should be taken when matching parameters and return values.

7.1 Memory Allocation/De-Allocation

Allocating Memory: sysAl3MemAlloc

This function allocates specified number of bytes of memory.

Prototype `#define sysAl3MemAlloc(numBytes), UINT1 *sysAl3MemAlloc(UINT4 numBytes)`

Inputs `numBytes`: number of bytes to be allocated

Outputs None

Returns Pointer to first byte of allocated memory

NULL pointer (memory allocation failed)

Freeing Memory: sysAl3MemFree

This function frees memory allocated using `sysAl3MemAlloc`.

Prototype `#define sysAl3MemFree(pFirstByte), void sysAl3MemFree(UINT1 *pFirstByte)`

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

Outputs None

Returns None

7.2 Buffer Management

All operating system provides some sort of buffer system, particularly for use in sending and receiving messages. The following calls, provided by the USER, allow the Driver to Get and Return buffers from the RTOS. It is the USER's responsibility to create any special resources or pools to handle buffers of these sizes during the `sysAl3BufferStart` call.

Starting Buffers: sysAl3BufferStart

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

Prototype

```
#define sysAl3BufferStart()
INT4 sysAl3BufferStart(void)
```

Inputs None

Outputs None

Returns AL3_OK
AL3_FAIL

Getting Buffers: sysAl3DPVBufferGet

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

Prototype

```
#define sysAl3DPVBufferGet()
sAL3_ISV * sysAl3ISVBufferGet(void)
```

Inputs None

Outputs None

Returns Success = (pointer to) a ISV buffer

Failure = NULL (pointer)

Getting Buffers: sysAl3ISVBufferGet

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

Prototype
```
#define sysAl3ISVBufferGet()
sAL3_ISV *sysAl3ISVBufferGet(void)
```

Inputs None

Outputs None

Returns Success = (pointer to) a ISV buffer

Failure = NULL (pointer)

Sending Buffers: sysAl3BufferSend

This function sends a buffer, through regular message channels, to the DPR task handler `sysAl3DPRTask`.

Prototype
```
#define sysAl3BufferSend(pISV)
INT4 sysAl3BufferSend (sAL3_ISV *pISV)
```

Inputs `pISV`: (pointer to) buffer to send

Outputs None

Returns Success = `0x00`

Failure = (-1)

Receiving Buffers: sysAl3BufferReceive

This function receives a DPV/ISV buffer from the RTOS.

Prototype
```
#define sysAl3BufferReceive ()
sAL3_ISV *sysAl3BufferReceive (void)
```

Inputs None

Outputs (pointer to) an ISV buffer

Returns None

Returning Buffers: sysAl3DPVBufferRtn

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

Prototype
```
#define sysAl3DPVBufferRtn(pDPV)
INT4 sysAl3DPVBufferRtn(sAL3_DPV *pdpv)
```

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

Outputs None

Returns Success = AL3_OK

Failure = AL3_FAIL

Returning Buffers: sysAl3ISVBufferRtn

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

Prototype
```
#define sysAl3ISVBufferRtn(pISV)
INT4 sysAl3ISVBufferRtn(sAL3_ISV *pisv)
```

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

Outputs None

Returns Success = AL3_OK

Failure = AL3_FAIL

Stopping Buffers: sysAl3BufferStop

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

Prototype
```
#define sysAl3BufferStop()
void sysAl3BufferStop (void)
```

Inputs None

Outputs None

Returns None

7.3 Timers

Creating Timer Objects: sysAl3TimerCreate

This function creates a timer object for general use.

Prototype
```
#define sysAl3TimerCreate()
void *sysAl3TimerCreate (void)
```

Inputs None

Outputs None

Returns Success = (pointer to) a timer object

Failure = NULL (pointer)

Starting Timers: sysAl3TimerStart

This function starts a timer.

Prototype
```
#define sysAl3TimerStart(pTimer, period, pFunc)
INT4 sysAl3TimerStart (void *ptimer, UINT4 period, void
*pfunc, INT4 arg)
```

Inputs

`ptimer`: (pointer to) timer object
`period`: time (in milliseconds)
`pfunc`: function to invoke when timer expires

Outputs None

Returns None

Aborting Timers: sysAl3TimerAbort

This function aborts a running timer.

Prototype
```
#define sysAl3TimerAbort(pTimer)
void sysAl3TimerAbort (void *ptimer)
```

Inputs `ptimer`: (pointer to) timer object

Outputs None

Returns AL3_OK

Deleting Timers: sysAl3TimerDelete

This function deletes a timer.

Prototype
```
#define sysAl3TimerDelete(pTimer)
void sysAl3TimerDelete (void *ptimer)
```

Inputs `ptimer`: (pointer to) timer object

Outputs None

Returns None

Suspending a Task: sysAl3TimerSleep

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

Prototype
```
#define sysAl3TimerSleep(time)
void sysAl3TimerSleep (UINT4 msec)
```

Inputs `msec`: sleep time in milliseconds

Outputs None

Returns None

7.4 Semaphores

Creating Semaphores: sysAl3SemCreate

This function creates an integer semaphore object.

Prototype
```
#define sysAl3SemCreate()
void *sysAl3SemCreate(void)
```

Inputs None

Outputs None

Returns Success = (pointer to) a semaphore object

Failure = NULL (pointer)

Taking Semaphores: sysAl3SemTake

Takes an integer semaphore.

Prototype
```
#define sysAl3SemTake(psem)
void sysAl3SemTake(void *psem)
```

Inputs `psem`: (pointer to) a semaphore object

Outputs None

Returns AL3_SUCCESS
AL3_FAILURE

Giving Semaphores: sysAl3SemGive

This function gives an integer semaphore.

Prototype
```
#define sysAl3SemGive(psem)
void sysAl3SemGive(void *psem)
```

Inputs `psem` : (pointer to) a semaphore object

Outputs None

Returns AL3_SUCCESS
AL3_FAILURE

Deleting Semaphores: sysAl3SemDelete

This function deletes an integer semaphore object.

Prototype
```
#define sysAl3SemDelete(psem)
void sysAl3SemDelete(void *psem)
```

Inputs `psem` : (pointer to) a semaphore object

Outputs None

Returns AL3_SUCCESS
AL3_FAILURE

7.5 Preemption

Disabling Preemption: sysAl3PreemptDisable

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

Prototype
```
#define sysAl3PreemptDisable ()
INT4 sysAl3PreemptDisable(void)
```

Inputs None

Outputs None

Returns Pre-emption key (passed back as an argument in sysAl3PreemptEn)

Disabling Preemption: sysAl3PreemptEnable

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

Prototype
```
#define sysAl3PreemptEnable (key)
void sysAl3PreemptEnable(INT4 key)
```

Inputs key - pre-emption key (returned by sysAl3PreemptEn)

Outputs None

Returns None

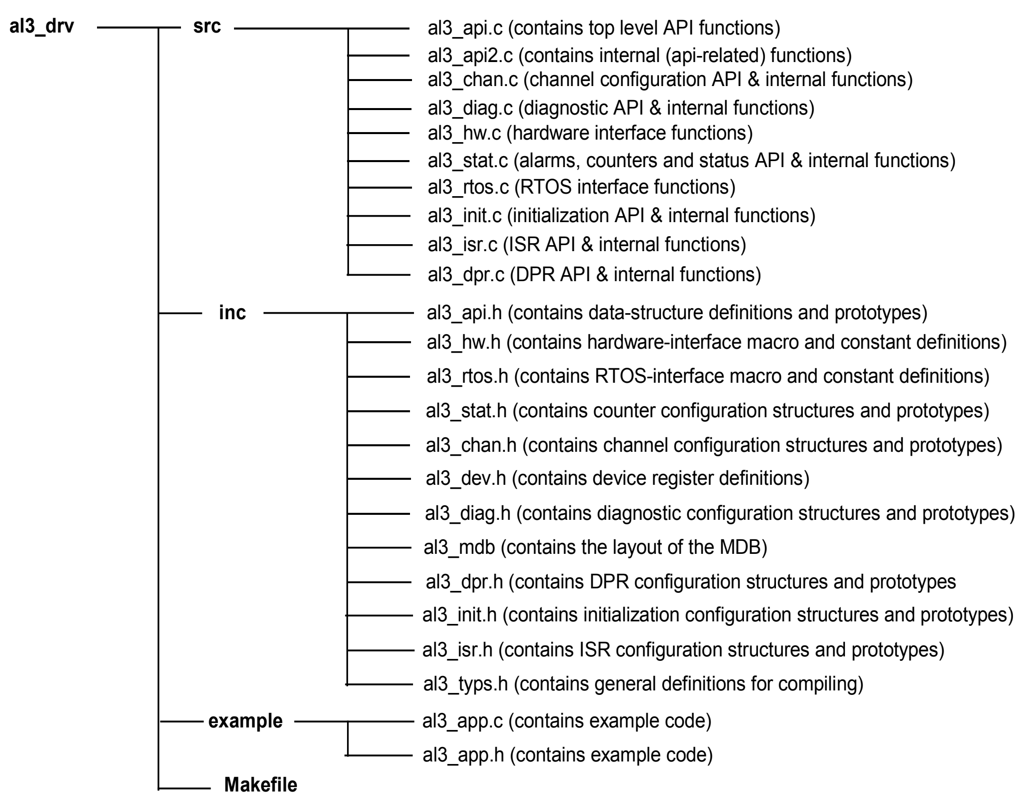
8 PORTING DRIVERS

This section outlines how to port the AAL1gator-32 device driver to your hardware and RTOS platform. However, this manual can offer only guidelines for porting the AAL1gator-32 driver because each platform and application is unique.

8.1 Driver Source Files

The C source files listed below contain the code for the AAL1gator-32 driver. You may need to modify the code or develop additional code. The code is in the form of constants, macros, and functions. For ease of porting, the code is grouped into source files (`src`) and include files (`inc`). The src files contain the functions and the inc files contain the constants and macros.

Figure 9: Driver Source Files



8.2 Driver Porting Procedures

The following procedures summarize how to port the AAL1gator-32 driver to your platform. The subsequent sections describe these procedures in more detail.

To port the AAL1gator-32 driver to your platform:

Procedure 1: Port the driver's RTOS extensions (page 116):

Procedure 2: Port the driver to your hardware platform (page 118):

Procedure 3: Port the driver's application-specific elements (page 119):

Procedure 4: Build the driver (page 120).

Procedure 1: Porting Driver RTOS Extensions

The RTOS extensions encapsulate all RTOS specific services and data types used by the driver. The `al3_typs.h` file contains data types and compiler-specific data-type definitions. The `al3_rtos.h` & `al3_rtos.c` files contain macros and functions for RTOS specific services used by the Driver. These RTOS services include:

- Memory Management
- Buffer Management
- Timers
- Task Management
- Semaphores

To port the driver's OS extensions:

1. Modify the data types in `al3_typs.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 macros in `al3_rtos.h` and/or the RTOS specific functions in `al3_rtos.c`. The flag '`USE_RTOS_MACROS`' (in `al3_rtos.h`) enables the macros in `al3_rtos.h` and disables the functions in `al3_rtos.c`. By default this flag is set. Clear this flag if you prefer to use the functions instead of macros. The following table outlines the macros/functions that need to be defined/coded:

Service Type	Macro Name	Description
Memory	`sysAl3MemAlloc`	Allocates a memory block
	`sysAl3MemFree`	Frees a memory block
	`sysAl3MemCopy`	Sets a memory block to one value
	`sysAl3MemCopy`	Copies a memory block
	`sysAl3BufferStart`	Allows the Application to pre-setup buffer pools for both ISV and DPV buffers

Service Type	Macro Name	Description
	`sysAl3DPVBufferGet`	Returns a DPV Buffer to the driver from the Application's buffer pool
	`sysAl3ISVBufferGet`	Returns a ISV Buffer to the driver from the Application's buffer pool
	`sysAl3BufferSend`	Allows the Application to choose the method for sending each initialized ISV from the ISR code to the DPR Task
	`sysAl3BufferReceive`	Allows the Application to choose the method for receiving each initialized ISV from the ISR code to the DPR Task
	`sysAl3DPVBufferRtn`	Returns a DPV to the Application's DPV pool
	`sysAl3ISVBufferRtn`	Returns a ISV to the Application's ISV pool
	`sysAl3BufferStop`	Allows the Application to clean-up and/or deallocate both the DPV and ISV buffer pools
Timer	`sysAl3TimerCreate`	Creates a new Timer for use by the driver
	`sysAl3TimerStart`	Starts a timer
	`sysAl3TimerAbort`	Aborts a timer
	`sysAl3TimerDelete`	Deletes a timer
	`sysAl3TimerSleep`	Causes a timer to trigger after a specified period of time

Service Type	Macro Name	Description
DPR / Statistics Management	sysAl3DPRTaskStart	Allows the Application to install/start the DPR task
	sysAl3DPRTask	Allows the Application to control the DPR Task
	sysAl3DPRTaskStop	Allows the Application to de-install/stop the DPR Task
	sysAl3StatTaskStart	Allows the Application to install/start the STAT task
	sysAl3StatTask	Allows the Application to control the STAT Task
	sysAl3StatTaskStop	Allows the Application to deinstall/stop the STAT Task
Semaphore	sysAl3SemCreate	Create an integer semaphore
	sysAl3SemTake	Sets an integer semaphore
	sysAl3SemGive	Clears an integer semaphore
	sysAl3SemDelete	Deletes an integer semaphore

Procedure 2: Porting Drivers to Hardware Platforms

This section describes how to modify the AAL1gator-32 driver for your hardware platform.

To port the driver to your hardware platform:

1. Define the Hardware system-configuration constants in the `al3_hw.h` file. Modify the following constants to reflect your system's hardware configuration:

Device Constant	Description	Default
AL3_SHIFT	Adjusts the `al3ReadXXX` and `al3WriteXXX` macros for address bus width	1

2. Modify the Hardware specific macros in `al3_hw.h` and/or the Hardware specific functions in `al3_hw.c`. The flag 'USE_HW_MACROS' (in `al3_hw.h`) enables the macros in `al3_hw.h` and disables the functions in `al3_hw.c`. By default this flag is set. Clear this flag if you prefer to use the functions instead of the macros. The following table outlines the macros/functions that need to be defined/coded:

Service Type	Macro Name	Description
Read/Write	sysAl3SafeReadReg	Create an integer semaphore
	sysAl3ReadReg	Sets an integer semaphore
	sysAl3WriteReg	Clears an integer semaphore
ISR	sysAl3ISRHandlerInstall	Installs the ISR Handler
	sysAl3ISRHandler	Services each ISR
	sysAl3ISRHandlerRemove	Removes the ISR Handler

Procedure 3: Porting Driver Application-Specific Elements

Application specific elements are configuration constants and callback functions used by the API for developing an application. This section describes how to modify the application specific elements in the AAL1gator-32 driver.

To port the driver's application-specific elements:

1. Define the following driver task-related callback functions. Each function can be defined for the Driver by passing its address via an initialization profile. The use of each callback is optional. Passing a NULL in place of the function's address disables the Driver's use of that function. The following table lists the callbacks that may be used by the application:

CallBack Function	Description
sysAl3CbackRAM	Handles events that relate to the RAM section of the Device
sysAl3CbackSBI	Handles events that relate to the SBI section of the Device
sysAl3CbackA1SP	Handles events that relate to the A1SP section of the Device
sysAl3CbackUtopia	Handles events that relate to the Utopia Bus section of the Device

Procedure 4: Building Drivers

This section describes how to build the AAL1gator-32 driver.

To build the driver:

2. Ensure that the directory variable names in the makefile reflect your actual driver and directory names.
3. Compile the source files and build the AAL1gator-32 API driver library using your make utility.
4. Link the AAL1gator-32 API driver library to your application code.

APPENDIX A: CODING CONVENTIONS

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

Variable Type Definitions

Table 37: Variable Type Definitions

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

Naming Conventions

Table 38 presents a summary of the naming conventions followed by all PMC driver software. A detailed description follows the 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 38: Naming Conventions

Type	Case	Naming convention	Examples
Macros	Uppercase	prefix with "m" and device abbreviation	`mAL3_WRITE`
Constants	Uppercase	prefix with device abbreviation	`AL3_REG`
Structures	Hungarian Notation	prefix with "s" and device abbreviation	`sAL3_DDB`
API Functions	Hungarian Notation	prefix with device name	`al3Add`
Porting Functions	Hungarian Notation	prefix with "sys" and device name	`sysAl3RawRead()`
Static Functions	Hungarian Notation		`MyStaticFunction()`
Variables	Hungarian Notation		`maxDevs`
Pointers to variables	Hungarian Notation	prefix variable name with "p"	`pmaxDevs`
Global variables	Hungarian Notation	prefix with device name	`al3Mdb`

Macros

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

- Macro names must be all uppercase.
- Words shall be separated by an underscore.
- The letter "m" in lowercase is used as a prefix to specify that it is a macro, then the device abbreviation must appear.
- Example: `mAL3_WRITE` is a valid name for a macro.

Constants

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

- Constant names must be all uppercase.
- Words shall be separated by an underscore.
- The device abbreviation must appear as a prefix.
- Example: `AL3_REG` is a valid name for a constant.

Structures

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

- Structure names must be all uppercase.
- Words shall be separated by an underscore.
- The letter "s" in lowercase must be used as a prefix to specify that it is a structure, then the device abbreviation must appear.
- Example: `sAL3_DDB` is a valid name for a structure.

Functions

API Functions

- Naming of the API functions must follow the hungarian notation.
- The device's full name in all lowercase shall be used as a prefix.
- Example: `al3Add`() is a valid name for an API function.

Porting Functions

- Porting functions correspond to all function that are hardware and/or RTOS dependant.
- Naming of the porting functions must follow the hungarian notation.
- The "sys" prefix shall be used to indicate a porting function.
- The device's name starting with an uppercase must follow the prefix.
- Example: `sysAl3RawRead()` is a hardware/RTOS specific.
- Static Functions
- Static Functions are internal functions and have no special naming convention. However, they must follow the hungarian notation.
- Example: `myDummyFunction()` is a valid name for an internal function.

Variables

- Naming of variables must follow the hungarian notation.
- A pointer to a variable shall use "p" as a prefix followed by the variable name unchanged. If the variable name already starts with a "p", the first letter of the variable name may be capitalized, but this is not a requirement. Double pointers might be prefixed with "pp", but this is not required.
- Global variables must be identified with the device's name in all lowercase as a prefix.
- Examples: `maxDevs` is a valid name for a variable, `pmaxDevs` is a valid name for a pointer to `maxDevs`, and `al3BaseAddress` 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 39 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. `al3_api.c` or `al3_api2.c`).

There are 5 different types of files:

- The Generic API files containing all the generic API functions (`al3ModuleOpen`, `al3Add`, `al3Activate`, etc..)
- Device Specific API files containing device specific API functions (Initialization, Stats, etc…)
- The hardware file containing the hardware dependent functions
- The RTOS file containing the RTOS dependent functions
- The other files containing all the remaining functions of the driver

Table 39: File Naming Conventions

File Type	File Name
Generic API	al3_api.c, al3_api.h
Device Specific API	al3_dpr.c, al3_isr.c, al3_diag.c, al3_init.c, al3_chan.c, al3_stat.c, al3_dpr,h, al3_isr.h, al3_diag.h, al3_init.h, al3_chan.h, al3_stat.h
Hardware Dependent	al3_hw.c, al3_hw.h
RTOS Dependent	al3_rtos.c, al3_rtos.h
Other	al3_dev.h, al3_mdb.h

Generic API Files

- The name of the Generic 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: `al3_api.c` is the only valid name for the file that contains the generic API functions. `al3_api.h` is the only valid name for the file that contains all of the generic API functions headers.

Device Specific API Files

- The name of the Device Specific API files must start with the device abbreviation followed by an underscore and a descriptive ending that relates to the functionality within.

- Examples: `al3_chan.c` is the name for the file that contains API and internal functions for configuring Channels in the device. `al3chan.h` is the name of the file that contains the constants and declarations for the channel configuration functions.

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: `al3_hw.c` is the only valid name for the file that contains all of the hardware dependent functions. `al3_hw.h` is the only valid name for the file that contains all of the hardware dependent functions headers.

RTOS Dependent Files

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

- Examples: `al3_rtos.c` is the only valid name for the file that contains all of the RTOS dependent functions, `al3_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: `al3_dev.h` is a valid name for a file that would deal with register map within the Device and `al3_mdb.h` is a valid name for a file that lays out the structure of the MDB.

APPENDIX B: ERROR CODES

The following describes the error codes used in the AAL1gator-32 device driver:

Error Code	Description
AL3_OK	Success
AL3_FAIL	Failure
AL3_ERR_HW	
AL3_ERR_SEM	
AL3_ERR_FREE	
AL3_ERR_READ	
AL3_ERR_RTOS	
AL3_ERR_ALLOC	Memory allocation failure
AL3_ERR_TIMER	Timer management error
AL3_ERR_WRITE	
AL3_ERR_BUFFER	Buffer management error
AL3_ERR_OPEN	Internal call to ModuleOpen failed
AL3_ERR_STOP	Internal call to ModuleStop failed
AL3_ERR_CLOSE	Internal call to ModuleClose failed
AL3_ERR_START	Internal call to ModuleStart failed
AL3_ERR_ISOPEN	Module is already open
AL3_ERR_STOPED	Module is currently closed
AL3_ERR_CLOSED	Module is currently stoped
AL3_ERR_ADD	Internal call to Add failed
AL3_ERR_INIT	Internal call to Init failed
AL3_ERR_RESET	Internal call to Reset failed
AL3_ERR_DELETE	Internal call to Delete failed
AL3_ERR_UPDATE	Internal call to Update failed
AL3_ERR_ACTIVATE	Internal call to Activate failed
AL3_ERR_DEACTIVATE	Internal call to DeActivate failed
AL3_ERR_ISIDLE	Module is already in the IDLE state
AL3_ERR_ISREADY	Module is already in the READY state
AL3_ERR_ISSTART	Module is already in the START state
AL3_ERR_ISACTIVE	Device is already in the ACTIVE state

Error Code	Description
AL3_ERR_ISPRESENT	Device is already in the PRESENT state
AL3_ERR_ISINACTIVE	Device is already in the INACTIVE state
AL3_ERR_NOTIDLE	Module not in the IDLE state
AL3_ERR_NOTREADY	Module not in the READY state
AL3_ERR_NOTSTART	Module not in the START state
AL3_ERR_NOTACTIVE	Device not in the ACTIVE state
AL3_ERR_NOTPRESENT	Device not in the PRESENT state
AL3_ERR_NOTINACTIVE	Device not in the INACTIVE state
AL3_ERR_ARG	Invalid argument
AL3_ERR_CFG	Invalid configuration
AL3_ERR_MDB	Module is invalid
AL3_ERR_ADDR	Invalid address
AL3_ERR_HNDL	Invalid device handle
AL3_ERR_MODE	Invalid mode
AL3_ERR_RANGE	Incorrect range
AL3_ERR_HWFAIL	Hardware failure
AL3_ERR_RAMFAIL	RAM failure
AL3_ERR_TIMEOUT	Timed out while polling
AL3_ERR_INUSE	Already in use
AL3_ERR_MAXPROF	Maximum profile already added
AL3_ERR_MAXDEVICE	Maximum device already reached
AL3_ERR_ARRAY_FULL	Array is full
AL3_ERR_CHAN_INUSE	Chain already in use
AL3_ERR_DEV_EXISTS	Device already exists
AL3_ERR_QUEUE_INUSE	Queue already in use

APPENDIX C: AAL1GATOR-32 EVENTS

This appendix describes the events used in the AAL1gator-32 device driver:

SBI Alarm Events

Event Code	Description
AL3_SBI_ALARMH_EVENT	SBI alarm state has changed for a high link
AL3_SBI_ALARML_EVENT	SBI alarm state has changed for a low link

SBI Extract Events

Event Code	Description
AL3_EXT_INS_DC_EVENT	Depth Check error has been detected
AL3_EXT_C1FP_EVENT	C1FP realignment has been detected
AL3_EXT_SYNC_EVENT	SBIIP_SYNC realignment has been detected
AL3_EXT_FIFO_UDR_EVENT	FIFO underrun has been detected
AL3_EXT_FIFO_OVR_EVENT	FIFO overrun has been detected
AL3_EXT_SBI_PERR_EVENT	SBI parity error has been detected

SBI Insert Events

Event Code	Description
AL3_INS_INS_DC_EVENT	Depth Check error has been detected
AL3_INS_C1FP_EVENT	C1FP realignment has been detected
AL3_INS_SYNC_EVENT	SBIIP_SYNC realignment has been detected
AL3_INS_FIFO_UDR_EVENT	FIFO underrun has been detected
AL3_INS_FIFO_OVR_EVENT	FIFO overrun has been detected

UTOPIA Events

Event Code	Description
AL3_UTOPIA_RX_RUNT_EVENT	A short cell (less than 53 bytes) has been received
AL3_UTOPIA_LFIFO_FULL_EVENT	UTOPIA Loopback FIFO is full
AL3_UTOPIA_TXFR_ERR_EVENT	Transmit UTOPIA interface has been requested to send a cell when it did not have one available
AL3_UTOPIA_TFIFO_FULL_EVENT	Transmit UTOPIA FIFO is full
AL3_UTOPIA_PAR_ERR_EVENT	Parity error encoutered in the UTOPIA interface

RAM Parity Events

Event Code	Description
AL3_RAM1_PAR_ERR_EVENT	Parity error encoutered in the RAM1 interface
AL3_RAM2_PAR_ERR_EVENT	Parity error encoutered in the RAM2 interface

A1SP Events

Event Code	Description
AL3_A1SP_TFIFO_FULL_EVENT	TALP FIFO is full
AL3_A1SP_RFIFO_FULL_EVENT	Receive Status FIFO is full
AL3_A1SP_RFIFO_EMPB_EVENT	Receive Status FIFO is empty
AL3_A1SP_IFIFO_FULL_EVENT	Transmit Idle State FIFO is full
AL3_A1SP_IFIFO_EMPB_EVENT	Transmit Idle State FIFO is empty
AL3_A1SP_OAM_EVENT	A1SP block has received a new OAM cell
AL3_A1SP_FFIFO_FULL_EVENT	Frame advance FIFO is full
AL3_RFIFO_R_LINE_RESYNC_EVENT	Receive line has entereed a resync state

Event Code	Description
AL3_RFIFO_T_LINE_RESYNC_EVENT	Transmit line has entereed a resync state
AL3_RFIFO_BITMASK_CHANGE_EVENT	Bitmask for active channels has changed
AL3_RFIFO_EXIT_UNDERRUN_EVENT	Queue just exited the underrun state
AL3_RFIFO_ENTER_UNDERRUN_EVENT	Queue just entered the underrun state
AL3_RFIFO_RECEIVE_QUEUE_ERR_EVENT	Error or status condition occurred on the receive queue (check sticky bit)

ACRONYMS

AAL: ATM Adaptation Layer

AAL1: ATM Adaptation Layer 1

API: Application Programming Interface

BERT: Bit error-rate test

BOOL: Boolean data type

CBR: Constant Bit Rate

CES: Circuit Emulation Service

DDB: Device Data Block

DIV: Device Initialization Vector

DPR: Deferred Processing Routine

DSB: DEVICE Status Block

FCS: Frame check sequence

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

MDB: Module Data Block

MIV: Module Initialization Vector

MSB: Module Status Block

MVIP: Multi-vendor integration protocol

PCI: Processor connection interface

PHY: Physical layer

RAPI: Receive Any-PHY packet interface

RCAS: Receive channel assignor

RHDL: Receive HDLC processor

RMAC: Receive memory access controller

RTOS: Real-Time operating system

SAR: Segmentation and Reassembly

SBI Interface: Scaleable bandwidth interconnect interface

SCD Interface: Serial clock and data interface

TAPI: Transmit Any-PHY packet interface

TCAS: Transmit channel assignor

THDL: Transmit HDLC processor

TMAC: Transmit memory access controller

LIST OF TERMS

APPLICATION: Refers to protocol software used in a real system as well as validation software written to validate the AAL1gator-32 driver on a validation platform.

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

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

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

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

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

DEVICE : ONE AAL1gator-32 Integrated Circuit. There can be many Devices, all served by this ONE Driver MODULE

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

MODULE: All of the code that is part of this driver, there is only ONE instance of this MODULE connected to ONE OR MORE AAL1gator-32 chips.

- MIV (MODULE Initialization Vector): Structure passed from the API to the MODULE during initialization, it contains parameters that identify the specific characteristics of the Driver MODULE being initialized.
- MDB (MODULE Data Block): Structure that holds the Configuration Data for this MODULE.
- MSB (MODULE Status Block): Structure that holds the Alarms, Status and Statistics for the MODULE

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

INDEX

F

T