PM73121

AAL1gator II DRIVER

EIGHT LINK CIRCUIT EMULATION SERVICE ON A CHIP DRIVER

USER'S MANUAL

Preliminary
Issue 2: November 1998

NOTE: The PM73121 AAL1gator II device contains SRTS logic for which Bellcore holds the patent. Please refer to the NOTE on page 39 for more information regarding Bellcore's SRTS patent.

Public Revision History

Issue Number	Issue Date	Details of Change
Issue 2	November 1998	• Throughout manual, changed "AAL1gator" references to "AAL1gator II". • Modified "What Should You Read?" on page 1. • Merged the Product Overview into Chapter 2, AAL1gator II Driver Overview. • Added the section "How the AAL1gator II Driver Interacts with Applications" on page 6. • Modified Figure 2 on page 12. • Added the section "OS Extensions" on page 19. • Removed the subsections "State S4" and "State S5" from the section "The Driver States" starting on page 7. • Removed part 2 of Figure 1 on page 8. • Moved data structures to "The Driver Data Structures" starting on page 15. • Added the AAL1 Enhanced Setup Parameter Structure on page 18. • Added Figure 5 on page 20. • To "Step 3: Port OS Extensions" on page 22, added the second paragraph. • To "Step 3b: Modify the oextport.h File" on page 22, added the first paragraph. • To "Step 3c: Code the oextport.c File" on page 23, added the NOTE. • Modified text in IMPORTANT note on page 23. • Added the returns `AAL1_LINENO_VC_MISMATCH` and `AAL1_NO_QUEUES` to the function "aal1ActivateChannel" on page 32. • Combined the Error Codes Common to All PMC Drivers table and the Error Codes Unique to the AAL1gator II Driver into one table: Table 10 on page 48. Deleted many error codes from table that are not applicable. • In Table 10 on page 48, corrected `AAL1_BAD_DEVICE_NO` to be `AAL1_BAD_DEV_NO`. • Added the API functions "aal1EnhancedActivateChannel" on page 33, "Additional Configuration Functions" on page 47, and "aal1SetMaxBuf" on page 47.
Issue 1	July 1998	Document created.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1
About this Manual

SCOPE

This user's manual describes the software driver for the PMC-Sierra Eight Link Circuit Emulation Service on a Chip device (the AAL1gator II™ Driver).

REFERENCES

This manual references the following documents:

- ANSI/ISO 9899-1990, *C Programming Language* standard (formerly, ANSI X3.159-1989).
- PMC-Sierra, *PM73121 AAL1gator II Long Form Data Sheet (document number PMC-980620).*

WHAT SHOULD YOU READ?

For the latest information about AAL1gator Driver issues...

Refer to the latest RELNOTES.TXT file included with the source code.

If you are an application programmer or system programmer. . .

Read Chapter 3, "AAL1gator II Driver Architecture", for a quick overview of how the driver operates. Then read Chapter 5, "AAL1gator II Driver API", to learn about the API functions specific to the AAL1gator II device.

If you will be porting the driver. . .

Read Chapter 4, "Porting Guidelines", for tips on compiling the driver on your host system, and bringing it up on your target system.

TYPOGRAPHICAL CONVENTIONS USED IN THIS MANUAL

Different fonts are used in this manual to help you understand what is explained. Table 1 describes how these different fonts are used.

Table 1. Conventions Used in this Manual

Font	Explanation	Example
Italic	Emphasis	The driver is fully tested and debugged so your initial testing will not involve *both* untested hardware and untested software.
`Courier`	Function names or Sample code	Code the `SigFn()` function. `#define MEM_FREE(`*a*`)` *xxx*

DEFINITIONS OF ACRONYMS USED IN THIS MANUAL

Table 2 lists the acronyms used in this manual and their definitions.

Table 2. Acronym Definitions

Acronym	Definition
AAL1	ATM Adaptation Layer 1
ANSI	American National Standards Institute
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
CAS	Channel Associated Signaling
CCB	Channel Control Block
CDV	Cell Delay Variation
CDVT	Cell Delay Variation Tolerance
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DCB	Device Control Block
DCF	Driver Control Function
DCT	Device Control Task
DDB	Device Data Block
DDG	Driver Diagnostic Function
DDL	Device Driver Library
DDO	Device Driver Operation
DRR	Driver Restart and Reinitialization Function
EPLD	Electrically Programmable Logic Device
GDDB	Global Device Driver Database
ISR	Interrupt Service Routine

Table 2. Acronym Definitions (Continued)

Acronym	Definition
LSB	Least Significant Bit
OAM	Operations, Administration, and Maintenance
OS	Operating System
pid	Process Identifier
RAM	Random Access Memory
ROM	Read Only Memory
RTOS	Real-Time Operating System
SN	Sequence Number
SNMP	Simple Network Management Protocol
SNP	Sequence Number Protection
SRTS	Synchronous Residual Time Stamp
UDF	Unstructured Data Format
UDF-HS	High-Speed Unstructured Data Format
VC	Virtual Channel
VP	Virtual Path

Chapter 2
AAL1gator II Driver Overview

ABOUT THIS CHAPTER

This chapter provides an overview of the AAL1gator II Driver.

OVERVIEW OF SOFTWARE FEATURES

The AAL1gator II Driver is a system-ready driver for the PM73121 AAL1gator II device from PMC-Sierra. The AAL1gator II Driver is written in ANSI "C" language and is portable to any Operating System (OS) environment. The AAL1gator II Driver offers a logical interface to the AAL1gator II device, and eliminates the need to know the device-specific functions and how to operate the device to achieve those functions. The AAL1gator II Driver significantly reduces the system integration time, since it has been tested with the AAL1gator II device. The AAL1gator II Driver performs diagnostics, sets up and clears AAL1 connections, monitors the device status, and accumulates statistics. A single AAL1gator II Driver can handle multiple AAL1gator II devices.

The following paragraphs detail some of the important AAL1gator II Driver functions.

Performing Diagnostics

At startup, the AAL1gator II Driver performs diagnostic tests on the AAL1gator II and the associated RAM. These tests detect interface errors between the CPU and the AAL1gator II. Specifically, the diagnostic software:

- Writes different patterns and reads them back to ensure the CPU-to-memory interface is functional.
- Writes patterns to detect RAM address aliasing errors.
- Writes values and reads them back from all device read/write registers to verify the CPU-to-AAL1gator II interface is functional.

Performing Initialization Functions

The AAL1gator II Driver performs the following intialization functions:

- Device initialization - Some registers should be preprogrammed, based on the hardware architecture of the target system.
- Self testing - Self testing includes testing on-chip RAM and associated external RAM.
- Interrupt Service Routine (ISR) installation - If the device supports an interrupt, the driver will supply an interrupt handler and a place to install it during initialization. The AAL1gator II Driver performs local processing for interrupts and generates events to the higher layer task.
- Custom event handlers installation - The driver provides hooks into which you can supply calls to custom event handlers to meet system requirements.
- Buffer allocation - Buffers requiring pre-allocation will be allocated at initialization time.
- Queue creation - Required queues are created.
- Task creation - If the driver's "sanity checks" are successful and the task exists for the driver, the driver will create the task and send an initial test message to the task. If the message is received by the task, it will log a message indicating successful startup. This process provides a checkpoint while you are porting the driver.

Configuring the Device with System Defaults

The AAL1gator II Driver configures the AAL1gator II with certain defaults. These defaults are in the porting section of the AAL1gator II Driver and can be easily changed. Examples of such default configuration data include:

- the line configuration
- the default size of the partially filled cell before sending the cell
- the default fill character
- the number of buffers associated for each queue
- the size of the Cell Delay Variation Tolerance (CDVT)
- the Virtual Paths (VPs) and the Virtual Channels (VCs) for each queue

After the diagnostic tests are complete, the AAL1gator II Driver configures the AAL1gator II and its data structures with those device values and prepares the device for real-time operation. Configures the system defaults for the AAL1gator II (for example,).

Providing APIs for Device Operation

The AAL1gator II Driver provides APIs to control the following AAL1gator II operations:

- Setting up and clearing connections (a connection can be a VP or a VC). In structured mode, the API also calculates the new structure size.
- Adding and removing $n \times 64$ channels into an ATM connection.
- Sending and receiving OAM cells, including a pacing function for transmitting OAM cells.
- Mapping T1/E1/T3 channels to ATM VCs. The AAL1gator II Driver offers API functions for the higher layer software to map and unmap lines (unstructured mode) and channels (structured mode) to ATM VCs.
- Activating and deactivating ATM channels. The AAL1gator II Driver offers API functions to activate and deactivate ATM channels, retaining the $n \times 64$ channel-to-ATM VC mapping (structured mode). The higher layer software can use this mapping capability to monitor the Channel Associated Signaling (CAS). In structured mode, the AAL1gator II Driver contains API functions to add or remove additional $n \times 64$ channels for an existing mapping.

Managing Events

While operating, the AAL1gator II generates an interrupt when it receives an OAM cell. The AAL1gator II Driver processes the interrupt locally and then generates an event for the higher layer processes. Upon receiving this event, the higher layer processes act on the OAM cell received. The AAL1gator II Driver also monitors the receive overruns and underruns on a connection, and offers APIs to read the number of overruns and underruns. The higher layer software can use the overrun and underrun counts to determine whether or not to adjust the Cell Delay Variation (CDV) configured for the connection setup during the connection definition time.

Collecting Statistics

The AAL1gator II Driver collects the statistics that are relevant to the AAL1gator II device operation and provides these statistics in the form of APIs for the higher layer software. Statistics collection can be performed in response to an interrupt from the device, or the device can be polled periodically. For statistical information, the driver accumulates select hardware register counters into larger counters. These larger counters do not need to be read as often as the corresponding hardware register counters.

From the AAL1gator II data structures, the AAL1gator II Driver retrieves statistics, such as the number of cells sent, the number of cells received, the conditioned data cells sent on a VC, and overruns and underruns on a connection.

The AAL1gator II Driver periodically monitors the status of errors (such as overruns and underruns), and accumulates the occurrence of those events. The AAL1gator II Driver uses the timer function in the OS Extensions (a well-defined set of OS wrapper functions) to run a periodic accumulation timer. The AAL1gator II Driver contains APIs for the higher layer software to read the values of those statistics.

HOW THE AAL1GATOR II DRIVER INTERACTS WITH APPLICATIONS

The AAL1gator II Driver interacts with the application layer in the following ways:

- The application invokes the driver via direct function calls (API functions). The driver then executes in the context of the application, and returns various data as specified by the API.
- Callback functions provided by the application to the AAL1gator II Driver DCT are to be installed into the AAl1 task (aal1port.c file) during porting. These functions will be used for event notifications by the DCT to the application.
- Event notifications by the DCT to the application.

Direct Function Calls

The direct function calls are the API functions that are part of the Device Driver Library (DDL) and are available to the application for a variety of actions. For example, the application uses direct function calls such as `Aal1ActivateChannel` (refer to "aal1ActivateChannel" on page 32) and `Aal1DeactivateChannel` (refer to "aal1DeactivateChannel" on page 34) for setting up and tearing down connections.

Callback Functions

These functions are provided by the application to the AAL1gator II Driver. When the DCT receives a signal from the ISR about the occurrence of a significant event, such as the reception of an OAM cell, it calls a function provided by the application as a callback routine to inform the application of this routine. Within this callback routine, the application can take whatever action the corresponding OAM cell requires.

Asynchronous Event Notification

Instead of using callback functions to notify the application layer of significant events, the DCT can also signal the application. This signaling can occur in the form of an event notification or sending a message in a queue or any other mechanism the user wants to employ. The signaling function used by the AAL1gator II Driver, `Aal1SigFn` (refer to "Step 4: Modify the aal1port.c File" on page 24), is a porting function and can be modified by the user to fit the system messaging scheme.

THE DRIVER STATES

The AAL1gator II Driver can be in one of four states. These states function as checkpoints to verify the proper initializations have occurred before calling the API functions, and to ensure the API functions are not called in the wrong order. Before the driver can enter the next state, a function must verify the driver is in the correct state. The function must then complete successfully so the driver can transition to the next state. The states are as follows:

- S0: Power-On Initialization State. The `Aal1PowerOnInit` function checks for the S0 state. If the S0 state exists and the function is successful, the state is changed to S1.
- S1: Power-On Self Test State. The `Aal1PowerOnSelfTest` function checks for the S1 state. If the S1 state exists and the function is successful, the state is changed to S2.
- S2: Final System Initialization State. The `Aal1FinalInit` function checks for the S2 state. If the S2 state exists and the function is successful, the state is changed to S3, which is the required state for most API functions.
- S3: Operational State.

In normal conditions, the transition sequence between the states is as follows: S0 followed by S1, followed by S2 (after permission from the management entity), followed by S3 (again, after permission from the management entity). The driver is usually in the S3 state. Most API functions, such as transmit and receive operations on the device, require the driver to be in the S3 operational state. As Figure 1 on page 8 shows, the initial software of a system should bring the driver to state S3 by making the following sequence of calls: `Aal1PowerOnInit()`; `Aal1PowerOnSelfTest()`; `Aal1FinalInit()`. This sequence of calls is equivalent to `open()` in a UNIX/DOS environment. The driver may exit this S3 state and enter any of the preceding states in case of abnormal conditions (such as system hang-up, and restart/reset) or during a system reconfiguration phase.

Figure 1 shows the driver states. The driver is assumed to be in the S0 state at power-up.

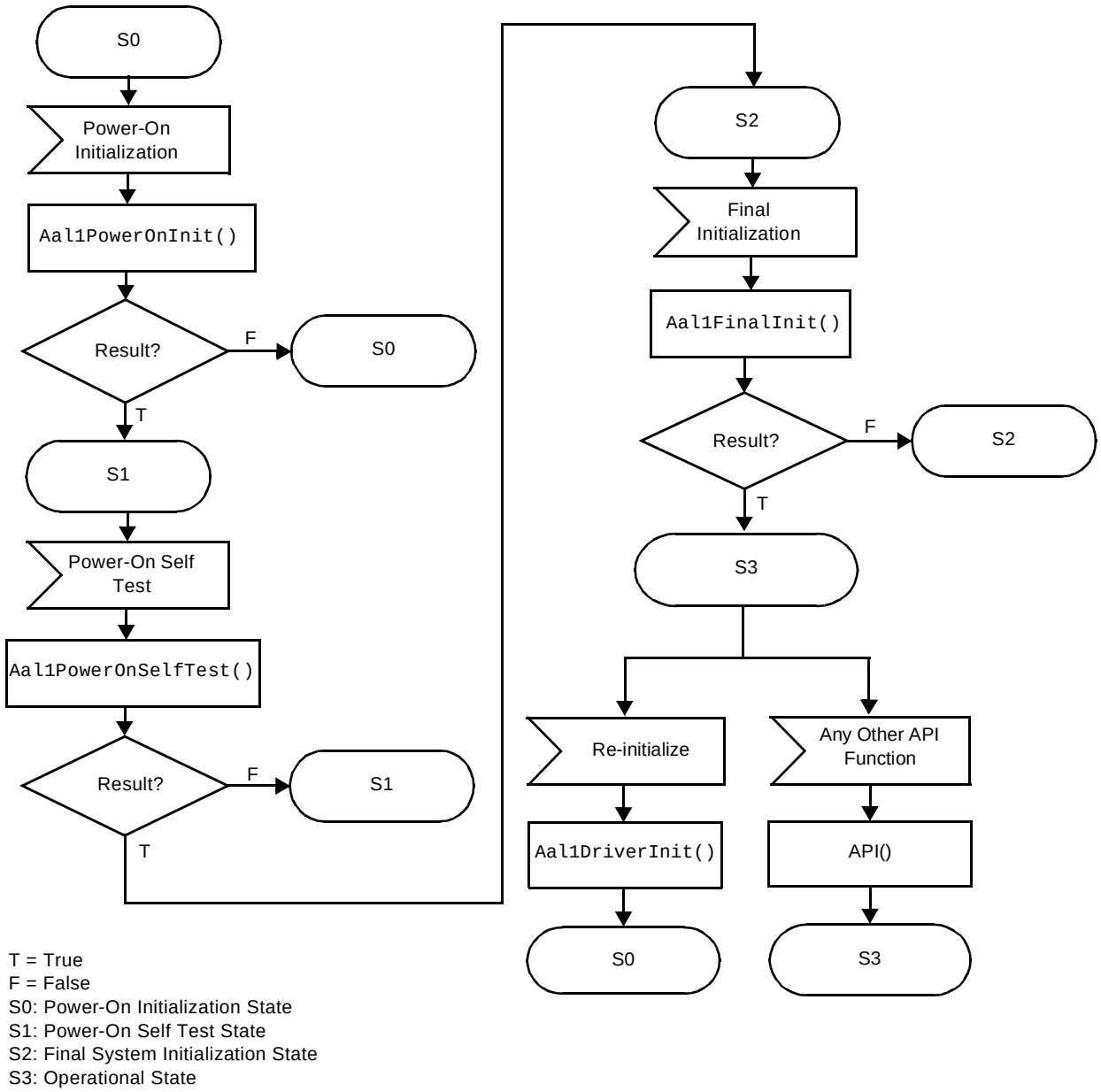

Figure 1. State Transition Diagram of an AAL1gator II Driver

The driver functions as a simple state machine. The events of interest are:

- Power-up.
- Operations the API function calls schedule for the device.
- Device interrupts.
- Timer events that schedule diagnostic checks and notify timeout conditions.

State S0 (Power-On Initialization State)

S0 is entered from the `Aal1DriverInit()` function. The driver (in S0) then proceeds to the `Aal1PowerOnInit()` function that checks for the S0 state. If the S0 state exists and the `Aal1PowerOnInit()` function is successful, the state is changed to S1.

NOTE: In the S0 state, the OS kernel services may not yet be available. Also, the interrupt system may not be fully initialized. Thus, the code executed in S0 should:

- Use only stack or scratch pad RAM for any local variables.
- Do not expect the interrupt services to be available. (To avoid problems with an uninitialized interrupt system, use status polling to recognize events of interest and disable interrupts from the device.)

`Aal1PowerOnInit()` performs these steps:

1. Determines if the results of the device's power-on diagnostics (if available) are acceptable. If not acceptable, `Aal1PowerOnInit()` exits with an error code.
2. Initializes the device to the configuration available in the ROM (Read Only Memory). Generally, you can change this new configuration by recompiling. This configuration should be similar to the expected configuration of the device when the system is operational. For example, if the device has a programmable clock rate, then initialize the device to the clock rate being used in the system.

NOTE: In this user's manual, "ROM" designates the data is contained in the initial load of the processor system. It may be in an electrical ROM, or may be a part of the boot-loaded code.

3. Changes the state to S1.

State S1 (Power-On Self Test State)

S1 is entered from the `Aal1PowerOnInit()` function. The driver (in S1) then proceeds to the `Aal1PowerOnSelfTest()` function that checks for the S1 state. If the S1 state exists and the `Aal1PowerOnSelfTest()` function is successful, the state is changed to S2.

NOTE: In the S1 state, the OS kernel services may not yet be available. Also, the interrupt system may not be fully initialized. Thus, code executed in S1 should:

- Use only stack or scratch pad RAM for any local variables.
- Do not expect the interrupt services to be available. (To avoid problems with an uninitialized interrupt system, use status polling to recognize events of interest and disable interrupts from the device.)

`Aal1PowerOnSelfTest()` performs these steps:

1. Temporarily changes the current chip configuration.
2. Prepares for the power-on self test.
3. Performs the power-on self test.
4. Stores the result.
5. If all tests pass, sets the state to S2. If all tests did not pass, sets the state to S1 and indicates the test(s) that failed.

State S2 (Final System Initialization State)

S2 is entered from the `Aal1PowerOnSelfTest()` function. The driver (in S2) then proceeds to the `Aal1FinalInit()` function that checks for the S2 state. If the S2 state exists and the `Aal1FinalInit()` function is successful, the state is changed to S3.

NOTE: In the S2 state, the OS kernel services are assumed to be available. Also, the interrupt system is assumed to be fully initialized. Thus, code executed in S2 can:

- Allocate memory.
- Create processes.
- Acquire timer handles.
- Expect the interrupt services to be available.

`Aal1FinalInit()` performs these steps:

PHASE A:

1. Allocates and initializes the DDB and the DCB for the driver.
2. Initializes the shadow registers from the values in the scratch pad RAM, and verifies the shadow registers against the values in the device registers.
3. Allocates the required memory buffer pools and timer handles.
4. Initializes the interrupt vectors. (This facility should be made available by the API of the driver for the interrupt subsystem.)
5. Enables device interrupts.

PHASE B:

6. Completes the remaining power-on diagnostic tests. These tests include the tests that examine the interrupt system, and those aspects of the device that can be effectively tested only in an environment defined by an OS kernel (for example, those requiring timer services).
7. Configures the device for the operational state (S3). (By this time, the final system configuration should be known to the management entities.) Optionally, this step can be skipped if it is known that the power-on initialization is also the initialization for the operational state.
8. If both Phase A and B pass successfully, exits to state S3; else enters the error state and exits.

State S3 (Operational State)

In the S3 state, the API is fully available. This is the operational state for the driver.

`Aal1DriverInit()` performs these steps:

1. Re-initializes all data structures.
2. Changes state to S0.

RE-ENTRANCY ISSUES

In a multitasking/multithreaded environment, such as exists in a real-time OS, the driver calls may be accidentally re-entered. The driver does not check to prevent such re-entries. In a re-entrance, the driver's data structures remain intact; however, there still may be unpredictable consequences for the application. For example, if two applications concurrently issue an API function call to set up the same Channel Control Block (CCB), the driver will return "OK" for both applications, but only one of the applications will actually set up the specified CCB. To avoid such "race" conditions, design the application so only DDO functions can be re-entered; non-DDO functions should not be re-entrant. Single supervisory tasks (those that are per driver) should be able to invoke non-DDO functions after notifying the other applications are using the DDOs.

Chapter 3
AAL1gator II Driver Architecture

ABOUT THIS CHAPTER

To help the application programmer design applications that use the AAL1gator II Driver, this chapter presents general information about how the driver operates. It does not provide the internal details and design of the driver.

AAL1GATOR II DRIVER DESIGN

The AAL1gator II Driver provides a high-level interface to the AAL1gator II device. Figure 2 shows the detailed architecture of the AAL1gator II Driver, including the main data structures of the driver. It also shows the interaction between the modules and their accesses to the data structures.

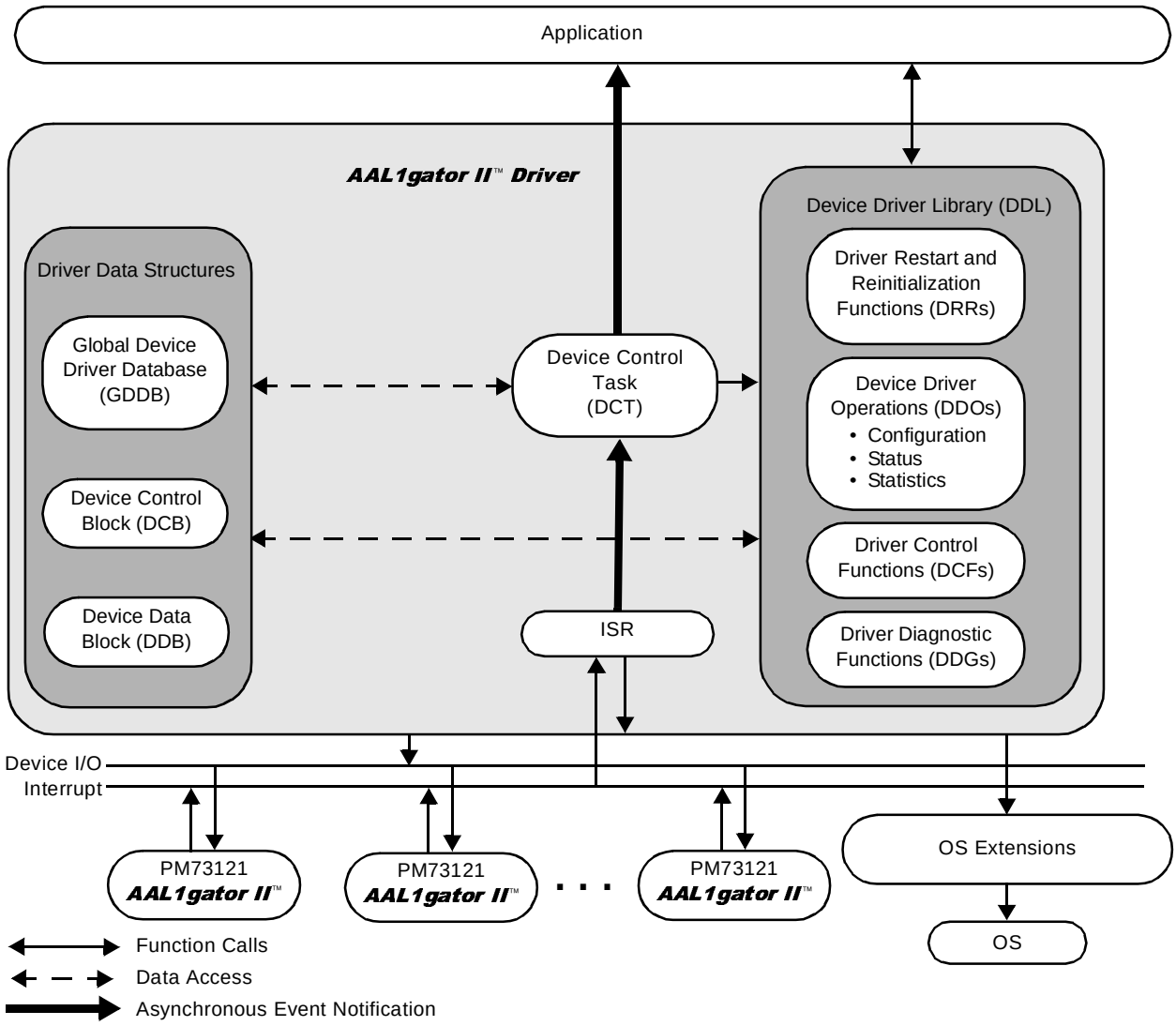

Figure 2. AAL1gator II Driver Architecture

The driver is divided into modules that perform different functions. These modules communicate with each other and access the AAL1gator II Driver data structure, as well as the AAL1gator II device's internal data structures to service the APIs called by the application.

The AAL1gator II Driver is divided into the following major functional blocks:

- The Device Driver Library (DDL)
- The Device Control Task (DCT)
- The Interrupt Service Routine (ISR)
- The driver data structures

The OS Extensions module is a software unit that is also provided with the AAL1gator II Driver.

For any number of AAL1gator II devices controlled by the driver, there is only one instance of the DDL and the DCT. The AAL1gator II devices are identified using an incremental number (starting at 0) called the *device_id*. All API functions require the *device_id* as an input. The driver may also expect a timer signal so it can schedule certain activites, such as updating counts and performing diagnostics.

The following sections explain the functional blocks of the AAL1gator II Driver.

The Device Driver Library (DDL)

The DDL provides the applications with the functions that execute in the context of the calling application and provide the driver services. The DDL can be divided into the following major sets of API functions:

- The Driver Restart and Reinitialization Functions (DRRs)
- The Device Driver Operations (DDOs)
- The Driver Control Functions (DCFs)
- The Driver Diagnostic Functions (DDGs)

The following sections describe each of these DLL API functions.

The Driver Restart and Reinitialization Functions (DRRs)

The DRR helps the application initialize the AAL1gator II Driver and the AAL1gator II devices. The functions in the DRR are explained in the following subsections.

Aal1PowerOnInit

The `Aal1PowerOnInit` function initializes the device to a default configuration, as specified in the Global Device Driver Database (GDDB), for that device.

NOTES:

- `Aal1PowerOnInit` does not require OS support.
- The GDDB must be properly initialized before executing `Aal1PowerOnInit`. The GDDB is initialized in the `Aal1DriverInit` function prior to calling `Aal1PowerOnInit`.

Aal1FinalInit

The `Aal1FinalInit` function allocates the system resources (the DCT, the DCB, the DDB, and the timer), and generates the complete environment for the driver API (the DDO) to be operational. `Aal1FinalInit` optionally initializes the device as specified by the initialization vector defined by the application. `Aal1FinalInit` assumes the availability of OS services.

The Driver Control Functions (DCFs)

`SetAal1Pid` is a DCF that allows the application to specify the process identifier (the *pid*) the driver will pass to `Aal1SigFn()` to signal events of interest. This allows communication between the AAL1gator II Driver ISR and the AAL1gator II Driver task.

The Device Driver Operations (DDOs)

The DDOs are the API functions used by the applications predominantly to exercise the functions for which the device exists. The API functions supported by the AAL1gator II Driver are described in Chapter 5, AAL1gator II Driver API, starting on page 28.

The Driver Diagnostic Functions (DDGs)

`Aal1PowerOnSelfTest` is a generic DDG function that performs an exhaustive test on the various functions of the device and returns the result of the tests.

The Device Control Task (DCT)

The AAL1gator II Driver contains a DCT. For a single device type, there is one driver and one DCT. For example, for three PM73121 devices, there is one driver with one DCT.
The DCT performs the following functions:

- Maintains the updated counts in the DDB corresponding to the counts maintained by a device.
- Receives the signals from the ISR and takes appropriate action on these signals.
- Schedules "sanity" checks on the device and the driver databases.
- Signals events of interest to a predefined application task.

The ISR signals events to the DCT with the `Aal1SigFn()` function. This function uses OS Extensions queue functions to send messages to the DCT. If queue send operations are not permitted inside the ISR, then an alternate OS-specific solution may be needed.
Timer events are scheduled in the DCT to provide periodic sanity checks and database updating. These timer events are accomplished with OS Extensions timer calls.

The DCT may signal significant events to a predefined application task. Since it is impossible to know what application is used, the application signaling is left as a porting issue. A suggestion is to use OS Extensions calls for this reporting since OS Extensions is already used by the DCT.

The Interrupt Service Routine (ISR)

The ISR performs the following functions:

- Traps hardware interrupts generated by the device.
- Acknowledges the interrupt source so the device deactivates the hardware interrupt signal.
- Signals the DCT of the interrupts.
- Updates the DCB and DDB.

The Driver Data Structures

The following are the types of data structures associated with the driver:

- The Global Device Driver Database (GDDB)
- The Device Control Block (DCB)
- The Device Data Block (DDB)
- The AAL1 Setup Parameter Structure
- The AAL1 Enhanced Setup Parameter Structure

Figure 3 shows the relationship between the GDDB, the DCB, and the DDB.

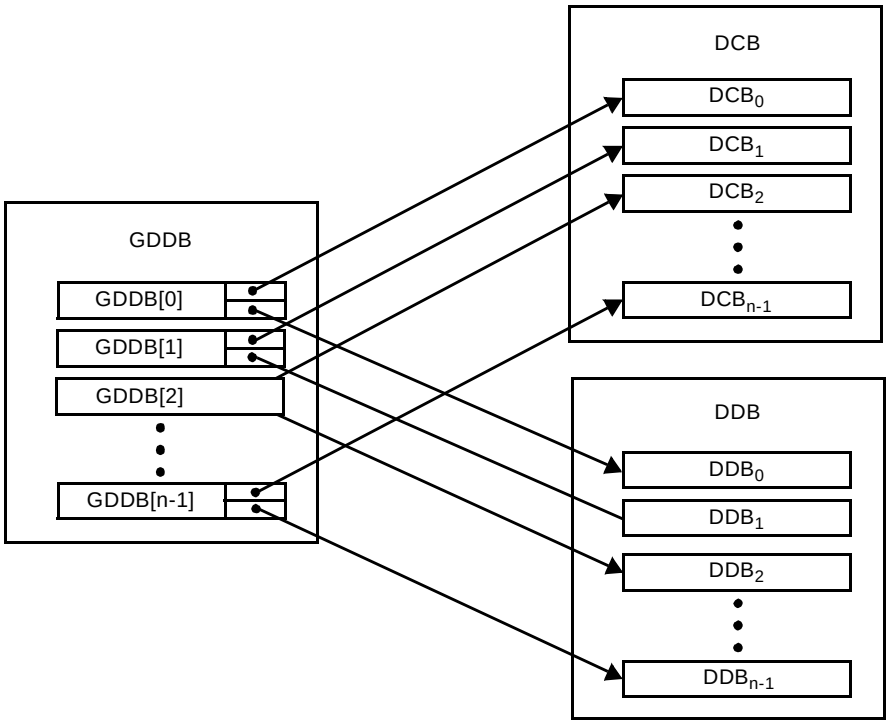

Figure 3. Relationship Between the GDDB, the DCB, and the DDB

The following sections explain the global data structures.

Global Device Driver Database (GDDB)

Description: An array of structures, in which each structure contains the following information about each device the driver controls and monitors:

- The physical address of the device.
- The current configuration details of the driver and the device (shadow registers).
- The data structures for internal driver use.

Data Type: Aal1GDDB (struct). This data type is declared in the aal1str.h file.

Identifier: `gddb[AAL1_MAX_DEVICES]`. This array is declared in the aal1ini.c file.

Table 3 lists and describes the data types in the GDDB.

Table 3. Global Device Driver Database (GDDB)

Data Type	Field Name	Description
char	sDeviceName[AAL1_NAME_LENGTH]	Indicates the device name.
UINT2	*pu2HardwareBaseAddress	Indicates the hardware base address of the AAL1gator II.
UINT2	*pu2MemoryAddress	Indicates the memory start address of the AAL1gator II.
UINT4	u4InterruptVector	Indicates the hardware interrupt vector.
tAal1InitVector	initialCfg	Indicates the device initial configuration.
tAal1DCB	*ptDcb	Points to the Device Control Block (DCB).
tAal1DCB	*ptDdb	Points to the Device Data Block (DDB).
UINT4	u4DriverState	Indicates the driver state (S0, S1, S2, or S3).
UINT4	u4ChipVersion	Indicates the version of the device.

Device Control Block (DCB)

Description: For each device being controlled by the driver, there is a unique DCB that contains the following control information about the device:

- The backup configuration of the device for recovery and diagnostics purposes.
- The driver control information; for example, the *pid* to be passed to `Aal1SigFn()` and the DCT's *pid*.

Data Type: DCB (struct). This data type is declared in the aal1str.h file.

Identifier: `Aal1GDDB[`*device_id*`].ptDcb`. This identifier is dynamically allocated.

Table 4 lists and describes the data types in the DCB.

Table 4. Device Control Block (DCB)

Data Type	Field Name	Description
UINT4	u4FillChar	Indicates the fill character for partially filled cells.
UINT4	u4DCTTaskId	Indicates the DCT task ID.
UINT4	u4TimerId	Indicates the timer ID for statistics collection.
UINT4	u4nVCs	Indicates the number of allocated VCs.
tAal1InitVector	finalCfg	Indicates the device's final configuration.
taal1MapStr	chQuMap[AAL1_MAX_QUEUE]	Indicates the channel-to-AAL1gator II queue mapping.

Device Data Block (DDB)

Description: Stores statistics for individual devices being monitored by the driver. There is one DDB for each device being monitored.

Data Type: DDB (struct). The data type is declared in the aal1str.h file.

Identifier: `Aal1Gddb[`*device_id*`].ptDdb`. This identifier is dynamically allocated.

Table 5 lists and describes the data types in the DDB.

Table 5. Data Types in the Device Data Block (DDB)

Data Type	Field Name	Description
UINT4	u4RecvUnderrun[AAL1_MAX_QUEUE]	Receive underruns.
UINT4	u4RecvOverrun[AAL1_MAX_QUEUE]	Receive overruns.
UINT4	u4PtrMismatch[AAL1_MAX_QUEUE]	Receive pointer mismatch realignments.
UINT4	u4InvalidSN[AAL1_MAX_QUEUE]	Invalid sequence number.
UINT4	u4FrcdUnderrun[AAL1_MAX_QUEUE]	Forced underrun.
UINT4	u4PtrSearch[AAL1_MAX_QUEUE]	Pointer search.
UINT4	u4BlnkAllocTbl[AAL1_MAX_QUEUE]	Blank allocation table.
UINT4	u4CellRecvSt[AAL1_MAX_QUEUE]	Cell received status.
UINT4	u4CellLost[AAL1_MAX_QUEUE]	Cell lost.
UINT4	u4TCellCount[AAL1_MAX_QUEUE]	Transmit cell count.
UINT4	u4RCellCount[AAL1_MAX_QUEUE]	Receive cell count.
UINT4	u4RDroppedCellCount [AAL1_MAX_QUEUE]	Receive dropped cell count.
UINT4	u4RIncorrectSnp[AAL1_MAX_QUEUE]	Receive incorrect Sequence Number Protection (SNP) count.
UINT4	U4RLostCellCount [AAL1_MAX_QUEUE]	Receive lost or replaced cell count.
UINT4	u4RMisinserted [AAL1_MAX_QUEUE]	Receive misinserted cells
UINT4	u4PtrParityErrs [AAL1_MAX_QUEUE]	Receive pointer parity errors
UINT4	u4RPtrReframeCount [AAL1_MAX_QUEUE]	Receive reframes count.
UINT4	u4TSuppressedCellCnt[AAL1_MAX_QUEUE]	Transmit suppressed cell count.
UINT4	u4TCondCellCount[AAL1_MAX_QUEUE]	Transmit conditioning cell count.

AAL1 Setup Parameter Structure

Description: Used to configure standard connections.

Data Type: taal1TxRxParam (struct). The data type is declared in the aal1str.h file.

Table 6 lists and describes the data types in the AAL1 setup parameter structure used by the Aal1ActivateChannel function (refer to "aal1ActivateChannel" on page 32) and the Aal1EnhancedActivateChannel function (refer to "aal1EnhancedActivateChannel" on page 33).

Table 6. Data Types in the AAL1 Setup Parameter Structure

Data Type	Field Name	Description
UINT4	u4Signalling	Signalling enable or disable.
UINT4	u4CheckParity	Parity enable or disable.
UINT2	u2TxVp	Transmit VP.
UINT2	u2TxVc	Transmit VC.
UINT2	u2RxVp	Receive VP.
UINT2	u2RxVc	Receive VC.

AAL1 Enhanced Setup Parameter Structure

Description: Used to configure enhanced connections.

Data Type: taal1EnhancedParam (struct). The data type is declared in the aal1str.h file.

Table 7 lists and describes the data types in the AAL1 enhanced setup parameter structure used by the Aal1EnhancedActiveChannel function (refer to "aal1EnhancedActivateChannel" on page 33).

Table 7. Data Types in the AAL1 Enhanced Setup Parameter Structure

Data Type	Field Name	Description
UINT2	u2MaxBuf	Defines the maximum buffer depth in cells.
UINT2	u2CDVT	Defines the Cell Delay Variation Time (CDVT).
UINT2	u2DisableSN	Disables Sequence Number (SN) processing.
UINT2	u2PartialCell	Defines the partial cell size in bytes.
UINT2	u2SNConfig	Defines the SN configuration. 1 Drop the first cell. 0 If the SNP is correct, recieve the first cell.

OS Extensions

The PMC OS Extensions module is an operating system wrapper that is designed to provide a consistent interface to the underlying OS. The PMC OS Extensions module provides the following functionalities:

- Message queues
- Periodic timers
- Event notification
- Task management
- Memory management
- Debug logging

The PMC OS Extensions module separates the RTOS porting into a separate module (see Figure 4). The porting section calls the PMC OS Extensions module interface to perform RTOS actions. The only modifications required in the porting section are for device I/O and system configuration.

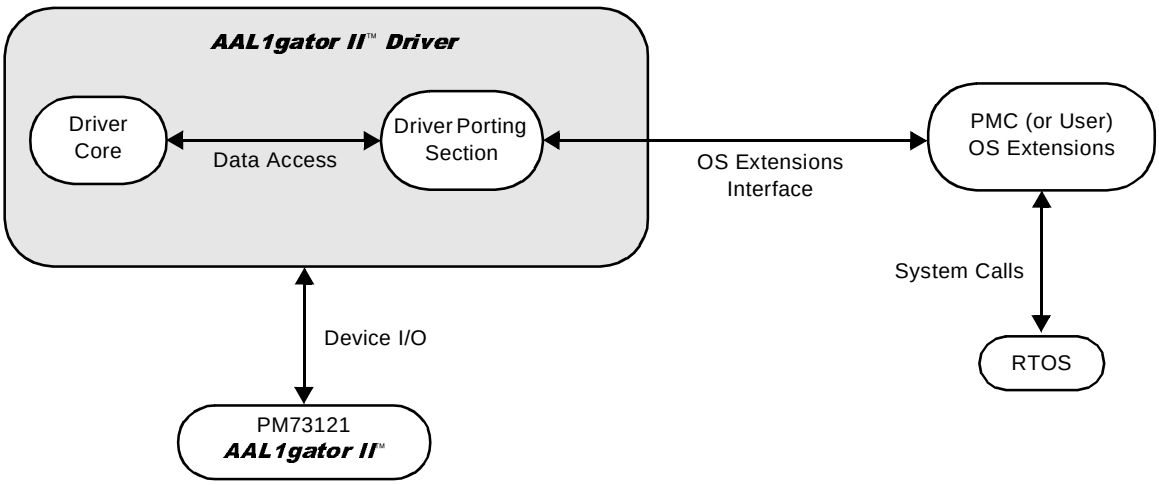

Figure 4. Model of an AAL1gator II Device and its Associated AAL1gator II Driver

Using the PMC OS Extensions module with the AAL1gator II Driver is optional. Customers may replace the PMC OS Extensions functions with their own OS-specific wrappers.

Chapter 4
Porting Guidelines

ABOUT THIS CHAPTER

This chapter describes the steps to port the AAL1gator II Driver to a specific hardware and OS environment. These steps include developing additional code and defining the various macros and preprocessor constants used by the driver code. For easy porting, the changes required for the driver are grouped into two files: the Aal1port.h file, and the Aal1port.c file. Global changes required by the AAL1gator Drivers are in two separate files: the gport.h file and the gport.c file.

In addition to the driver porting files, OS extensions must be ported for the target OS. PMC-Sierra provides OS Extensions, which is a wrapper that provides a consistent interface to the OS and is used in the AAL1gator II Driver (for more information on the OS Extensions, refer to "OS Extensions" on page 19). Since the AAL1gator II Driver uses only a subset of the functions available in OS Extensions, porting only those functions is sufficient. Refer to "Step 3: Port OS Extensions" starting on page 22 for information on porting OS Extension.

NOTE: PMC-Sierra recommends you do *not* modify the core files during porting.

HOW THE SOURCE CODE IS ORGANIZED

The code for the AAL1gator II Driver is organized into the C language files listed below.

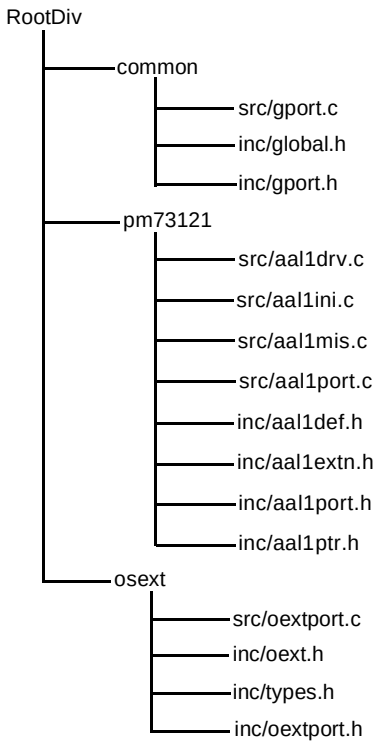

Figure 5. AAL1gator II Driver Source Files

The following header files are necessary for the AAL1gator II Driver API functions to operate:

global.h
: This file contains general declarations for the AAL1gator II Driver. Include this file before any other files pertaining to the driver are included.

gport.h
: This file contains general porting information for the AAL1gator II Driver..

aal1def.h
: This file contains pre-processor constants for: error code values for the AAL1gator II Driver API, registers numbers in the AAL1gator II device, and default initialization values for the device.

aal1extn.h
: This file contains ANSI function prototypes for each function in the driver's API.

aal1port.h
: This file contains porting macros and other porting information.

aal1str.h
: This file contains data structure declarations.

oextport.h
: This file contains general OS porting information.

PORTING STEPS

To port the AAL1gator II Driver to a specific environment, you will complete the steps in this section.

Step 1: Modify the gport.c File

Verify the implementations included in this file function correctly on the target system. By default, `EnablePreemption()` and `DisablePreemption()` use the PMC-Sierra OS Extensions to control task preemption. This requires the actual porting to be done in OS Extensions.

Step 2: Modify the gport.h File

The gport.h file should not require porting, since it references OS Extensions for all OS-dependent operations. If required, task-specific constants for the drivers can be changed here. For most installations, this file will need no modifications.

Step 3: Port OS Extensions

The PMC-Sierra OS Extensions module encapsulates all OS-specific operations required by the AAL1gator Driver . The PMC-Sierra OS Extensions provides operations for managing tasks, queues, timers, events, semaphores, memory, and debug logging. Queues, tasks, and semaphores are "named" in OS Extensions. A character string is assigned to each queue or task to uniquely identify it. For more information on the PMC-Sierra OS Extensions module, refer to "OS Extensions" on page 19.

You can port either the PMC-Sierra OS Extensions module or your own OS extensions. The following substeps (3a, 3b, and 3c) describe what changes are required in each file if you choose to use the PMC-Sierra OS Extensions. If you choose to port your own OS extensions, you can skip step 3b.

Step 3a: Modify the types.h File

The types.h file defines all the system- and compiler-specific type definitions required by OS Extensions. The types are identified by the number after the type. 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.

Step 3b: Modify the oextport.h File

This step is required only if you are using the PMC-Sierra OS Extensions module.

The oextport.h file contains the preprocessor constants used by each different class of OS Extensions calls. Most values defined in this file do not need to be changed, unless doing so eases the porting of the oextport.c file. For example, the constant EV_COND_OR is used by `qx_receive()` to indicate the wait condition is a logical OR of the event flags. The value of the EV_COND_OR constant in oextport.h can be changed to represent the actual value of this flag in the underlying OS call used in porting `qx_receive()`. This allows more efficient operation, since the value of EV_COND_OR does not need to be translated to the value used by the actual OS call. Placing these constants in this file gives the developer more flexibility in porting OS Extensions.

NOTE: The name of the constants in this file should *not* be changed since doing so would alter the interface to OS Extensions.

Step 3c: Code the oextport.c File

The oextport.c file contains the code that implements the actual OS Extensions function calls. Most of the code in this file will have to be ported specifically for the target OS. The AAL1gator II Driver uses a subset of the function calls defined in the oextport.c and oextport.h files. If porting the AAL1gator II Driver, then the implementation of the unused function calls can be removed to shorten the work required in porting. The oextport.c file contains comments identifying which function calls are used by the AAL1gator II Driver. The following list summarizes the function calls required by the AAL1gator II Driver and gives a brief description of each function call.

NOTE: If you choose to port your own OS extensions (that is, you are not using the PMC-Sierra OS Extensions module), substitute code that implements your OS extension calls.

- Events
  - `evx_send` - Send a set of events to a task.
  - `evx_receive` - Wait for a set of events governed by a logical operation and a timeout.
- Memory
  - `mx_create` - Allocate a memory block of a given size.
  - `mx_delete` - Free a memory block allocated with `mx_create`.
  - `mx_set_value` - Set a constant byte value in a memory region.
- Queues
  - `qx_create` - Create a named queue with a given size, and optionally specify an event that will be sent to a task when a message arrives in the queue.
  - `qx_get_buffer` - Allocate a message to send with `qx_send`.
  - `qx_ident` - Return the queue identifier associated with a named queue.
  - `qx_receive` - Receive a message sent to a queue with an optional timeout period.
  - `qx_return_buffer` - Free a message after receiving it with `qx_receive`.
  - `qx_send` - Send a message to a queue, and optionally specify a named queue for the response.
- Timers
  - `tmx_evevery` - Send a set of events to a task at the specified interval.
  - `tmx_wkafter` - Put the task to sleep for the specified time period.
- Tasks
  - `tx_mode` - Set the current mode of the task.
  - `tx_start` - Create and start a new task with the given stack size, priority, arguments, and starting point.
- Debug Logging
  - `x_trace` - Send a message to the debug log with a key and debug level.

IMPORTANT! The interface to the PMC OS Extensions module must not change during porting. Do *not* modify the function names, their parameters, or the constant names used by the functions while porting. Doing so will make it more difficult to use future versions of the AAL1gator II Driver. If a parameter or constant does not make sense for the target system, then leave it as defined and ignore it.

Step 4: Modify the aal1port.c File

Determine the desired configuration of the `Aal1DriverInit()` function within the driver code, and change the function accordingly. The `Aal1DriverInit()` function should basically initialize the GDDB entries for all the devices. Some typical entries to be initialized are:

- The physical base address of the device.
- The default parameter values with which `Aal1PowerOnInit()` will initialize the devices.
- The interrupt vector number to be used by the device (if any) so the ISR defined by the driver is executed as a part of the interrupt servicing.
- The configuration parameters of the driver software.

For more information, refer to "Step 6: Assign Proper Values to the Constants in aal1port.h" on page 24. Chapter 3, "AAL1gator II Driver Architecture", starting on page 12 defines the `Aal1DriverInit()` function.

The `Aal1StartTimer()` function sends periodic events to the DCT to allow the DCT to perform time-dependent operations. The implementation of `Aal1StartTimer()` uses OS Extensions' `tmx_evevery()` function to send the periodic events to the DCT. Because it uses OS Extensions, the `Aal1StartTimer()` function should not require modification during porting.

The `Aal1SigFn()` function is used to inform the DCT, if present, of events. The implementation of `Aal1SigFn()` uses OS Extensions' `qx_get_buffer()` and `qx_send()` functions to send messages. The `Aal1SigFn()` function is designed to be called from an ISR external to the driver code. In certain OS environments, the functions that can be called from within an ISR can be very limited. In such cases, it may be useful to pass all the signals to a dedicated task that will then signal the applications in an appropriate manner.

Step 5: Assign Proper Values to the Device Specification Constants in the aal1port.h File

Modify the values of the constants relating to the system-level hardware configuration. For example, change the `DEV_BASE_ADDRESS` constant to point to the proper absolute value memory address location for the device's I/O ports. If memory-mapped I/O is not used, modify the `IN` and `OUT` macros accordingly.

Step 6: Assign Proper Values to the Constants in aal1port.h

Following are the constants defined in the aal1port.h file. Define them as specified by the hardware environment.

`#define AAL1_MAX_DEVICES`

The number of AAL1gator devices present in the hardware module.
Default: `1`

`#define AAL1_CHIP_VERSION`

The revision number of the AAL1gator device used in the hardware module.
Default: `121A`

`#define AAL1_INIT_VECTOR`

The address of the hardware interrupt vector of the device. Change the default value in the aal1port.h file to the system-specific value.
Default: `1`

Step 7: Define the Pre-processor Macros

Define the following macros in the Makefile. The following macro definitions will be affected by the endianness of the hardware platform and the AAL1gator device type (for example, Application-Specific Integrated Circuit (ASIC) or Electronically Programmable Logic Device (EPLD)).

`BIG_ENDIAN`
If the platform is a big endian platform, define this `BIG_ENDIAN` constant. If the platform is a little endian platform, leave this macro undefined and the code will default to little endian.

Step 8: Replace Function Stubs in aal1port.c with Suitable Code

Following are the function stubs for hardware and OS/system initialization-specific functions defined in the aal1port.c file. Replace these stubs with appropriate code.

Function Name: `aal1DriverInit()`

Purpose: To initialize the GDDB contents to proper values so power-on initialization can be performed on the device, and so the device can configure itself.

To initialize the GDDB in the aal1port.c file, follow these steps:

a. Define the GDDB parameters for your system. Table 8 lists the GDDB parameters you should define. Refer to the aal1srt.h file for the exact structure.

Table 8. GDDB Parameters to Define Appropriately for Your System

Field	Description	How to Define
`aal1GDDB[n].DeviceName`	Indicates the device name (for example, `aal1`).	`aal1`
`aal1GDDB[n].pu2HardwareBaseAddress`	Indicates the hardware base address.	Define appropriately for your system.
`aal1GDDB[n].pu2MemoryAddress`	Indicates the memory start address of the device.	Define appropriately for your system.
`aal1GDDB[n].pu4InterruptVector`	Indicates the hardware interrupt vector.	Define appropriately for your system.
`aal1GDDB[n].u4ChipVersion`	Indicates the device revision number.	Define appropriately for your system.

b. Initialize the additional GDDB parameters in Table 9 to `NULL`.

Table 9. GDDB Parameters to Initialize to NULL

Field	Description	How to Define
`aal1GDDB[n].ptDcb`	Points to the DCB.	Initialize to `NULL`.
`aal1GDDB[n].ptDdb`	Points to the DDB.	Initialize to `NULL`.
`aal1GDDB[n].u4DeviceState`	Indicates state 0.	Initialize to `NULL`.

c. Initialize the following GDDB device register parameters according to the functional needs of the device on the hardware module.

Field: `aal1GDDB[n].initialCfg.u4CompLneReg`
Description: The values the driver assigns to the COMP_LIN_REG registers.
Choices: Refer to the COMP_LIN_REG register description in the *AAL1gator II Long Form Data Sheet*.

Field: `aal1GDDB[n].initialCfg.u4LineMode[8]`
Description: The values the driver assigns to the LIN_STR_MODE registers.
Choices: Refer to the LIN_STR_MODE register description in the *AAL1gator II Long Form Data Sheet*.

Function Name: `aal1ResetChip`

Purpose: To reset the AAL1gator II device pointed to by `u4DeviceId`.

Function Name: `aal1WriteCommand`

Purpose: To write to a microprocessor command register in the device. The value of the command registers and the device ID are passed as parameters to this function.

Function Name: `aal1ReadCommand`

Purpose: To read microprocessor command registers from the device. This function returns the value of the command register. The device ID is passed as a parameter to this function.

Function Name: `aal1InitInterruptVector`

Purpose: To initialize the hardware interrupt vector for the device. The address of the interrupt vector and the device ID are passed as parameters to this function. This function is required only for ASIC chips.

Step 9: Define Types in the aal1port.h File

Following are the types defined in the aal1port.h file. Define them appropriately for the compiler used to build the driver.

`UINT1`
Unsigned integer; one byte in length. Typically an unsigned character.

`UINT2`
Unsigned integer; two bytes in length.

`UINT4`
Unsigned integer; four bytes in length.

Step 10: Code and Install the Interrupt Handler

The ISR is expected to be executed when the device interrupts the processor. Program the operating system or the CPU so the device ISR is executed once for every interrupt caused by the device. Typically, you can do this by coding an interrupt handler that appropriately handles the interrupt hardware of the CPU (for example, code an ISR that issues acknowledgments to the interrupt controller hardware and masks lower priority interrupts), and then calls the ISR provided by the driver. Then, at the end of the `Aal1DriverInit()` function, install this interrupt handler so it executes when an interrupt for this device occurs.

Step 11: Compile and Link the Source Code Files into Library/Object Modules

Generate the driver libraries by compiling and linking the `Aal1SigFn()` and the `Aal1DriverInit()` functions with the other driver code modules. Follow the conventions required by the target OS to generate the driver. For example, apply the proper compile and link switches. Then define additional modules as required by the target OS's driver-interfacing conventions.

Chapter 5
AAL1gator II Driver API

ABOUT THIS CHAPTER

This chapter is organized as follows:

- The first part of this chapter describes the abilities of the AAL1gator II Driver and the primary driver API functions you need during normal device operation. The API functions listed in this section allow you to quickly and easily begin using the capabilities the driver offers. The AAL1gator II Driver manages the following device functions:
    - Initialization. For related API functions, refer to "Initializing All Internal Device Registers" on page 29.
    - Configuration. For related API functions, refer to "Mapping a Line or Channels in a Line to an ATM VP/VC" on page 29.
    - Processing Operations, Administration and Maintenance (OAM) cells. For related API functions, refer to "Processing OAM Cells" on page 38.
    - Controlling the Synchronous Residual Time Stamp (SRTS). For related API functions, refer to "Controlling the Synchronous Residual Time Stamp (SRTS)" on page 39.
- The second part of this chapter ("Statistics API Functions" starting on page 40) lists the statistics API functions you may need to read and monitor statistics within the device for network management purposes. Specifically, the API functions in this section are for the following purposes:
    - "Obtaining Counts of Out-of-Sequence Cells or Cells with Uncorrectable SN CRCs" starting on page 40.
    - "Obtaining Counts of Receive Underruns or Overruns" starting on page 42.
    - "Obtaining Counts of AAL1 Cells Transmitted or Received" starting on page 43.
    - "Obtaining Counts of Pointer Mismatches or Pointer Reframes" starting on page 44.
    - "Obtaining Counts of Lost or Replaced AAL1 Cells" starting on page 45.
    - "Obtaining Counts of Dropped or Misinserted AAL1 Cells" starting on page 46.
- The last part of this chapter lists the API functions you need to perform additional configuration tasks, such as obtaining the maximum buffer depth for a given queue and setting the maximum buffer depth for a given queue. For these additional configuration functions, refer to "Obtaining or Setting the Maximum Buffer Depth for a Queue" starting on page 47.

NOTE: Before any API function can be used, the driver must be initialized. The `aal1DriverInit()` function in the aal1port.c file initializes the driver with default parameters. Refer to "Step 8: Replace Function Stubs in aal1port.c with Suitable Code" on page 25.

PRIMARY DRIVER API FUNCTIONS TO PERFORM AAL1GATOR FUNCTIONS

Initializing All Internal Device Registers

The initialization code in `aal1DriverInit()` initializes the device with the parameters defined in aal1port.c. The initialization function also runs a self-test on the device to ensure the interface between the CPU and the AAL1gator II device(s) is functional.

Please refer to "Step 8: Replace Function Stubs in aal1port.c with Suitable Code" on page 25 for more information about initializing the GDDB contents to proper values so power-on initialization can be performed.

Mapping a Line or Channels in a Line to an ATM VP/VC

Figure 6 shows a high-level view of the relationships among the AAL1gator II device, the AAL1gator II Driver, other ATM devices, and an ATM network.

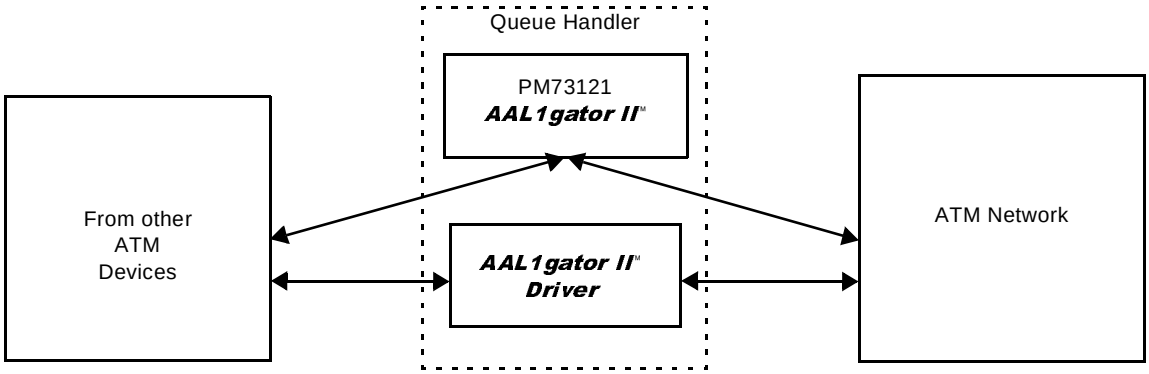

Figure 6. Relationships Among the AAL1gator II Driver, other ATM Devices, and an ATM Network

The application calls `aal1ActivateLine` or `aal1ActivateChannel` to define a mapping. One of the following mapping functions returns a handle that can be used for further operations on this mapping.

- `aal1getQhandle()`
- `aal1ActivateLine()`
- `aal1DeactivateLine()`
- `aal1ActivateChannel()`
- `aal1DeactivateChannel()`
- `aal1AssociateChannel()`
- `aal1DisassociateChannel()`
- `aal1EnableTxCond()`
- `aal1DisableTxCond()`
- `aal1EnableRxCond()`
- `aal1DisableRxCond()`

NOTE: When the configuration of an existing mapping is changed, the traffic is affected. For example, when `aal1AssociateChannel` is called to add more channels to an existing mapping, it affects the traffic on the existing channels.

aal1GetQhandle

Description: Returns the queue handle for an active line or channel. Applications such as the Simple Network Management Protocol (SNMP) agent can use this API function to obtain the queue handle for a group of channels and use the handle for status/statistics functions.

Invocation: `int aal1GetQhandle (u4DeviceId, UINT4 u4LineNo, UINT4 u4Channels)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`u4LineNo`	Specifies which line number to configure. Valid values are 0 to (AAL1_MAX_LINES -1).
`u4Channels`	Specifies the channel map.

Outputs: None.

Returns: Queue handle
`AAL1_BAD_POINTER`
`AAL1_NO_TX_QUEUES`

aal1ActivateLine

Description: Activates a T1 or an E1 line of the device in High-Speed Unstructured Data Format (UDF-HS) mode. Returns a queue handle that will be used for future operations on the line.

Invocation: `int aal1ActivateLine (UINT4 u4DeviceId, UINT4 u4LineNo, tAal1TxRxParam *ptParam)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`u4LineNo`	Specifies which line number to configure. Valid values are 0 to (AAL1_MAX_LINES -1).
`ptParam`	Points to the parameters needed for the line configuration.

Outputs: None.

Returns:
Queue handle
`ERROR`
`AAL1_BAD_DEV_NO`
`AAL1_BAD_POINTER`
`AAL1_INVALID_PARAM`
`AAL1_MAX_VCS_ERR`
`AAL1_NO_RX_QUEUES`
`AAL1_NO_TX_QUEUES`

aal1DeactivateLine

Description: Deactivates the line that is in use, and frees the queue handle.

Invocation: `int aal1DeactivateLine(UINT4 u4DeviceId, int qHandle)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`qHandle`	Specifies the queue handle for the line.

Outputs: None.

Returns:
`SUCCESS`
`ERROR`
`AAL1_BAD_DEV_NO`
`AAL1_BAD_POINTER`
`AAL1_INVALID_PARAM`
`AAL1_INVALID_QUEUE`

aal1ActivateChannel

Description: Maps the channel(s) of a T1 or an E1 line to a VP/VC. Enables full duplex mode after configuring the mapping. Initializes transmit and receive, conditioned signaling and data values to 0, and disables the conditioning. Initializes statistics counters to 0. Returns a queue handle that will be used for future operations on the mapping.

Invocation:
```
int aal1ActivateChannel (UINT4 u4DeviceId, UINT4 u4LineNo,
UINT4 u4Channels, tAal1TxRxParam *ptParam)
```

Inputs:

Argument	*Description*
u4DeviceId	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
u4LineNo	Specifies which line number to configure. Valid values are 0 to (AAL1_MAX_LINES -1).
u4Channels	Identifies the bit map of the channel(s). The Least Significant Bit (LSB) is channel number 0.
ptParam	Points to the parameters needed to configure the channel(s).

Outputs: None.

Returns:
Queue handle
ERROR
AAL1_BAD_DEV_NO
AAL1_BAD_POINTER
AAL1_INVALID_PARAM
AAL1_LINENO_VC_MISMATCH
AAL1_MAX_VCS_ERR
AAL1_NO_QUEUES
AAL1_NO_RX_QUEUES
AAL1_NO_TX_QUEUES

aal1EnhancedActivateChannel

Description: Maps the channel(s) of a T1 or an E1 line to a VP/VC. Enables full duplex mode after configuring the mapping. Initializes transmit and receive, conditioned signalling and data values to 0, and disables the conditioning. Initializes statistics counters to 0. Returns a queue handle that will be used for future operations on the mapping. In addition to the abilities of the aal1ActivateChannel function (refer to "aal1ActivateChannel" on page 32), this aal1EnhancedActivateChannel function allows the user, at connection setup, to configure the maximum buffer depth, define the CDVT, disable the SN processing, define the partial cell size, and define the SN configuration (refer to Table 7 on page 18). When `ptEnhanced` is equal to `NULL`, this function operates the same as the aal1ActivateChannel .

Invocation:
```
int aal1EnhancedActivateChannel(UINT4 u4DeviceId, UINT4 u4LineNo,
UINT4 u4Channels, tAal1TxRxParam *ptParam,
taal1EnhancedParam *ptEnhanced)
```

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`u4LineNo`	Specifies which line number to configure. Valid values are 0 to (AAL1_MAX_LINES -1).
`u4Channels`	Identifies the bit map of the channel(s). The Least Significant Bit (LSB) is channel number 0.
`ptParam`	Points to the parameters needed to configure the channel(s).
`ptEnhanced`	Points to the enhanced parameters used to configure the channel(s).

Outputs: None.

Returns: Queue handle
`ERROR`
`AAL1_BAD_DEV_NO`
`AAL1_BAD_POINTER`
`AAL1_INVALID_PARAM`
`AAL1_LINENO_VC_MISMATCH`
`AAL1_MAX_VCS_ERR`
`AAL1_NO_QUEUES`
`AAL1_NO_RX_QUEUES`
`AAL1_NO_TX_QUEUES`

aal1DeactivateChannel

Description: Deactivates the channel(s) on a line that is(are) in use. Frees the queue handle.

Invocation: `int aal1DectivateChannel(UINT4 u4DeviceId, int qHandle)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`qHandle`	Specifies the queue handle for the channel(s) to be deactivated.

Outputs: None.

Returns:
`SUCCESS`
`ERROR`
`AAL1_BAD_DEV_NO`
`AAL1_BAD_POINTER`
`AAL1_INVALID_PARAM`
`AAL1_INVALID_QUEUE`

aal1AssociateChannel

Description: Associates more channels to an existing mapping. After configuring the mapping, enables it. Uses the configuration of existing channels to associate the new channels.
This function may affect traffic on the existing channels.

Invocation:
```
int aal1AssociateChannel (UINT4 u4DeviceId, int qHandle,
UINT4 u4Channels)
```

Inputs:

Argument	*Description*
u4DeviceId	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
qHandle	Specifies the queue handle of the existing channel(s) to be associated with an existing VP/VC mapping.
u4Channels	Specifies the bit map of additional channel(s). The LSB is channel number 0.

Outputs: None.

Returns:
```
SUCCESS
ERROR
AAL1_BAD_DEV_NO
AAL1_BAD_POINTER
AAL1_INVALID_MODE
AAL1_INVALID_PARAM
AAL1_NO_TX_QUEUES
```

aal1DisassociateChannel

Description: Disassociates already mapped channels from an existing mapping. After configuring the mapping, enables it. If all channels are disassociated from the mapping, the mapping will not be enabled but the queue handle will remain available for future mapping.

Invocation:
```
int aal1DissociateChannel (UINT4 u4DeviceId, int qHandle,
UINT4 u4Channels)
```

Inputs:

Argument	*Description*
u4DeviceId	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
qHandle	Specifies the queue handle of the existing channel(s) to be disassociated with an existing VP/VC mapping.
u4Channels	Specifies the bit map of the channel(s) to be disassociated. The LSB is channel number 0.

Outputs: None.

Returns:
```
SUCCESS
ERROR
AAL1_BAD_DEV_NO
AAL1_BAD_POINTER
AAL1_INVALID_MODE
AAL1_INVALID_PARAM
AAL1_NO_TX_QUEUES
```

aal1EnableTxCond

Description: Enables transmit conditioning for an existing channel(s) to a VP/VC mapping.

Invocation: `int aal1EnableTxCond(UINT4 u4DeviceId, int qHandle)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`qHandle`	Specifies the queue handle of the existing channel(s) for which the signaling will be changed.

Outputs: None.

Returns:
`SUCCESS`
`AAL1_BAD_DEV_NO`
`AAL1_BAD_POINTER`

aal1DisableTxCond

Description: Disables transmit conditioning for any existing channel(s) to a VP/VC mapping.

Invocation: `int aal1DisableTxCond(UINT4 u4DeviceId, int qHandle)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`qHandle`	Specifies the queue handle of the existing channel(s) for which the signaling will be changed.

Outputs: None.

Returns:
`SUCCESS`
`AAL1_BAD_DEV_NO`
`AAL1_BAD_POINTER`

aal1EnableRxCond

Description: Enables receive conditioning for an existing channel(s) to a VP/VC mapping.

Invocation: `int aal1EnableRxCond(UINT4 u4DeviceId, int qHandle)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`qHandle`	Specifies the queue handle of the existing channel(s) for which the signaling will be changed.

Outputs: None.

Returns:
`SUCCESS`
`AAL1_BAD_DEV_NO`
`AAL1_BAD_POINTER`

aal1DisableRxCond

Description: Disables transmit conditioning for an existing channel(s) to a VP/VC mapping.

Invocation: `int aal1DisableRxCond(UINT4 u4DeviceId, int qHandle)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`qHandle`	Specifies the queue handle of the existing channel(s) for which the signaling will be changed.

Outputs: None.

Returns:
`SUCCESS`
`AAL1_BAD_DEV_NO`
`AAL1_BAD_POINTER`

Processing OAM Cells

The AAL1gator II Driver can be used to send and receive OAM cells to and from the ATM network. Use the following functions for those purposes:

- `aal1TxOAMcell()`
- `aal1RxOAMcell()`

aal1TxOAMcell

Description: Transmits an OAM cell. The payload of the cell is initialized in the initialization of the device.

Invocation: `int aal1TxOAMcell (UINT4 u4DeviceId, void *pOAMhead)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`pOAMhead`	Points to the head of the OAM cell.

Outputs: None.

Returns:
`SUCCESS`
`ERROR`
`AAL1_BAD_DEV_NO`
`AAL1_INVALID_PARAM`

NOTE: The `aal1TxOAMcell` function can fail for the following reasons:

- Cells are waiting in the two cell "slots".
- The internal pacing function determines that too many OAM cells are sent in the last "n" intervals. The number of cells is set at initialization by changing the `AAL1_PACE_COUNT` constant (in aal1port.h).

aal1RxOAMcell

Description: Copies the received OAM cell to the passed buffer. The ISR or the DCT may call this function.

Invocation: `int aal1RxOAMcell (UINT4 u4DeviceId, void *pOAMcell)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`pOAMcell`	Points to the area to which OAM cells will be copied.

Outputs: None.

Returns:
`AAL1_BAD_DEV_NO`
`AAL1_CRC_FAIL`
`AAL1_CRC_PASS`
`AAL1_INVALID_PARAM`

Controlling the Synchronous Residual Time Stamp (SRTS)

If the line is being used in Unstructured Data Format (UDF) mode, clock synchronization between the two ends of the connection can be achieved using the SRTS enable feature. The following functions allow you to do this:

- `aal1EnableSRTS()`
- `aal1DisableSRTS()`

aal1EnableSRTS

Description: Enables SRTS for the given T1 or E1 line of the AAL1gator II device. SRTS can be enabled only if the line is in UDF mode.

Invocation: `int aal1EnableSRTS(UINT4 u4DeviceId, UINT4 u4LineNo)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`u4LineNo`	Specifies which line number to configure. Valid values are 0 to (AAL1_MAX_LINES - 1).

Outputs: None.

Returns:
`SUCCESS`
`AAL1_BAD_DEV_NO`
`AAL1_INVALID_PARAM`

NOTE: As PMC-Sierra understands it, Bellcore will not license the SRTS patent to silicon manufacturers. Instead, it is Bellcore's desire to license the SRTS patent under a royalty arrangement only to equipment manufacturers. The ATM Forum states that Bellcore must make this patent available under fair and equitable conditions. Bellcore believes they are satisfying this requirement by offering the license to the equipment manufacturers rather than to the silicon manufacturers.

aal1DisableSRTS

Description: Disables SRTS for the given T1 or E1 line of the AAL1gator II device.

Invocation: `int aal1DisableSRTS(UINT4 u4DeviceId, UINT4 u4LineNo)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`u4LineNo`	Specifies which line number to configure. Valid values are 0 to (AAL1_MAX_LINES - 1).

Outputs: None.

Returns:
`SUCCESS`
`AAL1_BAD_DEV_NO`
`AAL1_INVALID_PARAM`

STATISTICS API FUNCTIONS

This section contains the API functions that you may need for network management or benchmarking purposes. For example, for network management purposes, you may need to read and monitor different registers and different status within the device, or you may need to benchmark the hardware by changing the defaults for the operational mode of the device.

The AAL1gator II Driver compiles all statistics from the device. The AAL1gator II Driver periodically collects statistics, and allows a higher layer to access these values using the following functions:

- `aal1GetRIncorrectSn`
- `aal1GetRIncorrectSnp`
- `al1GetRecvUnderrun`
- `aal1GetRecvOverrun`
- `aal1GetPtrMismatch`
- `aal1GetTCellCount`
- `aal1GetRCellCount`
- `aal1GetRLostCellCount`
- `aal1GetRDroppedCellCount`
- `aal1GetRMisinsertedCellCount`

The API functions use the handle returned by the AAL1gator II Driver when connections are set up.

Obtaining Counts of Out-of-Sequence Cells or Cells with Uncorrectable SN CRCs

aal1GetRIncorrectSn

Description: Obtains the number of AAL1 cells received out-of-sequence on a mapping.

Invocation: `int aal1GetRIncorrectSn(UINT4 u4DeviceId, int qHandle)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`qHandle`	Specifies the queue handle of the existing channel(s) to be queried.

Outputs: None.

Returns: Number of AAL1 cells received out-of-sequence.
`ERROR`

aal1GetRIncorrectSnp

Description: Obtains the number of AAL1 cells received with an uncorrectable sequence number Cyclic Redundancy Check (CRC).

Invocation: `int aal1GetRIncorrectSnp(UINT4 u4DeviceId, int qHandle)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`qHandle`	Specifies the queue handle of the existing channel(s) to be queried.

Outputs: None.

Returns: Number of AAL1 cells received with an uncorrectable sequence number CRC.
`ERROR`

Obtaining Counts of Receive Underruns or Overruns

aal1GetRevcUnderrun

Description: Obtains the number of receive underruns on a mapping.

Invocation: `int aal1GetRecvUnderrun(UINT4 u4DeviceId, int qHandle)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`qHandle`	Specifies the queue handle of the existing channel(s) to be queried.

Outputs: None.

Returns: Number of receive underruns.
`ERROR`

aal1GetRevcOverrun

Description: Obtains the number of receive overruns on a mapping.

Invocation: `int aal1GetRecvOverrun(UINT4 u4DeviceId, int qHandle)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`qHandle`	Specifies the queue handle of the existing channel(s) to be queried.

Outputs: None.

Returns: Number of receive overruns.
`ERROR`

Obtaining Counts of AAL1 Cells Transmitted or Received

aal1GetTCellCount

Description: Obtains the number of AAL1 cells transmitted on a mapping.

Invocation: `int aal1GetTCellCount(UINT4 u4DeviceId, int qHandle)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`qHandle`	Specifies the queue handle of the existing channel(s) be queried.

Outputs: None.

Returns: Number of AAL1 cells transmitted.
`ERROR`

aal1GetRCellCount

Description: Obtains the number of AAL1 cells received on a mapping.

Invocation: `int aal1GetRCellCount(UINT4 u4DeviceId, int qHandle)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`qHandle`	Specifies the queue handle of the existing channel(s) to be queried.

Outputs: None.

Returns: Number of AAL1 cells received.
`ERROR`

Obtaining Counts of Pointer Mismatches or Pointer Reframes

aal1GetPtrMismatch

Description: Obtains the number of receive pointer mismatches on a mapping.

Invocation: `int aal1GetPtrMismatch(UINT4 u4DeviceId, int qHandle)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`qHandle`	Specifies the queue handle of the existing channel(s) be queried.

Outputs: None.

Returns: Number of receive pointer mismatches.
`ERROR`

aal1GetRPtrReframeCount

Description: Obtains the number of AAL1 pointer reframes.

Invocation: `int aal1GetRPtrReframeCount(UINT4 u4DeviceId, int qHandle)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`qHandle`	Specifies the queue handle of the existing channel(s) to be queried.

Outputs: None.

Returns: Number of AAL1 cells transmitted.
`ERROR`

Obtaining Counts of Lost or Replaced AAL1 Cells

aal1GetRLostCellCount

Description: Obtains the number of AAL1 cells lost.

Invocation: `int aal1GetRLostCellCount(UINT4 u4DeviceId, int qHandle)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`qHandle`	Specifies the queue handle of the existing channel(s) to be queried.

Outputs: None.

Returns: Number of AAL1 cells transmitted.
`ERROR`

aal1GetRReplacedCellCount

Description: Obtains the number of AAL1 cells replaced.

Invocation: `int aal1GetRReplacedCellCount(UINT4 u4DeviceId, int qHandle)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`qHandle`	Specifies the queue handle of the existing channel(s) to be queried.

Outputs: None.

Returns: Number of AAL1 cells transmitted.
`ERROR`

Obtaining Counts of Dropped or Misinserted AAL1 Cells

aal1GetRDroppedCellCount

Description: Obtains the number of AAL1 cells dropped.

Invocation: `int aal1GetRDroppedCellCount(UINT4 u4DeviceId, int qHandle)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`qHandle`	Specifies the queue handle of the existing channel(s) to be queried.

Outputs: None.

Returns: Number of AAL1 cells dropped.
`ERROR`

aal1GetRMisinsertedCellCount

Description: Obtains the number of AAL1 cells misinserted.

Invocation: `int aal1GetRMisinsertedCellCount(UINT4 u4DeviceId, int qHandle)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`qHandle`	Specifies the queue handle of the existing channel(s) to be queried.

Outputs: None.

Returns: Number of AAL1 cells misinserted.
`ERROR`

ADDITIONAL CONFIGURATION FUNCTIONS

Obtaining or Setting the Maximum Buffer Depth for a Queue

aal1GetMaxBuf

Description: Obtains the maximum buffer depth for a given queue.

Invocation: `int aal1GetMaxBuf(UINT4 u4DeviceId, int qHandle)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`qHandle`	Specifies the queue handle of the existing channel(s) to be queried.

Outputs: None.

Returns: The maximum buffer depth.
`ERROR`

aal1SetMaxBuf

Description: Sets the maximum buffer depth for a given queue.

Invocation: `int aal1SetMaxBuf(UINT4 u4DeviceId, int qHandle, UINT2 u2depth)`

Inputs:

Argument	*Description*
`u4DeviceId`	Identifies the particular AAL1gator II device. Valid values are 0 to (AAL1_MAX_DEVICES - 1).
`qHandle`	Specifies the queue handle of the existing channel(s) to be queried.
`u2depth`	Specifies the maximum queue depth.

Outputs: None.

Returns: `SUCCESS`
`ERROR`

Appendix A
Error Codes

Table 10 lists the error codes that are used by the AAL1gator II Driver.

Table 10. AAL1gator II Driver Error Codes

Error Code	Description
AAL1_BAD_DEV_NO	Indicates an invalid device number was passed to this API function. (Defined by the value INVALID_DEVICE_NO in the gport.h file).
AAL1_BAD_POINTER	Indicates a bad pointer was passed to this API function.
AAL1_CRC_FAIL	Indicates the CRC failed.
AAL1_CRC_PASS	Indicates the CRC passed.
AAL1_INVALID_MODE	Indicates this API function is not valid in this mode.
AAL1_INVALID_PARAM	Indicates an invalid parameter was sent to an API. (Defined by the value INVALID_VALUE in the gport.h file).
AAL1_INVALID_QUEUE	Indicates an invalid queue handle was passed to this API function.
AAL1_LINENO_VC_MISMATCH	Indicates the line number and VC do not correspond.
AAL1_MAX_VCS_ERR	Indicates no more VCs are available.
AAL1_NO_QUEUES	Indicates no more queues are available.
AAL1_NO_RX_QUEUES	Indicates no more receive queues are available.
AAL1_NO_TX_QUEUES	Indicates no more transmit queues are available.
ERROR	Indicates a general error condition.

CONTACTING PMC-SIERRA, INC.

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
Application Information: apps@pmc-sierra.com
(604) 415-4533
Web Site: http://www.pmc-sierra.com

PMC-980622 (P2) Issue date: November, 1998