VisualDSP++® 4.0 Device Drivers and System Services Manual for Blackfin® Processors

Revision 1.0, February 2005

Part Number
82-000430-01

Analog Devices, Inc.
One Technology Way
Norwood, Mass. 02062-9106

ANALOG
DEVICES

CONTENTS

INTRODUCTION

INTERRUPT MANAGER

PREFACE

Thank you for using Analog Devices, Inc. development software for Analog Devices embedded processors.

Purpose of This Manual

The *Device Drivers and System Services Manual for Blackfin Processors* contains information about the Analog Devices Device Driver Model and System Services library suite. Included are architectural descriptions of the device driver design, and each of the System Service components. Also included is a description of the APIs into each library.

Intended Audience

The primary audience for this manual is a programmer who is familiar with Analog Devices processors. This manual assumes that the audience has a working knowledge of the appropriate processor architecture and instruction set. Programmers who are unfamiliar with Analog Devices processors can use this manual, but should supplement it with other texts (such as the appropriate hardware reference and programming reference manuals) that describe your target architecture.

Manual Contents Description

This manual contains:

- Chapter 1, "Introduction"
  provides an overview of System Services and Device Drivers
- Chapter 2, "Interrupt Manager"
  describes the System Interrupt Controller (SIC) Manager that supports the general-purpose interrupt events
- Chapter 3, "Power Management Module"
  describes the Power Management module that supports Dynamic Power Management of Blackfin processors
- Chapter 4, "External Bus Interface Unit Module"
  describes the External Bus Interface Unit (EBIU) module that is used to enable the Power Management module to manage the SDRAM Controller operation
- Chapter 5, "Deferred Callback Manager"
  describes the Deferred Callback Manager that is used by the application developer to effectively execute function calls
- Chapter 6, "DMA Manager"
  describes Direct Memory Access (DMA) Manager API
- Chapter 7, "Device Driver Manager"
  describes the device driver model used to control devices, both internal and external, to ADI processors

Technical or Customer Support

You can reach Analog Devices, Inc. Customer Support in the following ways:

- Visit the Embedded Processing and DSP products Web site at `http://www.analog.com/processors/technicalSupport`
- E-mail tools questions to `dsptools.support@analog.com`
- E-mail processor questions to `dsp.support@analog.com`
- Phone questions to **1-800-ANALOGD**
- Contact your Analog Devices, Inc. local sales office or authorized distributor
- Send questions by mail to:

```
Analog Devices, Inc.
One Technology Way
P.O. Box 9106
Norwood, MA 02062-9106
USA
```

Product Information

You can obtain product information from the Analog Devices Web site, from the product CD-ROM, or from the printed publications (manuals).

Analog Devices is online at `www.analog.com`. Our Web site provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

MyAnalog.com

MyAnalog.com is a free feature of the Analog Devices Web site that allows customization of a Web page to display only the latest information on products you are interested in. You can also choose to receive weekly E-mail notification containing updates to the Web pages that meet your interests. MyAnalog.com provides access to books, application notes, data sheets, code examples, and more.

Registration:

Visit www.myanalog.com to sign up. Click **Register** to use MyAnalog.com. Registration takes about five minutes and serves as means for you to select the information you want to receive.

If you are already a registered user, just log on. Your user name is your E-mail address.

Processor Product Information

For information on embedded processors and DSPs, visit our Web site at www.analog.com/processors, which provides access to technical publications, data sheets, application notes, product overviews, and product announcements.

You may also obtain additional information about Analog Devices and its products in any of the following ways.

- E-mail questions or requests for information to
  `dsp.support@analog.com`
- Fax questions or requests for information to
  1-781-461-3010 (North America)
  089/76 903-557 (Europe)
- Access the FTP Web site at
  `ftp ftp.analog.com` or `ftp 137.71.23.21`
  `ftp://ftp.analog.com`

Related Documents

For software/tools information, refer to VisualDSP++ user's documentation available online and in printed forms.

For hardware information, refer to your processors's hardware reference, programming reference, or data sheet. All documentation is available online. Most documentation is available in printed form.

Visit the Technical Library Web site to access all processor and tools manuals and data sheets:

`http://www.analog.com/processors/resources/technicalLibrary`

Online Technical Documentation

Online documentation includes the VisualDSP++ Help system, software tools manuals, hardware tools manuals, processor manuals, Dinkum Abridged C++ library, and Flexible License Manager (FlexLM) network license manager software documentation. You can easily search across the entire VisualDSP++ documentation set for any topic of interest using the Search function of VisualDSP++ Help system. For easy printing, supplementary `.PDF` files of most manuals are also provided.

Each documentation file type is described as follows.

File	Description
`.CHM`	Help system files and manuals in Help format
`.HTM` or `.HTML`	Dinkum Abridged C++ library and FlexLM network license manager software documentation. Viewing and printing the `.HTML` files requires a browser, such as Internet Explorer 4.0 (or higher).
`.PDF`	VisualDSP++ and processor manuals in Portable Documentation Format (PDF). Viewing and printing the `.PDF` files requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher).

Access the online documentation from the VisualDSP++ environment, Windows® Explorer, or the Analog Devices Web site.

Accessing Documentation From VisualDSP++

From the VisualDSP++ environment:

- Access VisualDSP++ online Help from the Help menu's **Contents**, **Search**, and **Index** commands.
- Open online Help from context-sensitive user interface items (toolbar buttons, menu commands, and windows).

Accessing Documentation From Windows

In addition to any shortcuts you may have constructed, there are many ways to open VisualDSP++ online Help or the supplementary documentation from Windows.

Help system files (.CHM) are located in the Help folder of VisualDSP++ environment. The .PDF files are located in the Docs folder of your VisualDSP++ installation CD-ROM. The Docs folder also contains the Dinkum Abridged C++ library and the FlexLM network license manager software documentation.

Using Windows Explorer

- Double-click the vdsp-help.chm file, which is the master Help system, to access all the other .CHM files.
- Open your VisualDSP++ installation CD-ROM and double-click any file that is part of the VisualDSP++ documentation set.

Using the Windows Start Button

- Access VisualDSP++ online Help by clicking the **Start** button and choosing **Programs**, **Analog Devices**, **VisualDSP++**, and **VisualDSP++ Documentation**.

Accessing Documentation From the Web

Download manuals in PDF format at the following Web site:

http://www.analog.com/processors/resources/technicalLibrary/manuals

Select a processor family and book title. Download archive (.ZIP) files, one for each manual. Use any archive management software, such as WinZip, to decompress downloaded files.

Printed Manuals

For general questions regarding literature ordering, call the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**) and follow the prompts.

VisualDSP++ Documentation Set

To purchase VisualDSP++ manuals, call **1-603-883-2430**. The manuals may be purchased only as a kit.

If you do not have an account with Analog Devices, you are referred to Analog Devices distributors. For information on our distributors, log onto `http://www.analog.com/salesdir/continent.asp`.

Hardware Tools Manuals

To purchase EZ-KIT Lite™ and In-Circuit Emulator (ICE) manuals, call **1-603-883-2430**. The manuals may be ordered by title or by product number located on the back cover of each manual.

Processor Manuals

Hardware reference and instruction set reference manuals may be ordered through the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**), or downloaded from the Analog Devices Web site. Manuals may be ordered by title or by product number located on the back cover of each manual.

Data Sheets

All data sheets (preliminary and production) may be downloaded from the Analog Devices Web site. Only production (final) data sheets (Rev. 0, A, B, C, and so on) can be obtained from the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**); they also can be downloaded from the Web site.

To have a data sheet faxed to you, call the Analog Devices Faxback System at **1-800-446-6212**. Follow the prompts and a list of data sheet code numbers will be faxed to you. If the data sheet you want is not listed, check for it on the Web site.

Notation Conventions

Text conventions used in this manual are identified and described as follows.

Example	Description
Close command (**File** menu)	Titles in reference sections indicate the location of an item within the VisualDSP++ environment's menu system (for example, the **Close** command appears on the **File** menu).
`{this | that}`	Alternative required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as `this` or `that`. One or the other is required.
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.
`[this,…]`	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipse; read the example as an optional comma-separated list of `this`.
`.SECTION`	Commands, directives, keywords, and feature names are in text with `letter gothic` font.
filename	Non-keyword placeholders appear in text with italic style format.
ⓘ	**Note:** For correct operation, ... A Note provides supplementary information on a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
⚡	**Caution:** Incorrect device operation may result if ... **Caution:** Device damage may result if ... A Caution identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word **Caution** appears instead of this symbol.
🛇	**Warning:** Injury to device users may result if ... A Warning identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for devices users. In the online version of this book, the word **Warning** appears instead of this symbol.

ⓘ Additional conventions, which apply only to specific chapters, may appear throughout this document.

1 INTRODUCTION

This manual describes the System Services and Device Driver architecture for Analog Devices processors.

The System Services form a collection of functions that are commonly found in embedded systems. Each system service focuses on a specific set of functionality such as Direct Memory Access (DMA), Power Management (PM), Interrupt Control (IC), and so on. Collectively, the system services provide a wealth of pre-built, optimized code that simplifies software development for users, allowing them to get their Blackfin processor-based designs to market more quickly.

The Device Driver model provides a simple, clean and familiar interface into device drivers for Blackfin processors. The primary objective of the device driver model is to create a concise, effective and easy to use interface through which applications can communicate with device drivers. Secondarily, the model and device manager software, significantly simplifies the development of device drivers, making it very straightforward for the development of new device drivers.

System Services Overview

The current revision of the System Services library consists of five services:

- Interrupt Control Service - The Interrupt Control service allows the application to control and leverage the event and interrupt processing of the processor more effectively. Specific functionality allows the application to:
  - Set and detect the mappings of the interrupt priority levels to peripherals.
  - Use standard ‘C’ functions as interrupt handlers.
  - Hook and unhook multiple interrupt handlers to the same interrupt priority level using both nesting and non-nesting capabilities.
  - Detect if a system interrupt is being asserted.
  - Protect and unprotect critical regions of code in a portable manner.
- Power Management Service - The Power Management service allows the application to control the Dynamic Power Management capabilities of the Blackfin processor. Specific functionality allows the application to:
  - Set core and system clock operating frequencies via a function call.
  - Set and detect the internal voltage regulator settings.
  - Transition the processor among the various operating modes including, Full-On, Active, Sleep, and so on.

- External Bus Interface Unit Control Service (EBIU) - The EBIU Control service provides a collection of routines to set up the external interfaces of the Blackfin processor, including the SDRAM controller. This functionality enables users to:
    - Adjust SDRAM refresh and timing rates to optimal values for given system clock frequencies.
    - Set individual bus interface settings.
    - Complete single function setup for known configurations, such as the Blackfin EZ-Kits.
- Deferred Callback Service - The Deferred Callback service allows the application to be notified of asynchronous events outside of high priority interrupt service routines. Using deferred callbacks typically improves the overall I/O capacity of the system while at the same time reducing interrupt latency. Specific functionality allows the application to:
    - Define how many callbacks can be pending at any point in time.
    - Define the interrupt priority level at which the callback service executes.
    - Create multiple callback services, each operating at a different interrupt priority level.
    - Post callbacks to a callback service with a relative priority among all other callbacks posted to the same callback service.
- DMA Management Service - The DMA Management service provides access into the DMA controller of the Blackfin processor. The DMA Management service allows the application to schedule

DMA operations, both peripheral and memory DMA, supporting both linear and two-dimensional transfer types. Specific functionality allows the application to:

- Set and detect the mapping of DMA channels to peripherals.
- Configure individual DMA channels for inbound/outbound traffic using circular (autobuffered) DMA or descriptor based DMA.
- Command the DMA Manager to issue "live" or deferred callbacks upon DMA completions.
- Queue descriptors, intermixing both linear and two-dimensional transfers, on DMA channels.
- Enable the DMA Manager to loopback on descriptor chains automatically.
- Continuously stream data into or out from a memory stream or peripheral.
- Initiate linear and two-dimensional memory DMA transfers with simple 'C' like, `memcpy`-type functions.

- Device Manager - The device driver model is used to control devices, both internal and external to Analog Devices processors. Specific functionality allow the application to:
  - Open and close devices used by the application.
  - Configure and control devices.
  - Receive and transmit data through the devices using a variety of dataflow methods.

Application Interface

Each system service exports an API that defines the interface into that service. Application software makes calls into the API of the system service to access the functionality that is to be controlled.

Each API is designed to be called using the standard calling interface of the development toolset's 'C' run-time model. The API of each service can be called by any 'C' or assembly language program that adheres to the calling conventions and register usage of the 'C' run-time model.

In addition to the application software using the API to make calls into a system service, some system services make calls into the API of other system services. For the most part, each service operates independently of the other services; however redundancies are eliminated by allowing one service to access the functionality of another service.

For example, should the application need to be notified when a DMA descriptor has completed processing, and the application has requested deferred callbacks. In this case, the DMA Management service invokes the Deferred Callback service to effect the callback into the application.

Another example of combined operation between services is in the case of the Power Management and EBIU services. Assume that the system has SDRAM and the application needs to conserve power by turning down the core and system clock frequencies. When the application calls the Power Management service to lower the operating frequencies, the Power Management service automatically invokes the EBIU service, which adjusts the SDRAM refresh rate to compensate for the reduced system clock frequency.

Figure 1-1 illustrates the current collection of system services and the API interactions among them.

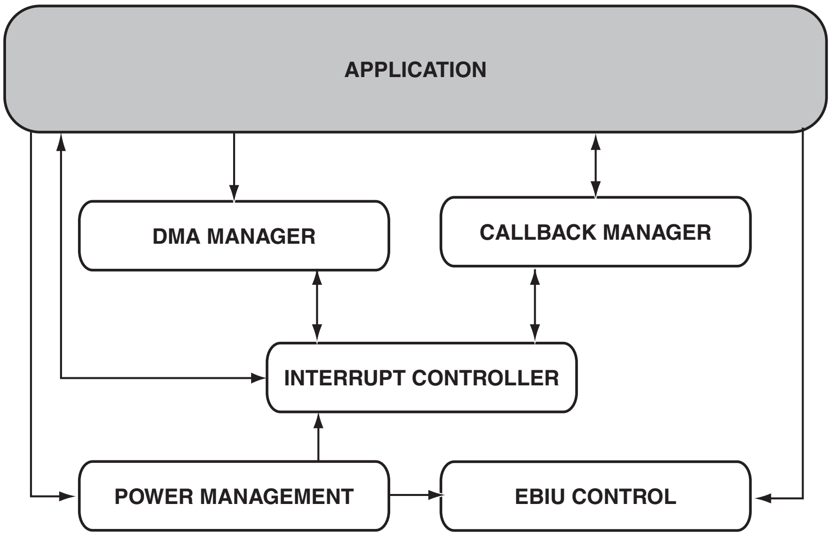

Figure 1-1. System Services and API Interactions

Dependencies

With few constraints, applications can choose to use any individual service or combination of services within their application. Applications do not have to use each and every service. Further, each service does not need all the resources associated with the system the service is controlling. For example, the DMA Manager does not need control over each and every DMA channel. The system can be configured for the DMA Manager to control some channels, leaving the application or some other software to control other DMA channels. (See the individual service chapters for more information on each individual service.) There are however, some dependencies within the services of which the application developer should be aware.

All the current services, except for the EBIU service, invoke the Interrupt Control service for the management of interrupt processing. The DMA Manager, Callback and Power Management Services each depend on the IC service to manage interrupt processing for them.

If directed by the application to adjust SDRAM timing automatically, the Power Management Service uses the EBIU Control Service to affect SDRAM timing parameter changes when the power/operating speed profile of the processor is changed.

When configured to use deferred callbacks (as opposed to "live" or interrupt-time callbacks) the DMA Manager leverages the capabilities of the Deferred Callback Service to provide deferred callbacks to the application. When configured for "live" callbacks however, the DMA Manager does not make use of the Deferred Callback Service.

The development toolset automatically determines these dependencies and links into the executable only those services that are required by the application. As each service is built as its own object file within the System Services Library file, it is possible to further reduce code size of the final executable by commanding the linker to eliminate any unused objects. Refer to the development toolset documentation for more information.

Initialization

The API of each system service includes an initialization function that must be called by the application, prior to accessing the other API functions of the service. All initialization functions of the services are of the form `adi_xxx_Init()` where `xxx` is the service name abbreviation. The initialization functions typically provide any data memory that is required of the service and perform any setup and initialization of the systems being controlled.

After reset, many applications need to adjust their operating frequency or voltage and configure the external memories of the system. As such, the Power Management and EBIU services are typically accessed when the application starts. Since the Power Management service uses the functionality of the Interrupt Manager to control the PLL, the Interrupt Manager should be initialized prior to the Power Management service. It is also preferable to initialize the Power Management service prior to the EBIU service.

The remaining services, assuming the application is using some combination of the remaining services, may require data memory as part of their initialization. As some applications provide external memory to these services, this is another reason to initialize the Power Management and EBIU services before any of the others.

In summary, most applications find the initialization sequence below optimal for their application. Any service not used by the application should simply be omitted from the sequence.

1. Interrupt Control Service
2. Power Management Service
3. EBIU Service
4. Deferred Callback Service
5. DMA Manager Service

After all services that are to be used in the application have been initialized, the remaining API functions in any of the services may be called by the application.

Termination

The API of each system service also includes a termination function that may be called by the application if the functionality of a service is no longer required. All termination functions of the services are of the form `adi_xxx_Terminate()` where `xxx` is the service name abbreviation. Many embedded systems run in an endless operating loop and never call the termination function of a service. Applications that operate in endless loops can save program memory by not calling the terminate functions.

Termination functions may make calls into other services whose functionality they may be using. For example, the DMA Manager uses the services of the Interrupt Control service to manage interrupts. As part of the process of terminating the DMA Manager Service, all DMA interrupt handlers are unhooked via the Interrupt Control service. This means the DMA Manager Service should be terminated prior to the Interrupt Control Service being terminated.

The sequence described below ensures that services are closed in a logical sequence and ensures that no service is closed unknowingly to some other service. Most applications find the initialization sequence below optimal for their application. Any service not used by the application should simply be omitted from the sequence.

1. DMA Manager Service
2. Deferred Callback Service
3. EBIU Service
4. Power Management Service
5. Interrupt Control Service

After a service has been terminated, it must be re-initialized before any of its functionality can be accessed again.

System Services Directory and File Structure

All files for the System Services are contained within the `blackfin` directory tree. In VisualDSP++ installations this is the same directory as the one used for core development tools. Other development toolsets may use other directory names for their toolkits, but the System Services can always be found within the `blackfin` directory tree.

To use the System Services, applications need only include a single include file in their source code, and link with a single System Services Library module that is appropriate for their configuration.

Accessing the System Services API

Applications using the System Services should include the `blackfin/include/services` directory in the compilers/assemblers pre-processor search path. User source files accessing any of the System Services APIs should simply include the `services.h` file, located in the `blackfin/include/services` directory. User files do not need to include any other files to use the System Services API.

The System Services API and functionality are uniform and consistent across all Blackfin processors, including all single and multi-core devices. Application software does not have to change regardless of which Blackfin processor is being targeted. For example application software running on a single-core ADSP-BF533 processor can operate unchanged on a multi-core ADSP-BF561 processor.

In order to provide this consistent API to the application, the System Services API needs to be aware of the specific processor variant being targeted. The user should ensure that the processor definition macro for the processor variant being targeted is defined when including the `services.h` include file.

The VisualDSP++ toolset automatically sets the processor definition macro when building projects. Application developers using the VisualDSP++ toolset need do nothing further to ensure the processor definition macro is defined.

Application developers using other toolsets, however, should ensure the processor definition macro is appropriately defined. The `services.h` file enumerates the specific processor variants that are supported. These processor variants include:

__ADSPBF531__	The ADSP-BF531 processor
__ADSPBF532__	The ADSP-BF532 processor
__ADSPBF533__	The ADSP-BF533 processor
…	

The `services.h` file contains the full and complete list of processor variants that are supported.

ⓘ Note: Although the API of the System Services does not change between processor variants, the internals of the System Services differ depending on the specific processor variant and processor revision number being targeted. For example, the number of DMA channels for the ADSP-BF533 differs from the number of DMA channels for the ADSP-BF561. Further, a workaround within the services for revision `x.y` of a processor may not be needed for revision `x.y` of that same processor. These differences are accounted for in the System Service Library module. See "System Services Directory and File Structure" for more information.

Linking in the System Services Library

All object code for the System Services is included in the System Services library file. This file is found in the `blackfin/lib` directory. In this directory is a System Services library file for each processor variant and processor revision that is supported. The user should ensure that the appropriate library is included in the list of object files for the linker. All System Service Library files are of the form `libsslxxx_yyyzz.dlb` where:

- `xxx` represents the processor variant - This is typically a 3-digit number identifying the processor variant, such as 532 for the ADSP-BF532, 534 for the ADSP-BF534, and so on.
- `_yyy` represents the operating environment - This suffix represents the operating environment being targeted such as `vdk` for VDK-based systems, `linux` for Linux-based systems, and so on. Libraries built for `standalone`, specifically non-RTOS environments, do not include the `_yyy` suffix.
- `zz` represents any special conditions for the library. The following combinations are used:
    - `d` - The library contains all debug information for the file.
    - `y` - The library is built to avoid all known anomalies for all revisions of silicon.
    - `dy` - The library contains all debug information for the file and is built to avoid all known anomalies for all revisions of silicon.
    - blank - A library without any additional suffix does not include debug information and does not contain workarounds to any anomalies.

Located within the `blackfin/lib` directory are subdirectories for individual silicon revisions. The libraries in these subdirectories are built for specific silicon revisions of the Blackfin processors.

Only a single System Services Library file should be included for the linker to process. Application developers should choose the correct library based on the processor variant, operating environment, and processor revision number for their system.

For example, an application developer who wants a debug version of the System Services Library and is targeting silicon revision 0.2 of the ADSP-BF532 without any RTOS should link with the `libss1532_d.dlb` file from the `blackfin/lib/bf532_rev_0.2` subdirectory. As another example, the application developer who wants a release version of the System Services Library that will run on any revision of ADSP-BF532 silicon and is using the VDK, should link with the `libss1532_vdky.dlb` file from the `blackfin/lib` directory.

It is strongly recommended that developers use the debug versions of the System Services Library during development as built-in error-checking code within the library can save countless hours of development time.

Rebuilding the System Services Library

Under normal situations, there is no need to rebuild the System Services Library. However, to accommodate unforeseen circumstances and provide the user the ability to tailor the System Services to their particular needs, all source code and include files necessary to rebuild the System Services Library are provided. In addition, VisualDSP++ project files are included for application developers using the VisualDSP++ development toolset.

All code for the System Services Library is located in the following directories:

- `blackfin/lib` - This directory contains the Analog Devices built versions of the System Service library files (`*.dlb`).
- `blackfin/lib/src/services` - This directory contains all the source code files and non-API include files for the System Services. Also in this directory are the VisualDSP++ project files that can be used to rebuild the libraries.
- `blackfin/include/services` - This directory contains all API include files for the System Services.

VisualDSP++ users can simply rebuild the System Services Library by using the `build` command after opening the appropriate VisualDSP++ project file.

To rebuild the libraries using other development toolsets, the following process should be performed:

1. Set the pre-processor include path to include `black-fin/include/services` and `blackfin/lib/src/services`.
2. Define the processor variant according to the definitions in the file `services.h`.
3. Define the silicon revision macro, `__SILICON_REVISION__`, to the proper value. See the `_si_revision` switch in the compiler for more information.

4. Compile/assemble all files in the `blackfin/lib/src/services` directory.
5. Link the appropriate compiled/assembled objects into a library. Include all object files without any operating environment extension (such as `vdk`) and all object files with the appropriate operating environment extension specific for the environment being targeted (such as `vdk`).

Examples

The System Services distribution includes many examples that illustrate how to use the System Services. Please refer to these examples for additional information on how to use the System Services effectively.

Device Driver Overview

Device drivers provide a mechanism for applications to control a device effectively. Devices may be on-chip or off-chip hardware devices, or even software modules that are best managed as virtual devices. Device drivers are typically constructed such that the application is insulated from the nuances of the hardware (or software) being controlled. In this way, both the device drivers and the devices that are being controlled can be updated or replaced without affecting the application.

The Analog Devices Device Driver Model has been created to provide a simple, convenient method for applications to control devices commonly found in and around Analog Devices processors. It has also been designed to provide a simple and efficient mechanism for the creation of new device drivers.

Application Interface

The Device Driver Model provides a consistent, simple and familiar Application Programming Interface (API) for device drivers. All devices drivers that conform to the model use the same simple interface into the driver.

Most devices either receive and/or transmit data, sometimes transforming the data in the process. This data is encapsulated in a buffer. The buffer may contain small bits of data, such as for a UART-type device that processes one character at a time, or large pieces of data, such as a video device that processes NTSC frames of approximately 1MB in size. Applications typically provide the buffers to the device, though it is possible for devices to pass buffers from one device to another without any application involvement.

The actual API into a model-compliant driver consists of the following basic functions:

- `adi_dev_Open()` - Opens a device for use.
- `adi_dev_Close()` - Closes down a device.
- `adi_dev_Read()` - Provides a device with buffers for inbound data.
- `adi_dev_Write()` - Provides a device with buffers for outbound data.
- `adi_dev_Control()` - Sets/detects control and status parameters for a device.

Like the System Service APIs, the Device Driver API is designed to be called using the standard calling interface of the development toolset's 'C' run-time model. The Device Driver API can be called by any C or assembly language program that adheres to the calling conventions and register usage of the C run-time model.

Device Driver Architecture

The Device Driver Model separates the functionality of device drivers into two main components, the Device Manager and the Physical Drivers.

The Device Manager is a software component that provides much of the functionality common to the vast majority of device drivers. For example, depending on how the application wants the device driver to operate, the application may command a device driver to operate in synchronous mode or asynchronous mode. In synchronous mode, when the application calls the `adi_dev_Read()` or `adi_dev_Write()` API function to read data from or send data to the device; the API function does not return to the application until the operation has completed. In asynchronous mode, the API function returns immediately to the application, while the data is moved in the background. It would be wasteful to force each and every device driver (or more accurately, each and every Physical Driver) to provide logic that operates both synchronously and asynchronously. The Device Manager provides this functionality, relieving each Physical Driver from re-implementing this capability.

The Device Manager also provides the API to the application for each device driver. This ensures that the application has the same consistent interface regardless of the peculiarities of each device.

While there is one and only one Device Manager exists in a system, there can be any number of Physical Drivers in a system. A Physical Driver is that component of a device driver that accesses and controls the physical device. The Physical Driver is responsible for all the “bit banging” and control and status register manipulations of the physical device. All device specific information is contained and isolated in the Physical Driver.

This architecture is illustrated in Figure 1-2:

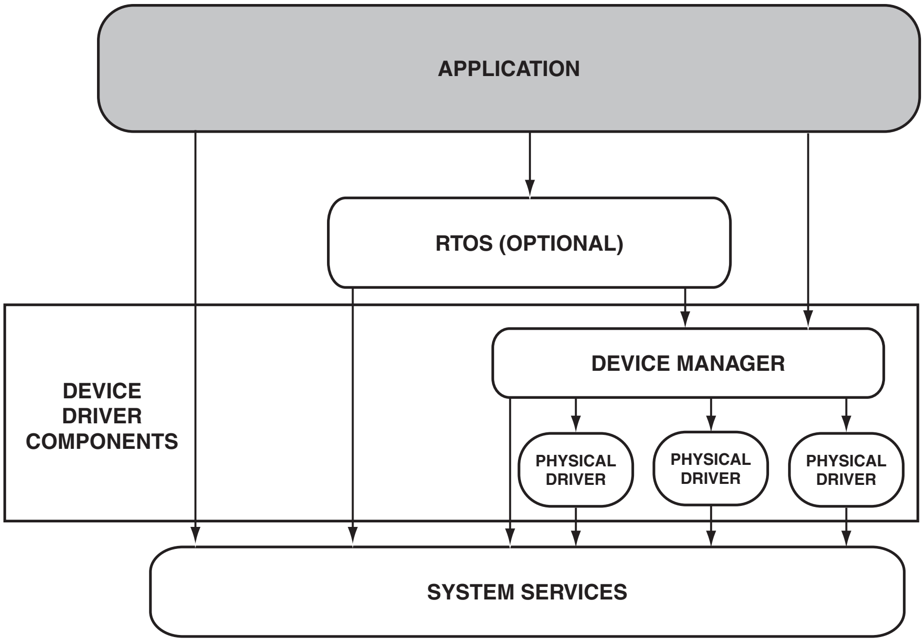

Figure 1-2. Device Manager Architecture

Interaction with System Services

As shown in Figure 1-2, the Device Driver Model leverages the capabilities of the System Services. Each software component in the system, whether it is the application, RTOS (if present), the Device Manager, or each Physical Driver can access and call into the System Services API.

The benefits of using this approach are enormous. In addition to the code size and data memory savings, this approach allows each software component access to the resources of the system and processor in a cooperative manner. Further, the amount of development effort for Physical Drivers is substantially reduced as each driver does not have to re-implement any of the functionality provided by the Device Manager or System Services.

Initialization

Prior to accessing any individual driver, the Device Manager must first be initialized. The initialization function, `adi_dev_Init()`, is called by the application to setup and initialize the Device Manager.

Though the Device Driver Model is dependent upon System Services, the initialization function of the Device Manager does not rely on any of the System Services. As such the current revision of the Device Manager can be initialized either before or after the System Services initialization.

However, future versions of the Device Manager initialization function may require some of the System Services capabilities. As such, it is good practice to initialize the required System Services prior to initializing the Device Manager. Refer to the "Initialization" on page 1-7 for more information on the initialization of the System Services.

Termination

The API of the Device Driver Model includes a termination function that may be called by the application if the device drivers are no longer required. The termination function, `adi_dev_Terminate()`, is called to free up the resources used by the Device Manager and any open Physical Drivers. Many embedded systems run in an endless operating loop and never call the termination function of the Device Manager. Applications that operate in endless loops can save program memory by not calling the terminate function.

As part of the termination function processing, the Device Manager closes all open Physical Drivers. The Physical Drivers are closed in an abrupt manner. If a more graceful shutdown is needed, the application may prefer to close any open Physical Drivers first, and then call the termination function.

Note that because of the reliance on the System Services, the termination function of the Device Manager should be called prior to any termination functions of the System Services. This ensures that the System Services can be called by the Device Manager and/or Physical Drivers as part of their shutdown procedure.

After the Device Manager has been terminated, it must be re-initialized before any of its functionality can be accessed again.

Device Driver Directory and File Structure

All files for the Device Driver Model are contained within the `blackfin` directory tree. In VisualDSP++ installations this is the same directory as the one used for storing the core development tools. Other development toolsets may use other directory names for their toolkits, but the Device Driver files can always be found within the `blackfin` directory tree.

To use the device drivers, applications need only use some include files in their source code, and link with a Device Driver Library and a System Services Library module.

Accessing the Device Driver API

Applications using the Device Driver Model should include the `black-fin/include/services` and `blackfin/include/drivers` directories in the compilers/assemblers pre-processor search path. User source files accessing the Device Manager API should include the files `services.h` and `adi_dev.h`, in order, located in the `blackfin/include/services` and `black-`

fin/include/drivers directories, respectively. In addition, the user's source file should use the include file of the Physical Driver that will be accessed.

For example, user code that is accessing the Analog Devices Parallel Peripheral Interface (PPI) driver would include the following lines in their source file (in order):

```
#include "services.h"     // system services
#include "adi_dev.h"       // device manager
#include "adi_ppi.h"       // ppi physical driver
```

The Device Driver API and functionality is uniform and consistent across all Blackfin processors, including all single and multi-core devices. Application software does not change regardless of which Blackfin processor is being targeted. For example application software running on a single core ADSP-BF533 processor can operate unchanged on a multi core ADSP-BF561 processor.

In order to provide this consistent API to the application, the System Services, Device Manager and Physical Drivers need to be aware of the specific processor variant being targeted. The user should ensure that the processor definition macro for the processor variant being targeted is defined when including the System Services, `services.h`, Device Manager, `adi_dev.h`, and physical driver include files.

The VisualDSP++ toolset automatically sets the processor definition macro when building projects. Application developers using the VisualDSP++ toolset need do nothing further to ensure the processor definition macro is defined.

Application developers using other toolsets should, however, ensure the processor definition macro is appropriately defined. The `services.h` file enumerates the specific processor variants that are supported. These processor variants include:

__ADSPBF531__	The ADSP-BF531 processor
__ADSPBF532__	The ADSP-BF532 processor
__ADSPBF533__	The ADSP-BF533 processor
...	

The `services.h` file contains the full and complete list of processor variants that are supported by the System Services. The `adi_dev.h` file contains the list of processor families that are supported by the Device Driver Model.

Linking in the Device Driver Library

All object code for the Device Manager and Analog Devices-supplied Physical Drivers is included in the Device Driver library file. This file is found in the `blackfin/lib` directory. In this directory is a Device Driver library file for each and every processor variant that are supported. The user should ensure that the appropriate library is included in the list of object files for the linker. The Device Driver Library file is of the form `libdrvxxxzz.dlb` where:

- `xxx` represents the processor variant - This is typically a 3-digit number identifying the processor variant such as 532 for the ADSP-BF532, 534 for the ADSP-BF534, and so on.
- `zz` represents any special conditions for the library. The following combinations are used:
  - `d` - the library contains all debug information for the file.
  - `y` - The library is build to avoid all known anomalies for all revisions of silicon.

- `dy` - The library contains all debug information for the file and is built to avoid all known anomalies for all revisions of silicon.
- blank - A library without an additional suffix does not include debug information and does not contain workarounds to any anomalies.

Located within the `blackfin/library` directory are subdirectories for individual silicon revisions. The libraries in these subdirectories are built for specific silicon revisions of the processors.

Only a single Device Driver Library file should be included for the linker to process. The application developer should choose the correct library based on the processor variant for their system.

For example, an application developer who wants a debug version of the Device Driver Library and is targeting silicon revision 0.2 of the ADSP-BF532 should link with the `libdrv532d.dlb` file from the `blackfin/lib/bf532_rev_0.2` subdirectory. As another example, the application developer who wants a release version of the Device Driver Library that will run on any revision of ADSP-BF532 silicon should link with the `libdrv532y.dlb` file from the `blackfin/lib` directory.

ⓘ It is strongly recommended that developers use the debug versions of the Device Driver Library during development, because built-in error-checking code within the library can save countless hours of development time.

Rebuilding the Device Driver Library

Under normal situations, there is no need to rebuild the Device Driver Library. However, to accommodate unforeseen circumstances and provide the user with the ability to tailor the implementation to their particular needs, all source code and include files necessary to rebuild the Device

Driver Library are provided. In addition, VisualDSP++ project files are included for application developers using the VisualDSP++ development toolset.

All code for the Device Driver Library is located in the following directories:

- `blackfin/lib` - This directory contains the Analog Devices built versions of the Device Driver library files (`*.dlb`).
- `blackfin/lib/src/drivers` - This directory contains all the source code files and non-API include files for the Device Manager and Analog Devices provided Physical Drivers. Also in this directory are the VisualDSP++ project files that can be used to rebuild the libraries.
- `blackfin/include/drivers` - This directory contains the Device Manager API include file and the include files for all Analog Devices provided Physical Drivers.

VisualDSP++ users can rebuild the Device Driver Library by using the `build` command after opening the appropriate VisualDSP++ project file.

To rebuild the libraries using other development toolsets, the following steps should be performed:

1. Set the pre-processor include path to include `blackfin/include/drivers` and `blackfin/lib/src/drivers`.
2. Define the processor variant according to the definitions in the `services.h` file.

3. Define the silicon revision macro, `__SILICON_REVISION__`, to the proper value. See the `-si-revision` switch in the compiler for more information.

4. Compile/assemble all files in the directory `blackfin/lib/src/drivers`.

- Link the appropriate compiled/assembled objects including all object files into a library.

Examples

The Device Driver distribution includes examples that illustrate how to use the Device Drivers. Please refer to these examples for additional information on how to use the Device Drivers effectively.

2 INTERRUPT MANAGER

This chapter describes the Interrupt Manager that controls and manages the interrupt and event operations of the Blackfin processor.

This chapter contains:

Introduction

The Blackfin processor employs a two-tiered mechanism for controlling interrupts and events. System level interrupts are controlled by the *System Interrupt Controller* (SIC). All peripheral interrupt signals are routed through the System Interrupt Controller and then, depending on the settings of the System Interrupt Controller, routed to the *Core Event Controller* (CEC). The Core Event Controller processes these events and, depending on the settings of the Core Event Controller, vectors the processor to handle the events.

The Interrupt Manager provides functions that allow the application to control every aspect of both the System Interrupt Controller and the Core Event Controller. It does this so that events and interrupts are handled and processed in an efficient, yet cooperative, manner.

The Blackfin processor provides 16 levels of interrupt and events. These levels, called *Interrupt Vector Groups* (IVG), are numbered from 0 to 15, with the lowest number having the highest the priority. Some IVG levels are dedicated to certain events, such as emulation, reset, *Non-Maskable Interrupt* (NMI) and so on. Other IVG levels, specifically levels 7 through 15, are considered general purpose events and are typically used for system level (peripheral) interrupts or software interrupts. All IVG processing is performed in the CEC. When a specific IVG is triggered, assuming the event is enabled, the CEC looks up the appropriate entry in the Event Vector Table, and vectors execution to the address in the table where the event is processed.

All system or peripheral interrupts are first routed through the SIC. Assuming the SIC has been so programmed, peripheral interrupts are then routed to the CEC for processing. The SIC provides a rich set of functionality for the processing and handling of peripheral interrupts. In addition to allowing/disallowing peripheral interrupts to be routed to the CEC, the

SIC allows peripheral interrupts to be mapped to any of the CEC's general purpose IVG levels, and controls whether or not these interrupts wake the processor from an idled operating mode.

In systems employing Blackfin processors, there are often more potential interrupt sources than there are IVG levels. As stated above, some events, such as NMI, map one to one to an IVG level. Others, typically infrequent interrupts such as peripheral error interrupts are often "ganged" in a single IVG level.

The Interrupt Manager allows the application to execute complete control over how interrupts are handled, whether they are masked or unmasked, mapped one to one or ganged together, whether the processor should be awakened to service an interrupt and so on. The Interrupt Manager also allows the creation of interrupt handler chains. An interrupt handler is a C-callable function that is provided by the application to process an interrupt. Through the Interrupt Manager, the application can hook in any number of interrupt handlers for any IVG level. In the case where multiple events are ganged to a single IVG level, this allows each handler to be designed independently from any other and allows the application to process these interrupts in a straightforward manner.

Further, the Interrupt Manager only processes those IVG levels and system interrupts that the application directs the Interrupt Manager to control. This allows the application developer to have complete unfettered access to any IVG level or system interrupt, if they want manual control of interrupts.

Multi-core Blackfin processors extend on this by including one System Interrupt Controller and one Core Event Controller for each core. This provides maximum flexibility by allowing application developers to decide how to map interrupts to individual cores, multiple cores and so on. When using multi-core Blackfin processors, typically one Interrupt Manager for each core is used. Because there is no reason to provide multiple

Interrupt Managers on single core devices, this service is not supported. Application developers should not attempt to instantiate more than one Interrupt Manager per core.

Following the convention of all the System Services, the Interrupt Manager uses a unique and unambiguous naming convention to guard against conflicts. All enumeration values, typedefs and macros use the `ADI_INT_` prefix, while all functions within the Interrupt Manager use the `adi_int_` prefix.

All Interrupt Manager API functions return the `ADI_INT_RESULT` return code. See the `adi_int.h` file for the list of return codes. Like all System Services, the return code that signals successful completion, `ADI_INT_RESULT_SUCCESS` for the Interrupt Manager, is defined to be 0, allowing applications to quickly and easily determine if any errors occurred in processing.

Initialization

In order to use the functionality of the Interrupt Manager, the Interrupt Manager must first be initialized. The initialization function of the Interrupt Manager is called `adi_int_Init`. The application passes to the initialization function memory that the Interrupt Manager can use during its lifetime.

The amount of memory that should be provided depends on the number of secondary handlers that are to be used by the application. When using interrupt handler chaining, the Interrupt Manager considers the first interrupt handler that is hooked into an IVG level to be the primary interrupt handler. Any additional interrupt handlers that hooked into that same IVG level are considered secondary handlers. Without any additional memory from the application, the Interrupt Manager can support one primary interrupt handler for each IVG level. If the application never has more than one interrupt handler on each IVG level, in other words only the primary interrupt handler and no secondary handlers are present, then the application does not need to provide memory to the Interrupt Manager's initialization function. If however, the application will be hooking in secondary interrupt handlers, the application needs to provide additional memory to support the secondary handlers. The macro `ADI_INT_SECONDARY_MEMORY` is defined to be the amount of memory, in bytes, that is required to support a single secondary handler. Therefore, the application should provide 'n' times `ADI_INT_SECONDARY_MEMORY` to the initialization function, where 'n' is the number of secondary handlers that are to be supported.

Another parameter passed to the initialization function is the parameter that the Interrupt Manager passes to the `adi_int_EnterCriticalRegion()` function. This value is dependent upon the operating environment of the application. See the `adi_int_EnterCriticalRegion` function below for more information.

When called, the initialization function initializes its internal data structures and returns. No changes are made to either the CEC or SIC during initialization. After initialization, any of the other Interrupt Manager API functions may be called.

Termination

When the functionality of the Interrupt Manager is no longer required, the application can call the termination function of the Interrupt Manager, `adi_int_Term()`. Many applications operate in an endless loop and never call the termination function.

When called, the termination function unhooks all interrupt handlers, masking off (disabling) all interrupts that the Interrupt Manager was controlling. After calling the termination function, any memory provided to the initialization function may be re-used by the application. No other Interrupt Manager functions can be called after termination. If Interrupt

Manager services are required after the termination function is called, the application must re-initialize Interrupt Manager services by calling the `adi_pwr_Init` function.

Core Event Controller Functions

Only two functions are necessary to provide complete control over the Core Event Controller.

adi_int_CECHook()

The `adi_int_CECHook()` function is used to hook an interrupt handler into the handler chain for an IVG level. When called, the application passes in the IVG number that is to be handled, the address of the handler function, a parameter that the Interrupt Manager automatically passes back to the interrupt handler when the interrupt handler is invoked, and a flag indicating whether or not interrupt nesting should be enabled for this IVG level.

The handler function itself is a simple C-callable function that conforms to the `ADI_INT_HANDLER_FN typedef`. The interrupt handler is not an Interrupt Service Routine (ISR) but a standard C-callable function. When the IVG level triggers it, the Interrupt Manager calls the interrupt handler to process the event. The Interrupt Manager passes the client argument that was passed to the Interrupt Manager via the `adi_int_CECHook()` function to the interrupt handler. The interrupt handler takes whatever action is necessary to process the interrupt, then returns with either the `ADI_INT_RESULT_PROCESSED` or `ADI_INT_RESULT_NOT_PROCESSED` return code.

Interrupt handlers should be written in such a way so as to interrogate the system quickly to determine if the event that triggered the interrupt should be processed by the interrupt handler. If the event that caused the interrupt is not the event the interrupt handler was expecting, the interrupt handler should immediately return with the `ADI_INT_RESULT_NOT_`

PROCESSED return code. If the interrupt handler returns the ADI_INT_RESULT_NOT_PROCESSED return code, then the Interrupt Manager automatically invokes the next interrupt handler, if any, that is hooked into the same IVG level. If the event that caused the interrupt is expected by the interrupt handler, the interrupt handler performs whatever processing is necessary and then returns the ADI_INT_RESULT_PROCESSED return code. If an interrupt handler returns the ADI_INT_RESULT_PROCESSED return code, the Interrupt Manager does not invoke any other interrupt handlers that may be hooked into that IVG chain.

The nesting flag parameter is of significance only when the first interrupt handler is hooked into an IVG chain. The first interrupt handler that hooks into an IVG chain is called the primary handler. Any additional handlers that are hooked into that same IVG chain are called secondary handlers. When the primary handler is hooked into the chain, the Interrupt Manager loads an ISR into the appropriate entry of the Event Vector Table. If the nesting flag is set, the ISR that the Interrupt Manager loads is one that supports interrupt nesting. If the nesting flag is clear, then the ISR that the Interrupt Manager loads is one that does not support interrupt nesting. When secondary handlers are hooked into an IVG chain, the nesting flag is ignored.

Lastly, the adi_int_CECHook() function unmasks the appropriate bit in the CEC's IMASK register, thereby enabling the interrupt to be processed.

In most applications, users take great care to optimize the processing that occurs for the highest frequency and highest urgency interrupts. Typically, the highest frequency or highest urgency interrupts are assigned their own IVG level, while less frequent or lower urgency interrupts, such as error processing, are ganged together on a single IVG level.

The Interrupt Manager continues that thinking and has been optimized to allow extremely efficient processing for primary interrupt handlers. Though still efficient, secondary handlers are called after the primary handler. Secondary handlers are hooked into the IVG chain in a stacked or Last In First Out (LIFO) fashion. This means that when an event is triggered, after calling the primary handler (and assuming the primary handler did not return the `ADI_INT_RESULT_PROCESSED` return code), the Interrupt Manager calls the last secondary handler that was hooked, followed by the second to last installed handler, and so on.

To ensure optimal performance, the application developer should manage which interrupt handlers are hooked as primaries and which are hooked as secondary handlers.

adi_int_CECUnhook()

The `adi_int_CECUnhook()` function is used to unhook an interrupt handler from the interrupt handler chain for a particular IVG level. When called, the application passes in the IVG number and the address of the interrupt handler function that is to be unhooked from the chain.

The function removes the interrupt handler from the chain of handlers for the given IVG level. If the primary handler is being removed, the last secondary handler that was hooked becomes the new primary handler. If after removing the given interrupt handler there are no interrupt handlers left in the IVG chain, the `adi_int_CECUnhook()` function masks the appropriate bit in the CEC's `IMASK` register, thereby disabling the interrupt.

Interrupt Handlers

Since the interrupt handlers registered with the Interrupt Manager are invoked from within the built-in IVG interrupt service routine, and since there may be several interrupts pending for the same IVG level, individual interrupt handlers must not invoke the `RTI` instruction on completion. Instead, they should return using the RTS return function. Interrupt handlers are in fact nothing more than typical C-callable subroutines.

Each peripheral interrupt handler must, therefore, conform to the following template,

```
ADI_INT_HANDLER(mjk_SPORT_RX_handler)
{
 ...    ...          // user code
}
```

where the `ADI_INT_HANDLER` macro is defined as

```
#define ADI_INT_HANDLER(NAME) \
        void (*NAME)(void *ClientArg)
```

System Interrupt Controller Functions

The following functions are provided to give the application complete control over the System Interrupt Controller:

- `adi_int_SICEnable()` - Enables peripheral interrupts to be passed to the CEC.
- `adi_int_SICDisable()` - Disables peripheral interrupts from being passed to the CEC.
- `adi_int_SICSetIVG()` - Sets the IVG level to which a peripheral interrupt is mapped.
- `adi_int_SICGetIVG()` - Detects the IVG level to which a peripheral interrupt is mapped.
- `adi_int_SICWakeup()` - Establishes whether or not a peripheral interrupt wakes up the processor from an idled state.
- `adi_int_SICInterruptAsserted()` - Detects whether or not a peripheral interrupt is asserted.

All SIC functions take as a parameter an enumeration value that uniquely identifies a peripheral interrupt. The enumeration `ADI_INT_PERIPHERAL_ID` identifies each possible peripheral interrupt source for the processor. This enumeration is defined in the `adi_int.h` file. Refer to the file for the complete list of values for each supported Blackfin processor.

adi_int_SICDisable()

The `adi_int_SICDisable()` function is used to disable a peripheral interrupt from being passed to the Core Event Controller. When called, this function programs the appropriate SIC IMASK register to disable the given peripheral interrupt.

adi_int_SICEnable()

The `adi_int_SICEnable()` function is used to enable a peripheral interrupt to be passed to the Core Event Controller. When called, this function programs the appropriate SIC IMASK register to enable the given peripheral interrupt.

adi_int_SICGetIVG()

The `adi_int_SICGetIVG()` function is used to detect the IVG level to which a peripheral interrupt is mapped. In addition to the `ADI_INT_PERIPHERAL_ID` parameter, this function is passed pointer-to-memory location information. The function interrogates the proper field of the appropriate SIC Interrupt Assignment register and stores the IVG level (0 to 15) to which the given peripheral interrupt is mapped into the memory location.

adi_int_SICInterruptAsserted()

The `adi_int_SICInterruptAsserted()` function is used to detect whether or not the given peripheral interrupt is asserted. Though it can be called at any time, it is intended that this function is called immediately by the application's interrupt handlers to determine if a given peripheral interrupt is being asserted, allowing the interrupt handler to determine if its peripheral is asserting the interrupt.

Instead of using the usual `ADI_INT_RESULT_SUCCESS` return code, this function returns the `ADI_INT_RESULT_ASSERTED` or `ADI_INT_RESULT_NOT_ASSERTED` return code upon a successful completion. If errors are detected with the calling parameters, this function may return a different error code.

adi_int_SICSetIVG()

The `adi_int_SICSetIVG()` function is used to set the IVG level to which a peripheral interrupt is mapped. Upon powerup, the Blackfin processor invokes a default mapping of peripheral interrupts to IVG level. This function alters that mapping. In addition to the `ADI_INT_PERIPHERAL_ID` parameter, this function is passed the IVG level (0 to 15) to which the peripheral interrupt should be mapped. The function modifies the proper field within the appropriate SIC Interrupt Assignment register to the new mapping.

adi_int_SICWakeup()

The `adi_int_SICWakeup()` function is used to enable or disable a peripheral interrupt from waking up the core when the interrupt trigger and the core are in an idled state. In addition to the `ADI_INT_PERIPHERAL_ID` parameter, this function is passed a TRUE/FALSE flag. If the flag is TRUE, the SIC Interrupt Wakeup register is programmed such that the given peripheral interrupt wakes up the core when the interrupt is triggered. If the flag is FALSE, the SIC Interrupt Wakeup register is programmed such that the given peripheral interrupt does not wake up the core when the interrupt is triggered.

Note that this function does not enable or disable interrupt processing. Also note that it is possible to configure the SIC so that a peripheral interrupt wakes up the core from an idled state but does not process the interrupt. This may or may not be the intended operation.

Protecting Critical Regions

In embedded systems it is often necessary to protect a critical region of code while it is being executed. This is often necessary while one logical programming sequence is updating or modifying a piece of data. In these cases, another logical programming sequence, such as interrupt processing in one system, or different thread in an RTOS based system, is prevented from interfering while the critical data is being updated.

To that end, the Interrupt Manager provides two functions that can be used to bracket a critical region of code. These functions are `adi_int_EnterCriticalRegion()` and `adi_int_ExitCriticalRegion()`. The application calls the `adi_int_EnterCriticalRegion()` function at the beginning of the critical section of code, and then calls the `adi_int_ExitCriticalRegion()` function at the end of the critical section. These functions should always be used in pairs.

The actual implementation of these functions varies from operating environment to operating environment. For example in a standalone system, in systems without any RTOS, what actually happens in these functions may be different than the version of these functions for an RTOS based system. The principle and usage however, are always the same regardless of implementation. In this way, application code always operates the same way, and does not have to change regardless of the operating environment.

The adi_int_EnterCriticalRegion() function is passed an argument of type void * and returns an argument of type void *. The value that is returned from the adi_int_EnterCriticalRegion() function must always be passed to the corresponding adi_int_ExitCriticalRegion() function. For example, examine the following code sequence:

```
…
Value = adi_int_EnterCriticalRegion(pArg);
…            // critical section of code
adi_int_ExitCriticalRegion(Value);
…
```

The value that is returned from the adi_int_EnterCriticalRegion() function must be passed to the corresponding adi_int_ExitCriticalRegion() function. While nesting of calls to these functions is allowed, the application developer minimizes the use of these functions to only those critical sections of code, and realize that in all likelihood the processor is being placed in some altered state. This could affect the performance of the system, while in the critical regions. For example, it could be that interrupts are disabled in the critical region. The application developer typically does not want to have interrupts disabled for long periods of time. These functions should be used sparingly and judiciously.

Nesting of these calls is allowed. For example consider the following code sequence that makes a call to the function Foo() while in a critical section of code. The function Foo() also has a critical region of code.

```
…
Value = adi_int_EnterCriticalRegion(pArg);
…            // critical section of code
Foo();       // call to Foo()
adi_int_ExitCriticalRegion(Value);
…

void Foo(void) {
void *Value;
…
Value = adi_int_EnterCriticalRegion(pArg);
…             // critical section of code
adi_int_ExitCriticalRegion(Value);
…
}
```

This practice is allowed, however the application developer is cautioned that overuse of these functions can affect system performance.

The `pArg` value that is passed into the `adi_int_EnterCriticalRegion()` function is dependent upon the actual implementation for the given operating environment. In some operating environments the value is not used and can be NULL. The user should check the source file for the specific operating environment, `adi_int_xxx.c` in the `blackfin/lib/src/services` directory where `xxx` is the operating environment, for more information on the `pArg` parameter.

ⓘ All system services and device drivers use these functions exclusively to protect critical regions of code. Application software should also use these functions exclusively to protect critical regions of code within the application.

Modifying IMASK

Though applications rarely need to have the processor's IMASK register value modified, the Interrupt Manager itself modifies the IMASK register value to control the CEC properly. In some RTOS-based operating environments, the RTOS tightly controls the IMASK register and provides functions that allow the manipulation of IMASK.

In order to ensure compatibility across all operating environments, the Interrupt Manager provides functions that allow bits within the IMASK register to be set or cleared. Depending on the operating environment, these function may modify the IMASK value directly, or use the RTOS provided IMASK manipulation functions. Regardless of how the IMASK value is changed, the Interrupt Manager API provides a uniform and consistent mechanism for this.

Two operating environment implementation dependent functions are provided to set and clear bits in the `IMASK` register. These functions are `adi_int_SetIMASKBits()` and `adi_int_ClearIMASKBits`. These functions take as a parameter a value that corresponds to the `IMASK` register of the processor being targeted. When the `adi_int_SetIMASKBits()` function is called, the function sets to 1 those bits in the `IMASK` register that have a one in the corresponding bit position of the value passed in. When the `adi_int_ClearIMASKBits()` function is called, the function clears those bits (to 0) in the `IMASK` register that have a 1 in the corresponding bit position of the value passed in.

Consider the following example code. Assume that IMASK is a 32-bit value and contains 0x00000000 upon entry into the code:

```
…
…            // IMASK = 0x00000000
ReturnCode = adi_int_SetIMASKBits(0x00000003);
…            // IMASK now equals 0x00000003
ReturnCode = adi_int_ClearIMASKBits(0x00000001);
…            // IMASK now equals 0x00000002
ReturnCode = adi_int_ClearIMASKBits(0x00000002);
…            // IMASK now equals 0x00000000
```

While it is very unlikely that the application ever needs to control individual `IMASK` bit values, the Interrupt Manager uses these functions to control the CEC.

Examples

Examples demonstrating use of the Interrupt Manager can be found in the `blackfin/EZ-Kits` subdirectories.

File Structure

The API for the Interrupt Manager is defined in the file `adi_int.h`. This file is located in the `blackfin/include/services` subdirectory and is automatically included by the `services.h` file in that same directory. Only the `services.h` file should be included in the application code.

Applications should link with one and only one of the System Services library files. These files are located in the `blackfin/lib` directory. See the appropriate section in the "Introduction" on page 6-1 in DMA Manager for more information on selecting the proper library file.

For convenience, all source code for the Interrupt Manager is located in the `blackfin/lib/src/services` directory. All operating environment dependent code is located in the file `adi_int_xxx.c` where `xxx` is the operating environment being targeted. These files should never be linked into an application, as the appropriate System Services Library file contains all required object code.

Interrupt Manager API Reference

This section provides descriptions of the Interrupt Manager module's Application Programming Interface (API) functions.

Notation Conventions

The reference pages for the API functions use the following format:

Name and purpose of the function

Description – Function specification

Prototype – Required header file and functional prototype

Arguments – Description of function arguments

Return Value – Description of function return values

adi_int_Init

Description

This function sets aside and initializes memory for the Interrupt Manager. It also initializes other tables and vectors within the Interrupt Manager. This function should only be called once per core. Separate memory areas should be assigned for each core.

Prototype

```
ADI_INT_RESULT adi_int_CECInit(
        void            *pMemory,
        const size_t    MemorySize,
        u32             *pMaxEntries,
        void            *pEnterCriticalArg
);
```

Arguments

`*pMemory`	This is the pointer to an area of memory to be used by the Interrupt Manager.
`MemorySize`	This is the size, in bytes, of memory being supplied for the Interrupt Manager.
`*pMaxEntries`	On return, this argument contains the number of secondary handler entries that the Interrupt Manager can support given the memory supplied.
`*pEnterCriticalArg`	Parameter passed to the `adi_int_EnterCriticalRegion`.

Return Value

Return values include:

`ADI_INT_RESULT_SUCCESS`	Successfully initialized

adi_int_Terminate

Description

This function closes down the Interrupt Manager. All memory used by the Interrupt Manager is freed up, all handlers are unhooked, and all Interrupt Vector Groups that were enabled and controlled by the Interrupt Manager are disabled.

Note that the `adi_int_Terminate` function does not alter the System Interrupt Controller settings. Should changes to the SIC be required, the application should make the appropriate calls into the relevant SIC control functions before calling `adi_int_Terminate()`.

Prototype

```
ADI_INT_RESULT adi_int_Terminate(void);
```

Arguments

none

Return Value

The function returns `ADI_INT_RESULT_SUCCESS` if successful. Any other value indicates an error.

adi_int_CECHook

Description

This function instructs the Interrupt Manager to hook (insert) the given interrupt handler into the interrupt handler chain for the given IVG.

On a return from this call, the core event controller is programmed such that the given IVG is unmasked (enabled) and the system is properly configured to service the interrupt via the Interrupt Manager's built-in ISRs. The ISRs then invoke the interrupt handler supplied by the caller. Depending on the state of the `NestingFlag` parameter, the Interrupt Manager installs its built-in interrupt service routine with interrupt nesting, either enabled or disabled.

On the first call for a given IVG level, the Interrupt Manager registers its built-in IVG interrupt service routine against that level and establishes the supplied interrupt handler as the primary interrupt handler for the given IVG level. Subsequent calls to `adi_int_CECHook` for the same IVG level create a chain of secondary interrupt handlers for the IVG level. When the interrupt for the IVG level is triggered, the primary interrupt handler is first called, and then if present, each secondary interrupt handler is subsequently called.

The `ClientArg` parameter provided in the `adi_int_CECHook` function is passed to the interrupt handler as an argument when the interrupt handler is called in response to interrupt generation.

Prototype

```
ADI_INT_RESULT adi_int_CECHook(
        u32                 IVG,
        ADI_INT_HANDLER_FN  Handler,
        void                *ClientArg,
        u32                 NestingFlag
);
```

Arguments

`IVG`	This is the interrupt vector group number that is being addressed.
`Handler`	The client's interrupt handler to be inserted into the chain for the given IVG.
`ClientArg`	A `void *` value that is passed to the interrupt handler.
`NestingFlag`	This is the argument that selects whether nesting of interrupts is allowed or disallowed for the IVG (TRUE/FALSE).

Return Value

Return values include:

`ADI_INT_RESULT_SUCCESS`	The interrupt handler was successfully hooked into the chain.
`ADI_INT_RESULT_NO_MEMORY`	Insufficient memory is available to insert the handler into the chain.
`ADI_INT_RESULT_INVALID_IVG`	The IVG level is invalid.

adi_int_CECUnhook

Description

This function instructs the Interrupt Manager to unhook (remove) the given interrupt handler from the interrupt handler chain for the given IVG.

If the given interrupt handler is the only interrupt handler in the chain, the CEC is programmed to disable (mask) the given IVG and the Interrupt Manager built-in interrupt service routine is removed from the IVG entry within Event Vector Table.

If the chain for the given IVG contains multiple interrupt handlers, the given interrupt handler is simply purged from the chain. If the primary interrupt handler is removed and there are secondary interrupt handlers in the chain are present, one of the secondary interrupt handlers becomes the primary interrupt handler.

Prototype

```
ADI_INT_RESULT adi_int_CECUnhook(
        u32                IVG,
        ADI_INT_HANDLER_FN  Handler,
);
```

Arguments

`IVG`	The interrupt vector group number that is being addressed.
`Handler`	The client's interrupt handler to be removed from the chain for the given IVG.

Return Value

Return values include:

`ADI_INT_RESULT_SUCCESS`	The interrupt handler was successfully unhooked from the chain.
`ADI_INT_RESULT_INVALID_IVG`	The IVG level is invalid.

adi_int_ClearIMaskBits

Description

This function is used by the Interrupt Manager to clear bits in the IMASK register. Though it can also be called by the application, the application should not attempt to modify bits in the IMASK register that represent interrupt vector groups that are under the control of the Interrupt Manager.

The implementation of this function depends upon the operating environment. In the standalone version of the service, this function detects if the processor is within a protected region of code (see the adi_int_EnterCriticalRegion and adi_int_ExitCriticalRegion functions). If it is, the saved value of IMASK is updated accordingly and the current "live" IMASK value is left unchanged. When the outermost adi_int_ExitCriticalRegion function is called, the saved IMASK value with the new bit settings, is restored. If upon entering this function, the processor is not within a protected region of code, the "live" IMASK register is updated accordingly.

Information on the implementation details for this function in other operating environments can be found in the file adi_int_xxx.h, located in the blackfin/include/services/ directory, where xxx is the operating environment.

Note that regardless of the implementation details, the API is consistent from environment to operating environment. Changes to application software are not required when code is moved to a different operating environment.

Prototype

```
void adi_int_ClearIMASKBits(
        ADI_INT_IMASK      BitsToClear
);
```

Arguments

`BitsToClear`	Replica of the IMASK register containing bits that are to be cleared in the real IMASK register. A bit with a value of '1' clears the corresponding bit in the IMASK register. A bit with the value of '0' leaves the corresponding bit in the IMASK register unchanged.

Return Value

None

adi_int_EnterCriticalRegion

Description

This function creates a condition that protects a critical region of code. The companion function, `adi_int_ExitCriticalRegion`, removes the condition. These functions should be used to bracket a section of code that needs protection from other processing. These functions should be used in pairs, sparingly and only when critical regions of code need protecting.

The return value from this function should be passed to the corresponding `adi_int_ExitCriticalRegion` function.

The actual condition that is created is dependent upon the operating environment. In the standalone version of the service, this function effectively disables interrupts, saving the current value of `IMASK` to a temporary location. The `adi_int_ExitCriticalRegion` function restores the original `IMASK` value. These functions employ a usage counter so that they can be nested. When nested, the `IMASK` value is altered only at the outermost levels. In the standalone version, the `pArg` parameter to the `adi_int_EnterCriticalRegion` is meaningless.

Information on the implementation details for this function in other operating environments can be found in the file `adi_int_xxx.h`, located in the `blackfin/include/services/` directory, where `xxx` is the operating environment.

Note that regardless of the implementation details, the API is consistent from environment to operating environment and from processor to processor. Application software does not need to change when moving to a different operating environment or moving from one Blackfin derivative to another.

Prototype

```
void *adi_int_EnterCriticalRegion(
         void         *pArg
);
```

Arguments

`pArg`	Implementation dependent. Refer to the `adi_int_xxx.h` file for details on this parameter for the `xxx` environment.

Return Value

The return value from this function should always be passed to the corresponding `adi_int_ExitCriticalRegion` function.

adi_int_ExitCriticalRegion

Description

This function removes the condition that was established by the adi_int_EnterCriticalRegion to protect a critical region of code. These functions should be used to bracket a section of code that needs protection from other processing. These functions should be used sparingly and only when critical regions of code need protecting.

The pArg parameter that is passed to this function should always be the return value from the corresponding adi_int_EnterCriticalRegion function.

See the adi_int_EnterCriticalRegion function for more information.

Prototype

```
void adi_int_ExitCriticalRegion(
          void      *pArg
);
```

Arguments

pArg	The return value from the corresponding adi_int_EnterCriticalRegion() function call.

Return Value

None

adi_int_SICDisable

Description

This function configures the System Interrupt Controller to disable the given interrupt and prevent it from being passed to the Core Event Controller.

The `adi_int_SICDisable` function simply programs the System Interrupt Mask register to mask interrupts from the given peripheral, thereby preventing them from being passed to the Core Event Controller.

Prototype

```
ADI_INT_RESULT adi_int_SICDisable(
        const ADI_INT_PERIPHERAL_ID   PeripheralID
);
```

Arguments

`PeripheralID`	This is the `ADI_INT_PERIPHERAL_ID` enumeration value that identifies an interrupt source.

Return Value

`ADI_INT_RESULT_SUCCESS`	The System Interrupt Controller has been successfully configured.
`ADI_INT_RESULT_INVALID_PERIPHERALID`	The peripheral ID specified is invalid.

adi_int_SICEnable

Description

This function configures the System Interrupt Controller to enable the given interrupt and allow it to be passed to the Core Event Controller.

The `adi_int_SICEnable` function simply programs the System Interrupt Mask register to allow interrupts from the given peripheral to be passed to the Core Event Controller.

Prototype

```
ADI_INT_RESULT adi_int_SICEnable(
        const ADI_INT_PERIPHERAL_ID    PeripheralID,
);
```

Arguments

`PeripheralID`	This is the `ADI_INT_PERIPHERAL_ED` enumeration value that identifies a peripheral interrupt source.

Return Value

Return values include:

`ADI_INT_RESULT_SUCCESS`	The System Interrupt Controller has been successfully configured.
`ADI_INT_RESULT_INVALID_PERIPHERAL_ID`	The peripheral ID specified is invalid.

adi_int_SICGetIVG

Description

This function detects the mapping of a peripheral interrupt source to an IVG level. When called, this function reads the appropriate System Interrupt Assignment register(s) of the given peripheral and stores the IVG level to which the peripheral is mapped into the location provided by the application. This function does not modify any parameters of the interrupt controller.

Prototype

```
ADI_INT_RESULT adi_int_SICSetIVG(
        const ADI_INT_PERIPHERAL_ID   PeripheralID,
        u32                           *pIVG
);
```

Arguments

PeripheralID	The ADI_INT_PERIPHERAL_ID enumeration value that identifies a peripheral interrupt source
*pIVG	The pointer to an unsigned 32-bit memory location into which the function writes the IVG level to which the given peripheral is mapped.

Return Value

The function returns ADI_INT_RESULT_SUCCESS if successful. Other possible return values include:

ADI_INT_RESULT_INVALID_ PERIPHERAL_ID	The peripheral ID specified is invalid.
ADI_INT_RESULT_INVALID_IVG	The interrupt vector group level is invalid.

adi_int_SICInterruptAsserted

Description

This function determines if a given peripheral interrupt source is asserting an interrupt. This function is typically called in an application's interrupt handler to determine if the peripheral in question is asserting an interrupt. This function does not modify any parameters of the interrupt controller but simply interrogates the appropriate interrupt status register(s).

Prototype

```
ADI_INT_RESULT adi_int_SICInterruptAsserted(
        const ADI_INT_PERIPHERAL_ID   PeripheralID
);
```

Arguments

`PeripheralID`	The `ADI_INT_PERIPHERAL_ID` enumeration value that identifies a peripheral interrupt source.

Return Value

The function returns one of the following values:

`ADI_INT_RESULT_INVALID_ PERIPHERAL_ID`	The peripheral ID specified is invalid.
`ADI_INT_RESULT_ASSERTED`	The specified peripheral is asserting an interrupt.
`ADI_INT_RESULT_NOT_ASSERTED`	The specified peripheral is not asserting an interrupt.

adi_int_SICSetIVG

Description

This function sets the mapping of a peripheral interrupt source to an IVG level. When called, this function modifies the appropriate System Interrupt Assignment register(s) of the given peripheral to the specified IVG level. This function does not enable or disable interrupts.

Prototype

```
ADI_INT_RESULT adi_int_SICSetIVG(
        const ADI_INT_PERIPHERAL_ID   PeripheralID,
        const u32                     IVG
);
```

Arguments

`PeripheralID`	The `ADI_INT_PERIPHERAL_ID` enumeration value that identifies a peripheral interrupt source
`IVG`	The interrupt vector group that the peripheral to which the peripheral is being assigned.

Return Value

The function returns `ADI_INT_RESULT_SUCCESS`, if successful. Other possible return values include:

`ADI_INT_RESULT_INVALID_PERIPHERAL_ID`	The peripheral ID specified is invalid.
`ADI_INT_RESULT_INVALID_IVG`	The interrupt vector group level is invalid.

adi_int_SetIMaskBits

Description

This function is used by the Interrupt Manager to set bits in the IMASK register. Though it can also be called by the application, the application should not attempt to modify bits in the IMASK register that represent interrupt vector groups that are under the control of the Interrupt Manager.

The implementation of this function is dependent upon the operating environment. In the standalone version of the service, this function detects if the processor is within a protected region of code (see the adi_int_EnterCriticalRegion and adi_int_ExitCriticalRegion functions). If it is, the saved value of IMASK is updated accordingly and the current "live" IMASK value is left unchanged. When the outermost adi_int_ExitCriticalRegion function is called, the saved IMASK value, with the new bit settings, is restored. If upon entering this function the processor is not within a protected region of code, the "live" IMASK register is updated accordingly.

Information on the implementation details for this function in other operating environments can be found in the file adi_int_xxx.h, located in the blackfin/include/services/ directory, where xxx is the operating environment.

Note that regardless of the implementation details, the API is consistent from environment to operating environment. Application software does not have to change when moving to a different operating environment.

Prototype

```
void adi_int_SetIMASKBits(
        ADI_INT_IMASK      BitsToSet
);
```

Arguments

`BitsToSet`	Replica of the IMASK register containing bits that are to be set in the real IMASK register. A bit with a value of '1' sets the corresponding bit in the IMASK register. A bit with the value of '0' leaves the corresponding bit in the IMASK register unchanged.

Return Value

None

adi_int_SICWakeup

Description

This function configures the System Interrupt Controller Wakeup register to enable or disable the given peripheral interrupt from waking up the core processor.

The `adi_int_SICWakeup` function simply programs the System Interrupt Controller Wakeup register accordingly. The actual servicing of interrupts is not affected by this function.

Prototype

```
ADI_INT_RESULT adi_int_SICWakeup(
        const ADI_INT_PERIPHERAL_ID   PeripheralID,
        u32                           WakeupFlag
);
```

Arguments

`PeripheralID`	This is the `ADI_INT_PERIPHERAL_ID` enumeration value that identifies a peripheral interrupt source.
`WakeupFlag`	Enables/disables waking up the core(s) upon triggering of the peripheral interrupt (TRUE/FALSE).

Return Value

Return values include:

`ADI_INT_RESULT_SUCCESS`	The System Interrupt Controller has been successfully configured.
`ADI_INT_RESULT_INVALID_PERIPHERAL_ID`	The peripheral ID specified is invalid.

3 POWER MANAGEMENT MODULE

This chapter describes the Power Management (PM) module that supports Dynamic Power Management of Blackfin processors.

This chapter contains:

Introduction

The Power Management (PM) module provides access to all aspects of Dynamic Power Management:

- Dynamic switching from one operating mode to another: Full-On, Active, Sleep, Deep Sleep and Hibernate.
- Dynamic setting of voltage levels and clock frequencies to ensure that an application can be tuned to achieve the best performance while minimizing power consumption.
- When coupled with the EBIU module*, it enables the SDRAM settings to be adjusted upon changes to the system clock to ensure that the best performance is obtained for the complete system.

The module supports two strategies for setting the core and system clock frequencies:

- For a given voltage level, the core clock (CCLK) is set to the highest available frequency. The system clock (SCLK) is set accordingly.
- For a given combination of core and system clock frequencies, the valid values nearest to the chosen ones are used and the voltage level of the processor adjusted accordingly.

In both cases validity checks are performed at all stages, making it impossible to stall or harm the processor.

"PM Module Operation – Getting Started" describes the basic operating stages required to use the Power Management module.

* See Chapter 3, "External Bus Interface Unit Module" for more information.

The Power Management Module uses an unambiguous naming convention to safeguard against conflicts with other software libraries provided by Analog Devices, Inc. or other companies. To this end, all enumeration values and `typedefs` use the `ADI_PWR_` prefix, while functions and global variables use the lower case, `adi_pwr_` equivalent.

Two versions of the library exist for each processor. These correspond to the debug and release configurations in VisualDSP++ Release 4.0. In addition to the usual defaults for the debug configuration, the API functions perform checks on the arguments passed and report appropriate error codes, as required. In the release version of the library, most functions return one of two result codes: `ADI_PWR_RESULT_SUCCESS` on successful completion, or `ADI_PWR_RESULT_CALL_IGNORED` if the PM module has not been initialized prior to the function call.

PM Module Operation – Getting Started

A following example illustrates how to use the PM module to configure a 600Mz ADSP-BF533 processor on an EZ-KIT Lite board to run at the requested core and system clock frequencies or to minimize power consumption by pegging the voltage level at 0.95 V.

Step 1:
Initialize the module by setting the parameters for the hardware configuration used. In the following example it is assumed that the ADSP-BF533 EZ-KIT Lite (Rev 1.3) is to be configured. The simplest way is to specify the EZ-KIT board as follows:

```
ADI_PWR_COMMAND_PAIR ezkit_pwr[] = {
    // ADSP-BF533 EZ-KIT LITE REV 1.3
    { ADI_PWR_CMD_SET_EZKIT, ADI_PWR_EZKIT_BF533_600MHz },
    { ADI_PWR_CMD_END, 0 }
};
adi_pwr_Init(ezkit_pwr);
```

To illustrate what is required for non EZ-Kit boards, the above command table is included an abbreviated form of the following code:

```
ADI_PWR_COMMAND_PAIR ezkit_pwr[] = {
                    /* 600Mhz ADSP-BF533 variant  *
    { ADI_PWR_CMD_SET_PROC_VARIANT,(void*)ADI_PWR_PROC_
BF533SKBC600 },
                    /* in MBGA packaging, as on all EZ-KITS */
    { ADI_PWR_CMD_SET_PACKAGE,    (void*)ADI_PWR_PACKAGE_MBGA },
                    /* External Voltage supplied to the     */
                    /*voltage regulator is 3.3V             */
    { ADI_PWR_CMD_SET_VDDEXT, (void*)ADI_PWR_VDDEXT_330 },
                    /* The CLKIN frequency                  */
(ADI_PWR_CLKIN_EZKIT=27Mhz */
    { ADI_PWR_CMD_SET_CLKIN,          (void*)ADI_PWR_CLKIN_
EZKIT_REV_1_5 },
                    /* command to terminate the table       */
    { ADI_PWR_CMD_END,        0 }
};
adi_pwr_Init(ezkit_pwr);
```

Step 2:
If used in conjunction with the EBIU controller to adjust SDRAM settings, the EBIU module is initialized (for EZ-KIT) with the following call:

```
ADI_EBIU_COMMAND_PAIR ezkit_ebiu[] = {
    { ADI_EBIU_CMD_SET_EZKIT,(void*)ADI_EBIU_EZKIT_BF533 },
    { ADI_EBIU_CMD_END, 0 }
};
adi_ebiu_Init(
        ezkit_ebiu,    // default is EZ-KIT
        FALSE         // Do not adjust refresh settings
);
```

Step 3:
Decide on which power management strategy to implement. For example, the following code segments demonstrate how to configure the PM module for optimal speed or optimal power consumption.

Optimal Speed

The following statement requests that the PM module set the core and system clock frequencies to the maximum values possible:

```
adi_pwr_SetFreq(
        0,               // Core clock frequency (MHz)
        0,               // System clock frequency (MHz)
        ADI_PWR_DF_ON    // Do not adjust the PLL input divider
);
```

Optimal Power Consumption

The following statement requests that the PM module set the core and system clock frequencies to the maximum that can be sustained at a voltage level of 0.85 V:

```
adi_pwr_SetMaxFreqForVolt(ADI_PWR_VLEV_085);
```

Power Management API Reference

This section provides descriptions of the PM module's Application Programming Interface (API) functions.

Notation Conventions

The reference pages for the API functions use the following format:

Name and purpose of the function

Description – Function specification

Prototype – Required header file and functional prototype

Arguments – Description of function arguments

Return Value – Description of function return values

adi_pwr_AdjustFreq

Description

This function allows the core and system clocks to be modified by specifying the core and system clock divider ratios, `CSEL` and `SSEL`, in the `PLL_DIV` register. The processor is not idled.

Prototype

```
ADI_PWR_RESULT adi_pwr_AdjustFreq(
          const ADI_PWR_CSEL csel,
          const ADI_PWR_SSEL ssel
);
```

Arguments

`csel`	An `ADI_PWR_CSEL` value specifies how the Voltage Core Oscillator (VCO) frequency is to be divided to obtain a new Core Clock frequency (see "ADI_PWR_CSEL" on page 3-35). The divider value cannot exceed the `ssel` value.
`ssel`	An `ADI_PWR_SSEL` value specifies how the VCO frequency is to be divided to obtain a new System Clock frequency (see "ADI_PWR_SSEL" on page 3-44).

Return Value

In the debug variant of the library, the function `adi_pwr_AdjustSpeed` returns one of the following result codes. Otherwise the function returns `ADI_PWR_RESULT_SUCCESS`.

`ADI_PWR_RESULT_SUCCESS`	This process completed successfully.
`ADI_PWR_RESULT_CALL_IGNORED`	The PM module has not been initialized.
`ADI_PWR_RESULT_INVALID_CSEL`	An invalid value for `CSEL` has been specified.

`ADI_PWR_RESULT_INVALID_SSEL`	An invalid value for `SSEL` has been specified.
`ADI_PWR_INVALID_CSEL_SSEL_COMBINATION`	The core clock divider is greater that the System clock divider value, or both `ADI_PWR_CSEL_NONE` and `ADI_PWR_SSEL_NONE` are specified.

adi_pwr_Control

Description

This function enables the Dynamic Power Management registers to be configured or queried according to command-value pairs ("ADI_PWR_COMMAND_PAIR" on page 3-35), specified in one of three ways:

1. A single command-value pair is passed.

```
adi_pwr_Control(
         ADI_PWR_CMD_SET_INPUT_DELAY,
         (void*)ADI_PWR_INPUT_DELAY_ENABLE,
);
```

2. A single command-value pair structure is passed.

```
ADI_PWR_COMMAND_PAIR cmd = {
        ADI_PWR_CMD_SET_INPUT_DELAY,
        (void*)ADI_PWR_INPUT_DELAY_ENABLE,
};
adi_pwr_Control(ADI_PWR_CMD_PAIR,(void*)&cmd);
```

3. A table of `ADI_PWR_COMMAND_PAIR` structures is passed. The last entry in the table must be `ADI_PWR_CMD_END`.

```
ADI_PWR_COMMAND_PAIR table[] = {
    { ADI_PWR_CMD_SET_INPUT_DELAY, (void*)ADI_PWR_INPUT_DELAY_
ENABLE
    { ADI_PWR_CMD_SET_OUTPUT_DELAY, (void*)ADI_PWR_OUTPUT_DELAY_
ENABLE
    { ADI_PWR_CMD_END, 0}
};
adi_pwr_Control(
         ADI_PWR_CMD_TABLE,
         (void*)table
);
```

Refer to ADI_PWR_COMMAND on page 3-31 and "Public Data Types and Enumerations" on page 3-30 for the complete list of commands and associated values.

Prototype

```
ADI_PWR_RESULT adi_pwr_Control(
            ADI_PWR_COMMAND command,
            void *Value
);
```

Arguments

Command	An ADI_PWR_COMMAND enumeration value specifies the meaning of the associated value argument.
Value	This is the required value (see "ADI_PWR_COMMAND" on page 3-31).

Return Value

In debug mode the adi_pwr_Control function returns one of the following values. Otherwise, ADI_PWR_RESULT_SUCCESS is returned.

ADI_PWR_RESULT_SUCCESS	This function completed successfully.
ADI_PWR_RESULT_BAD_COMMAND	An invalid command has been specified.
ADI_PWR_RESULT_CALL_IGNORED	The PM module has not been initialized.
ADI_PWR_RESULT_INVALID_INPUT_DELAY	The input delay value is invalid.
ADI_PWR_RESULT_INVALID_OUTPUT_DELAY	The output delay value is invalid.
ADI_PWR_RESULT_INVALID_LOCKCNT	The PLL lock count value is invalid.

adi_pwr_GetConfigSize

Description

This function returns the number of bytes required to save the current configuration data. This value is also available via the `ADI_PWR_SIZEOF_CONFIG` macro.

The return value of `adi_pwr_GetConfigSize` as well the `ADI_PWR_SIZEOF_CONFIG` macro incorporate the size of the EBIU Module configuration, whether the latter is initialized or not.

Prototype

```
size_t adi_pwr_GetConfigSize(void);
```

Return Value

The size of the configuration structure.

adi_pwr_GetFreq

Description

This function returns the current values of the `CCLK`, `SCLK` and Voltage Core Oscillator (VCO) frequencies

Prototype

```
ADI_PWR_RESULT adi_pwr_GetFreq(
        u32 *fcclk,
        u32 *fsclk,
        u32 *fvco);
```

Arguments

`fcclk`	This is an address of location to store the current CCLK value (MHz).
`fsclk`	This is an address of location to store the current SCLK value (MHz).
`fvco`	This is an address of location to store the VCO frequency (MHz).

Return Value

In the debug variant of the library, the function `adi_pwr_GetFreq` returns one of the following result codes. Otherwise the function returns `ADI_PWR_RESULT_SUCCESS`.

`ADI_PWR_RESULT_SUCCESS`	This process completed successfully.
`ADI_PWR_RESULT_CALL_IGNORED`	The PM module has not been initialized.

adi_pwr_GetPowerMode

Description

This function returns the current power mode of the processor (only applicable for Full-on and Active modes).

Prototype

```
ADI_PWR_MODE adi_pwr_GetPowerMode(void);
```

Return Value

The current power mode as an `ADI_PWR_MODE` value.

adi_pwr_GetPowerSaving

Description

This function calculates the power saving value for the current PLL and voltage regulator settings, as per the data sheet formulae with the time ratio set to `unity`, and the nominal values as per the maximum possible (that is, `at VLEV=1.3V`).

Prototype

```
u16 adi_pwr_GetPowerSaving(void);
```

Return Value

The percentage power saving value.

adi_pwr_Init

Description

This function initializes the Power Management module. The following values are required to be set for successful initialization:

Processor variant	An `ADI_PWR_PROC_KIND` value describes the processor variant (see "ADI_PWR_PROC_KIND" on page 3-41).
Package kind	An `ADI_PWR_PACKAGE_KIND` value describes the packaging type of the processor (see"ADI_PWR_PACKAGE_KIND" on page 3-39).
Core voltage (V_{DDINT})	An `ADI_PWR_VLEV` value specifying the internal voltage, applied to the core by an external voltage regulator.The internal voltage regulator is bypassed. Its absence in the command table implies that the internal regulator is to be used. An external voltage regulator is required for the ADSP-BF533SKBC750 processor, as the internal voltage regulator cannot supply the 1.4V required for the processor to run at 750 MHz.
External voltage (V_{DDEXT})	An `ADI_PWR_VDDEXT` value specifies the external voltage supplied to the voltage regulator. This value, when coupled with the packaging, determines the maximum system clock (`SCLK`) frequency available.
`CLKIN`	Frequency of the external clock oscillator in MHz supplied to the processor. Macros are available for a range of input clocks. For example, the EZ-KIT Lite value is `ADI_PWR_CLKIN_EZKIT` (see "ADI_PWR_VDDEXT" on page 3-45).

These are communicated to the `adi_pwr_Init` function by passing a pointer to a table of command-value pairs, terminated with the `ADI_PWR_CMD_END` command.

For example, the following `ADI_PWR_COMMAND_PAIR` table gives the EZ-KIT Lite values:

```
ADI_PWR_COMMAND_PAIR ezkit_init[] = {
    { ADI_PWR_CMD_SET_PROC_VARIANT, ADI_PWR_PROC_BF533SKBC600 },
    { ADI_PWR_CMD_SET_PACKAGE,   ADI_PWR_PACKAGE_MBGA },
    { ADI_PWR_CMD_SET_VDDEXT,    ADI_PWR_VDDEXT_330 },
    { ADI_PWR_CMD_SET_CLKIN,     ADI_PWR_CLKIN_EZKIT },
    { ADI_PWR_CMD_END,           0 }
};
```

The following table lists valid command-value pairs.

`ADI_PWR_CMD_SET_CCLK_TABLE`	The address of a table containing `ADI_PWR_NUM_VLEVS` values of type `u16` detailing the maximum CCLK frequency for each `ADI_PWR_VLEV` value. These values will be used instead of the data sheet values.
`ADI_PWR_CMD_SET_EZKIT`	An `ADI_PWR_EZKIT` value to identify the EZ-KIT for which the power management is to be configured. This command establishes all the required values, as detailed above (see "ADI_PWR_EZKIT" on page 3-37]).
`ADI_PWR_CMD_SET_PROC_VARIANT`	An `ADI_PWR_PROC_KIND` value specifies the processor variant. (Mandatory) See "ADI_PWR_PROC_KIND" on page 3-41.
`ADI_PWR_CMD_SET_PACKAGE`	An `ADI_PWR_PACKAGE_KIND` value describes the packaging type of the processor. (Mandatory) See "ADI_PWR_PACKAGE_KIND" on page 3-39.
`ADI_PWR_CMD_SET_CLKIN`	A `u16` value specifies the external clock frequency, `CLKIN`, supplied to the processor. (Mandatory).
`ADI_PWR_CMD_SET_VDDINT`	An `ADI_PWR_VLEV` value specifies the Core Voltage Level. This should only be passed to `adi_pwr_Init` if an external voltage regulator is to be used, as its presence instructs the module to bypass the internal regulator (see "ADI_PWR_VLEV" on page 3-46).
`ADI_PWR_CMD_SET_VDDEXT`	An `ADI_PWR_VDDEXT` value specifies the external voltage level applied to the internal voltage regulator. (Mandatory) See "ADI_PWR_VDDEXT" on page 3-45.
`ADI_PWR_CMD_SET_IVG`	An `interrupt_kind` value (see `exception.h`) specifies the IVG level for the `PLL_WAKEUP` event.

ADI_PWR_CMD_SET_INPUT_DELAY	An ADI_PWR_INPUT_DELAY value specifies whether or not to add approximately 200ps of delay to the time when inputs are latched on the external memory interface (see "ADI_PWR_INPUT_DELAY" on page 3-37).
ADI_PWR_CMD_SET_OUTPUT_DELAY	An ADI_PWR_OUTPUT_DELAY value specifies whether or not to add approximately 200ps of delay to external memory output signals (see "ADI_PWR_OUTPUT_DELAY" on page 3-37).

The adi_pwr_Init function can only be called once. Subsequent calls to adi_pwr_Init are ignored with the ADI_PWR_RESULT_CALL_IGNORED result code returned.

Prototype

```
ADI_EBIU_RESULT adi_pwr_Init(
        ADI_PWR_COMMAND_PAIR *table
);
```

Arguments

ConfigData	The address of a table of command-value pairs as defined by "ADI_PWR_COMMAND_PAIR" on page 3-35 and "Public Data Types and Enumerations" on page 3-30. The last command in the table must be the ADI_EBIU_CMD_END command.

Return Value

In the debug variant of the library, `adi_pwr_Init` returns the following results codes. Otherwise either the value of `ADI_PWR_RESULT_SUCCESS` is returned, or the value of `ADI_PWR_RESULT_CALL_IGNORED` is returned if the PM module has already been initialized.

`ADI_PWR_RESULT_SUCCESS`	This function completed successfully.
`ADI_PWR_RESULT_BAD_COMMAND`	An invalid command has been specified.
`ADI_PWR_RESULT_CALL_IGNORED`	The module has already been initialized.
`ADI_PWR_RESULT_INVALID_VLEV`	An Invalid core voltage level has been specified.
`ADI_PWR_RESULT_INVALID_VDDEXT`	An Invalid external voltage level has been specified.
`ADI_PWR_RESULT_INVALID_PROCESSOR`	The processor type specified is invalid.
`ADI_PWR_RESULT_INVALID_IVG`	The IVG level supplied is invalid.
`ADI_PWR_RESULT_INVALID_INPUT_DELAY`	The input delay value is invalid.
`ADI_PWR_RESULT_INVALID_OUTPUT_DELAY`	The output delay value is invalid.
`ADI_PWR_RESULT_INVALID_LOCKCNT`	The PLL lock count value is invalid.
`ADI_PWR_RESULT_INVALID_EZKIT`	Invalid EZ-Kit type specified.

adi_pwr_LoadConfig

Description

This function restores the current configuration values from the memory location pointed to by the `hConfig` argument. The PLL controller and Voltage Regulator are reprogrammed. If the EBIU Module is initialized, its configuration is also loaded and the SDRAM Controller programmed.

Prototype

```
ADI_PWR_RESULT adi_pwr_LoadConfig(
        const ADI_PWR_CONFIG_HANDLE hConfig,
        const size_t szConfig
);
```

Arguments

`hConfig`	This is the address of the memory area from which the current configuration is to be restored.
`szConfig`	This is the number of bytes available at the given address. This value must be greater than or equal to the `adi_pwr_GetConfigSize()` return value.

Return Value

In the debug variant of the library, `adi_pwr_Init` returns the following results codes. Otherwise, the value of `ADI_PWR_RESULT_SUCCESS` is returned.

`ADI_PWR_RESULT_SUCCESS`	This function completed successfully.
`ADI_PWR_RESULT_NO_MEMORY`	The `szConfig` value is insufficient.
`ADI_PWR_RESULT_FAILED`	The address of `hConfig` is zero.
`ADI_PWR_RESULT_CALL_IGNORED`	The PM module has not been initialized.

adi_pwr_Reset

Description

This function resets the PLL controller to its hardware reset values.

Prototype

```
void adi_pwr_Reset(void);
```

Arguments

None

Return Value

None

adi_pwr_SaveConfig

Description

This function stores the current configuration values into the memory area pointed to by the `hConfig` argument. If the EBIU Module is initialized, its configuration is also saved; otherwise, the appropriate fields are undefined.

Prototype

```
ADI_PWR_RESULT adi_pwr_SaveConfig(
        const ADI_PWR_CONFIG_HANDLE hConfig,
        const size_t szConfig
);
```

Arguments

`hConfig`	The address of the memory location into which the current configuration is to be stored.
`szConfig`	Number of bytes available at the given address. The value must be greater than or equal to the `adi_pwr_GetConfig-Size()` return value.

Return Value

In the debug variant of the library, `adi_pwr_Init` returns the following results codes. Otherwise, the value of `ADI_PWR_RESULT_SUCCESS` is returned.

`ADI_PWR_RESULT_SUCCESS`	This function completed successfully.
`ADI_PWR_RESULT_NO_MEMORY`	The `szConfig` value is insufficient.
`ADI_PWR_RESULT_FAILED`	The address of `hConfig` is zero.
`ADI_PWR_RESULT_CALL_IGNORED`	The PM module has not been initialized.

adi_pwr_SetFreq

Description

This function sets the PLL controller to provide CCLK and SCLK values as close as possible to the requested values, in MHz. If the voltage regulator is not disabled, it is adjusted (where necessary) to provide the minimum voltage that can sustain the requested frequencies.

The processor is idled to effect the changes.

This function always finds a solution where the CSEL divider in the PLL_DIV register is unity. If the PLL Input Divider is requested, then the difference between the requested and obtained values is minimized.

To determine the values set by this function, use adi_pwr_GetFreq.

Prototype

```
ADI_PWR_RESULT adi_pwr_SetFreq(
         const u32 fcclk,
         const u32 fsclk,
         const ADI_PWR_DF df);
```

Arguments

fcclk	This is the requested CCLK value in MHz. If this is set to zero, the adi_pwr_SetFreq function gives priority to matching the given SCLK frequency and calculates and sets a CCLK frequency as close as possible to the maximum possible for the current voltage level.
fsclk	This is the requested SCLK value in MHz.
df	An ADI_PWR_DF enumeration value indicates whether or not the PLL input divider is to be enabled (see "ADI_PWR_DF" on page 3-36). If enabled, then it can lead to lower power dissipation. Passing a value of ADI_PWR_DF_NONE indicates that the routine should decide whether to enable or disable it.

Return Value

In the debug variant of the library, the `adi_pwr_SetFreq` function returns one of the following result codes. Otherwise it returns `ADI_PWR_RESULT_SUCCESS`.

`ADI_PWR_RESULT_SUCCESS`	This process completed successfully.
`ADI_PWR_RESULT_IGNORED`	The PM module has not been initialized.

adi_pwr_SetMaxFreqForVolt

Description

This function sets the Voltage Regulator control register, `VR_CTL`, with the required voltage level and adjusts the processor's `CCLK` and `SCLK` values to the maximum sustainable level.

The processor is idled to effect the changes.

Prototype

```
ADI_PWR_RESULT adi_pwr_SetMaxFreqForVolt(
        const ADI_PWR_VLEV vlev
);
```

Arguments

`vlev`	The required voltage level is set as an `ADI_PWR_VLEV` enumeration value (see "ADI_PWR_VLEV" on page 3-46).

Return Value

In debug variant of the library, the `adi_pwr_SetMaxFreqForVolt` function returns the following result codes. Otherwise, `ADI_PWR_RESULT_SUCCESS` is returned.

`ADI_PWR_RESULT_INVALID_VR_VLEV`	The `vlev` value is invalid.
`ADI_PWR_RESULT_VR_BYPASSED`	The voltage regulator is bypassed. A call to `adi_dma_SetVoltageRegulator` with a non-zero switching frequency value is required prior to this call (see "adi_pwr_SetVoltageRegulator" on page 3-26).
`ADI_PWR_RESULT_CALL_IGNORED`	This process completed successfully.

adi_pwr_SetPowerMode

Description

This function sets the Power mode of the processor. There are five modes:

- **Full-On** – The processor core clock, `CCLK`, and system clock, `SCLK`, run at the frequencies set via `adi_pwr_SetFreq` or `adi_pwr_SetVoltageRegulator` and full DMA is enabled.
- **Active** – The PLL is bypassed so that the processor core clock and system clock run at the `CLKIN` input clock frequency. DMA access is available to configured L1 memories appropriately.
- **Sleep** – The core processor is idled. The system clock continues to run at the speed set via `adi_pwr_SetFreq` or `adi_pwr_SetVoltageRegulator` and DMA is restricted to external memory.
- **Deep Sleep** – The processor core and all peripherals except the Real-Time Clock (RTC) are disabled. DMA is not supported in this mode.

  SDRAM is set to Self-Refresh Mode. The voltage regulator is powered up on RTC interrupt or a hardware reset event. In both cases the core reset sequence is initiated.
- **Hibernate** - The internal voltage regulator is powered down. SDRAM is set to Self-Refresh Mode. The voltage regulator is powered up on hardware reset.

IMPORTANT, please note that until SDRAM is properly configured and the refresh rate appropriate, data held in SDRAM will decay! This only applies to exiting Hibernate mode or Deep Sleep by a hardware reset event. For ADSP-BF531, ADSP-BF532 and ADSP-BF533 cores, the `SCKE` pin on the processor is always asserted on reset, causing the SDRAM to exit self-refresh mode. This behavior is a constraint of PC-133 compliance. For the ADSP-BF534, ADSP-BF536 and ADSP-BF537 cores, this restriction can be circumvented by enabling the `CKELOW` bit in the `VR_CTL` register (see "adi_pwr_SetVoltageRegulator" on page 3-26). This can also be achieved by inserting the following command value pair to the table to be passed to the `adi_pwr_Init` function:

```
{ ADI_PWR_CMD_SET_PC133_COMPLIANCE, 0 }
```

Prototype

```
ADI_PWR_RESULT adi_pwr_SetPowerMode(
         const ADI_PWR_MODE mode);
```

Arguments

`mode`	The `ADI_PWR_MODE` value indicates the state to which the processor is to be transitioned (see "ADI_PWR_MODE" on page 3-38).

Return Value

In the debug variant of the library, the function `adi_pwr_SetPowerMode` returns one of the following result codes. Otherwise the function returns `ADI_PWR_RESULT_SUCCESS`.

`ADI_PWR_RESULT_SUCCESS`	This process completed successfully.
`ADI_PWR_RESULT_CALL_IGNORED`	The PM module has not been initialized.
`ADI_PWR_RESULT_INVALID_MODE`	Either an incorrect mode has been requested or the requested mode cannot be reached from the current mode.

adi_pwr_SetVoltageRegulator

Description

This function sets the Voltage Regulator control register, VR_CTL, with one or more of the following fields.

The following fields are applicable to all Blackfin processors.	
VLEV	This is the required voltage level as an ADI_PWR_VLEV enumeration value (see "ADI_PWR_VLEV" on page 3-46).
FREQ	This is the required voltage regulator switching oscillator frequency as an ADI_PWR_VR_FREQ enumeration value (see "ADI_PWR_VR_FREQ" on page 3-50). Please note, supply ADI_PWR_VR_FREQ_POWERDOWN to bypass the onboard voltage regulator.
GAIN	This is the required gain value as an ADI_PWR_VR_GAIN enumeration value (see "ADI_PWR_VR_GAIN" on page 3-50).
WAKE	An ADI_PWR_VR_WAKE enumeration value indicating whether the voltage regulator can be awakened from power down upon an interrupt from the Real Time Clock or a low going edge on the RESET# pin (see "ADI_PWR_VR_WAKE" on page 3-51).
The following fields are applicable only to the ADSP-BF534, ADSP-BF536 and ADSP-BF537 processors:	
PHYWE	An ADI_PWR_VR_PHYWE enumeration value indicating whether the voltage regulator can be awakened from power down by activity on the Ethernet PHY (see "ADI_PWR_VR_PHYWE" on page 3-50).
CANWE	An ADI_PWR_VR_CANWE enumeration value indicating whether the voltage regulator can be awakened from power down by activity on the CAN bus (see "ADI_PWR_VR_CANWE" on page 3-47).

CLKBUFOE	An `ADI_PWR_VR_CLKBUFOE` enumeration value to govern whether or not other devices, most likely the Ethernet PHY, are to be clocked by the input clock, CLKIN.This bit should be set if the Ethernet PHY is to be used on the ADSP-BF537 EZ-Kit (see "ADI_PWR_VR_CLKBUFOE" on page 3-49).
CKELOW	An `ADI_PWR_VR_CKELOW` enumeration value to govern whether to protect against the default reset state behavior of setting the EBIU pins to their inactive state. This bit should be set if the SDRAM is to be placed into self-refresh mode while the processor is in Hibernate state (see "ADI_PWR_VR_CKELOW" on page 3-48).

These values are communicated to the `adi_pwr_SetVoltageRegulator` function by passing either a single command-value pair or a sequence of pairs in a table terminated with the `ADI_PWR_CMD_END` command, in the same way as for the `adi_pwr_Control` function. For more detailed information, refer to "adi_pwr_Control" on page 3-9.

For example, to bypass the built-in voltage regulator, this code could be used.

```
adi_pwr_SetVoltageRegulator(ADI_PWR_SET_VR_FREQ,  (void*) ADI_
PWR_VR_FREQ_POWERDOWN);
```

The following table defines the command-value pairs that can be used with the `adi_pwr_SetVoltageRegulator` function. Use of any other pairs is invalid.

Command	**Associated data value**
The following commands are applicable to all Blackfin processors.	
`ADI_PWR_CMD_END`	The data value is ignored as the command simply marks the end of a table of command pairs.
`ADI_PWR_CMD_PAIR`	Used to tell `adi_pwr_SetVoltageRegulator` that a single command pair is being passed.
`ADI_PWR_CMD_TABLE`	Used to tell `adi_pwr_SetVoltageRegulator` that a table of command pairs is being passed.

ADI_PWR_CMD_SET_VR_VLEV	An ADI_PWR_VLEV value specifying the Voltage Level required of the voltage regulator (see "ADI_PWR_VLEV" on page 3-46).
ADI_PWR_CMD_SET_VR_FREQ	An ADI_PWR_VR_FREQ value specifying the required voltage regulator switching oscillator frequency (see "ADI_PWR_VR_FREQ" on page 3-50). Use the ADI_PWR_VR_FREQ_POWERDOWN value to bypass the onboard voltage regulator.
ADI_PWR_CMD_SET_VR_GAIN	An ADI_PWR_VR_GAIN value specifying the internal loop gain of the switching regulator loop (see "ADI_PWR_VR_GAIN" on page 3-50).
ADI_PWR_CMD_SET_VR_WAKE	An ADI_PWR_VR_WAKE value indicating whether to enable/disable the WAKE bit (see "ADI_PWR_VR_WAKE" on page 3-51).
The following commands are applicable to ADSP-BF534, ADSP-BF536 and ADSP-BF537 processors:	
ADI_PWR_CMD_SET_VR_PHYWE	An ADI_PWR_VR_PHYWE enumeration value indicating whether to enable/disable the PHYWE bit (see "ADI_PWR_VR_PHYWE" on page 3-50).
ADI_PWR_CMD_SET_VR_CANWE	An ADI_PWR_VR_CANWE enumeration value indicating whether to enable/disable the CANWE bit (see "ADI_PWR_VR_CANWE" on page 3-47).
ADI_PWR_CMD_SET_VR_CLKBUFOE	An ADI_PWR_VR_CLKBUFOE enumeration value indicating to enable/disable the CLKBUFOE bit (see "ADI_PWR_VR_CLKBUFOE" on page 3-49).
ADI_PWR_CMD_SET_VR_CKELOW	An ADI_PWR_VR_CKELOW enumeration value indicating whether to enable/disable the CKELOW bit (see "ADI_PWR_VR_CKELOW" on page 3-48).

The processor's CCLK and SCLK frequencies are not adjusted. The processor is idled to effect the changes, except in the case of the ADI_PWR_CMD_SET_VR_WAKE ADI_PWR_CMD_SET_VR_CANWE commands. If the requested voltage level is insufficient to sustain the current frequency values, the function return an error without amending any settings.

Prototype

```
ADI_PWR_RESULT adi_pwr_SetVoltageRegulator(
            ADI_PWR_COMMAND command,
            void *Value
);
```

Arguments

`Command`	An `ADI_PWR_COMMAND` enumeration value specifies the meaning of the associated value argument.
`Value`	This is the required value (see "adi_pwr_SetVoltageRegulator" on page 3-26).

Return Value

In debug variant of the library, the `adi_pwr_SetVoltageRegulator` function returns the following result codes. Otherwise, `ADI_PWR_RESULT_SUCCESS` is returned.

`ADI_PWR_RESULT_INVALID_VR_VLEV`	The `VLEV` argument is invalid or insufficient to sustain the current core and system clock frequencies.
`ADI_PWR_RESULT__INVALID_VR_FREQ`	The `FREQ` value is invalid.
`ADI_PWR_RESULT__INVALID_VR_GAIN`	The `GAIN` value is invalid.
`ADI_PWR_RESULT__INVALID_VR_WAKE`	The `WAKE` value is invalid.
`ADI_PWR_RESULT_INVALID_VR_PHYWE`	The `PHYWE` value is invalid.
`ADI_PWR_RESULT_INVALID_VR_CANWE`	The `CANWE` value is invalid.
`ADI_PWR_RESULT_INVALID_VR_CLKBUFOE`	The `CLKBUFOE` value is invalid.

ADI_PWR_RESULT_INVALID_VR_CKELOW	The CKELOW value is invalid.
ADI_PWR_RESULT__BAD_COMMAND	The Command argument is unrecognized.
ADI_PWR_RESULT_IGNORED	The PM module has not been initialized.

Public Data Types and Enumerations

This section provides descriptions of the PM public data types and enumerations

ADI_PWR_COMMAND

The ADI_PWR_COMMAND enumeration type describes the command type in an ADI_PWR_COMMAND_PAIR structure. The following table details the available commands, the associated data values and the valid context for their use:

Commands	Associated Data Value
Commands that can be used with the adi_pwr_Init, adi_pwr_Control, and adi_pwr_SetVoltageRegulator functions:	
ADI_PWR_CMD_END	The data value is ignored as the command simply marks the end of a table of command pairs.
Commands that can be used with either the adi_pwr_Control or adi_pwr_SetVoltageRegulator functions:	
ADI_PWR__CMD_PAIR	Indicates that a single command pair is being passed.
ADI_PWR__CMD_TABLE	Indicates that a table of command pairs is being passed.
Commands that can be used with either the adi_pwr_Init or adi_pwr_Control functions:	
ADI_PWR_CMD_SET_INPUT_DELAY	An ADI_PWR_INPUT_DELAY value specifying whether or not to add approximately 200ps of delay to the time when inputs are latched on the external memory interface (see "ADI_PWR_INPUT_DELAY" on page 3-37).
ADI_PWR_CMD_SET_OUTPUT_DELAY	An ADI_PWR_OUTPUT_DELAY value specifying whether or not to add approximately 200ps of delay to external memory output signals (see "ADI_PWR_OUTPUT_DELAY" on page 3-37.
ADI_PWR_CMD_SET_PLL_LOCKCNT	A u16 value specifying the number of SCLK cycles to occur during the IDLE stage of the PLL programming sequence before the processor sets the PLL_LOCKED bit in the PLL_STAT register. This value is held in the PLL_LOCKCNT register.
Commands valid only when passed to the adi_pwr_Init function:	
ADI_PWR_CMD_SET_EZKIT	An ADI_PWR_EZKIT value to identify the EZ-KIT for which the power management is to be configured (see "ADI_PWR_EZKIT" on page 3-37).

ADI_PWR_CMD_SET_PROC_VARIANT	An ADI_PWR_PROC_KIND value specifying the processor variant (see "ADI_PWR_PROC_KIND" on page 3-41).
ADI_PWR_CMD_SET_PACKAGE	An ADI_PWR_PACKAGE_KIND value describing the packaging type of the processor (see "ADI_PWR_PACKAGE_KIND" on page 3-39).
ADI_PWR_CMD_SET_CLKIN	A u16 value specifying the external clock frequency, CLKIN, in MHz supplied to the processor.
ADI_PWR_CMD_SET_VDDINT	An ADI_PWR_VLEV value specifying the Core Voltage Level provided by an external voltage regulator (see "ADI_PWR_VLEV" on page 3-46).
ADI_PWR_CMD_SET_VDDEXT	An ADI_PWR_VDDEXT value specifying the external voltage level applied to the internal voltage regulator (see "ADI_PWR_VDDEXT" on page 3-45).
ADI_PWR_CMD_FORCE_DATASHEET_VALUES	Enforces the Core Clock frequency limits for each voltage level as defined in the relevant data sheet. (default).
ADI_PWR_CMD_SET_CCLK_TABLE	The address of a table containing ADI_PWR_NUM_VLEVS values of type u16 detailing the max CCLK frequency for each ADI_PWR_VLEV value. These values will be used instead of the data sheet values.
ADI_PWR_CMD_SET_IVG	An u16 value specifying the IVG level for the PLL_WAKEUP event. This defaults to 7.
ADI_PWR_CMD_SET_PC133_COMPLIANCE	An ADI_PWR_PC133_COMPLIANCE value specifying whether or not the SDRAM is to comply with the PC-133 standard. Non-compliance to the standard is required to enable the processor to return from Hibernate mode without losing the contents of SDRAM. This value prevents SDRAM decay during reset, enabling the contents of SDRAM to be preserved through the Hibernate-reset or Deep Sleep reset cycle. (This command applies to ADSP-BF534, ADSP-BF536, and ADSP-BF537 cores only).
Commands valid only when passed to the adi_pwr_SetVoltageRegulator function:	
ADI_PWR_CMD_SET_VR_VLEV	An ADI_PWR_VLEV value specifying the voltage level required of the voltage regulator (see "ADI_PWR_VLEV" on page 3-46).

ADI_PWR_CMD_SET_VR_FREQ	An ADI_PWR_VR_FREQ value specifying the required voltage regulator switching oscillator frequency. Use the ADI_PWR_FREQ_POWERDOWN value to bypass the onboard voltage regulator (see "ADI_PWR_VR_FREQ" on page 3-50).
ADI_PWR_CMD_SET_VR_GAIN	An ADI_PWR_VR_GAIN value specifying the internal loop gain of the switching regulator loop ("ADI_PWR_VR_GAIN" on page 3-50).
ADI_PWR_CMD_SET_VR_WAKE	An ADI_PWR_VR_WAKE value specifying if the voltage regulator is to be awakened from powerdown upon an interrupt from the RTC or a low going edge on the RESET# pin ("ADI_PWR_VR_WAKE" on page 3-51).
ADI_PWR_CMD_SET_VR_PHYWE	An ADI_PWR_VR_PHYWE enumeration value indicating whether to enable/disable the PHYWE bit. (ADSP-BF534, ADSP-BF536 and ADSP-BF537 cores only) ("ADI_PWR_VR_PHYWE" on page 3-50).
ADI_PWR_CMD_SET_VR_CANWE	An ADI_PWR_VR_CANWE enumeration value indicating whether to enable or disable the CANWE bit. (ADSP-BF534, ADSP-BF536, and ADSP-BF537 cores only) (see "ADI_PWR_VR_CANWE" on page 3-47).
ADI_PWR_CMD_SET_VR_CLKBUFOE	An ADI_PWR_VR_CLKBUFOE enumeration value indicating whether to enable or disable the CLKBUFOE bit. (ADSP-BF534, ADSP-BF536 and ADSP-BF537 cores only) ("ADI_PWR_VR_CLKBUFOE" on page 3-49).
ADI_PWR_CMD_SET_VR_CKELOW	An ADI_PWR_VR_CKELOW enumeration value indicating whether to enable or disable the CKELOW bit. (ADSP-BF534, ADSP-BF536, and ADSP-BF537 cores only) ("ADI_PWR_VR_CKELOW" on page 3-48).
Commands valid only when passed to the adi_pwr_Control function:	
ADI_PWR_CMD_GET_VDDINT	An ADI_PWR_VLEV value containing the maximum core voltage level ("ADI_PWR_VLEV" on page 3-46).

ADI_PWR_CMD_GET_VR_VLEV	An ADI_PWR_VLEV value containing the current voltage level of the internal voltage regulator. Not applicable when the internal regulator is bypassed ("ADI_PWR_VLEV" on page 3-46).
ADI_PWR_CMD_GET_VR_FREQ	An ADI_PWR_FREQ value containing the current voltage regulator switching oscillator frequency ("ADI_PWR_VR_FREQ" on page 3-50).
ADI_PWR_CMD_GET_VR_GAIN	An ADI_PWR_GAIN value containing the internal loop gain of the switching regulator loop ("ADI_PWR_VR_GAIN" on page 3-50).
ADI_PWR_CMD_GET_VR_WAKE	An ADI_PWR_VR_WAKE value (specifying if the voltage can be awakened from powerdown upon an interrupt from the RTC or a low going edge on the RESET# pin ("ADI_PWR_VR_WAKE" on page 3-51).
ADI_PWR_CMD_GET_VR_PHYWE	An ADI_PWR_VR_PHYWE enumeration value indicating if the PHYWE bit has been enabled/ disabled. (ADSP-BF534, ADSP-BF536 and ADSP-BF537 cores only) ("ADI_PWR_VR_PHYWE" on page 3-50).
ADI_PWR_CMD_GET_VR_CANWE	An ADI_PWR_VR_CANWE enumeration value indicating if the CANWE bit has been enabled or disabled. (ADSP-BF534, ADSP-BF536 and ADSP-BF537 cores only) ("ADI_PWR_VR_CANWE" on page 3-47).
ADI_PWR_CMD_GET_VR_CLKBUFOE	An ADI_PWR_VR_CLKBUFOE enumeration value indicating if the CLKBUFOE bit has been enabled or disabled. (ADSP-BF534, ADSP-BF536, and ADSP-BF537 cores only) ("ADI_PWR_VR_CLKBUFOE" on page 3-49).
ADI_PWR_CMD_GET_VR_CKELOW	An ADI_PWR_VR_CKELOW enumeration value indicating if the CKELOW bit has been enabled or disabled. (ADSP-BF534, ADSP-BF536, and ADSP-BF537 cores only) ("ADI_PWR_VR_CKELOW" on page 3-48).
ADI_PWR_CMD_GET_PLL_LOCKCNT	A u16 value containing the value in the PLL_LOCKCNT register.

ADI_PWR_COMMAND_PAIR

This data type is used to enable the generation of a table of control commands to be sent to the Power Management module via the `adi_pwr_Init`, `adi_pwr_SetVoltageRegulator`, and `adi_pwr_Control` functions:

```
typedef struct _ADI_PWR_COMMAND_PAIR {
        ADI_PWR_COMMAND kind;
        void *value;
} ADI_PWR_COMMAND_PAIR;
```

Valid values for the `kind` field are provided in `ADI_PWR_COMMAND`.

ADI_PWR_CSEL

This data type defines the Core Clock divider bit field in the `PLL_DIV` register. Valid values are:

`ADI_PWR_PWR_CSEL_1`	Divides Voltage Core Oscillator frequency by 1
`ADI_PWR_PWR_CSEL_2`	Divides Voltage Core Oscillator frequency by 2
`ADI_PWR_PWR_CSEL_4`	Divides Voltage Core Oscillator frequency by 4
`ADI_PWR_PWR_CSEL_8`	Divides Voltage Core Oscillator frequency by 4

ADI_PWR_DF

This data type defines the values for the `DF` bit in the PLL Control register. A value of `ADI_PWR_DF_ON` causes the value of `CLKIN/2` to be passed to the PLL module. According to the hardware reference manual for the ADSP-BF533 processor, this leads to lower power dissipation[*],

`ADI_PWR_DF_NONE`	Indicates that no PLL input divider value is to be set.
`ADI_PWR_DF_OFF`	Pass `CLKIN` to the PLL.
`ADI_PWR_DF_ON`	Pass `CLKIN/2` to the PLL.

[*] See *ADSP-BF533 Blackfin Hardware Reference Manual*, Revision 1.0, December 2003, page 8-4.

ADI_PWR_EZKIT

This enumeration type describes the Revision of the EZ-KIT board for which the power management module is to be configured. For Blackfin these are:

`ADI_PWR_EZKIT_BF533_750MHZ`	The ADSP-BF533 EZ-KIT LITE board with the SKBC750 processor. Please note, this option disables the internal voltage regulator, since it is assumed the external voltage regulator has been enabled. To use the 750MHz kit with the internal regulator, use the `ADI_PWR_EZKIT_BF533_600MHZ` option instead.
`ADI_PWR_EZKIT_BF533_600MHZ`	The ADSP-BF533 EZ-KIT LITE board with either the SKBC600 or SKBC750 processor, with the internal voltage regulator enabled, which caps the latter's core clock (CCLK) at 600MHz.
`ADI_PWR_EZKIT_BF537_600MHZ`	The ADSP-BF537 EZ-KIT LITE board with the SKBC600 processor.

ADI_PWR_INPUT_DELAY

This data type defines the values that the input delay bit can take in the PLL Control register.

`ADI_PWR_INPUT_DELAY_OFF`	Do not add input delay.
`ADI_PWR_INPUT_DELAY_ON`	Add approximately 200ps of delay to the time when inputs are latched on the external memory interface.

ADI_PWR_OUTPUT_DELAY

This data type defines the values that the output delay bit can take in the PLL Control register.

`ADI_PWR_OUTPUT_DELAY_OFF`	Do not add output delay.
`ADI_PWR_OUTPUT_DELAY_ON`	Add approximately 200ps of delay to external memory output signals.

ADI_PWR_MODE

This data type defines the power mode of the processor. Valid power mode values are:

`ADI_PWR_MODE_FULL_ON`	Processor is in *Full-On* mode; clock speeds are as programmed.
`ADI_PWR_MODE_ACTIVE`	Processor is in *Active* mode with only L1 DMA access allowed. `CCLK` and `SCLK` are pegged to `CLKIN` as the PLL controller is bypassed, providing medium power saving.
`ADI_PWR_MODE_ACTIVE_PLLDISABLED`	Processor is in *Active* mode with only L1 DMA access allowed. `CCLK` and `SCLK` are pegged to `CLKIN` as the PLL controller is bypassed *and* disabled, providing medium power saving.
`ADI_PWR_MODE_SLEEP`	Processor is in *Sleep* mode. It can be woken up with any interrupt appropriately masked in the `SIC_IWR` register, providing high power saving.
`ADI_PWR_MODE_DEEP_SLEEP`	Processor is in *Deep Sleep* mode. It can only be woken up with an appropriately masked RTC interrupt or Reset, providing high power saving.
`ADI_PWR_MODE_HIBERNATE`	The processor is in Hibernate mode. It can only be awakened on system Reset, providing maximum power saving.

ADI_PWR_PACKAGE_KIND

This data type defines the package type of the processor. Along with the external voltage ("ADI_PWR_VDDEXT" on page 3-45), this determines the heat dissipation of the part.

`ADI_PWR_PACKAGE_MBGA`	`MBGA` - identified by the hemispherical contacts on the under surface of the processor.
`ADI_PWR_PACKAGE_LQFP`	`LQFP` - identified by the leg contacts around the edges of the processor.

ADI_PWR_PCC133_COMPLIANCE

This data type defines the valid values for setting PC-133 compliance or otherwise. This value governs whether or not the SCKE pin on the processor is asserted on reset.

ADI_PWR_PC133_COMPLIANCE_DISABLED	SCKE is asserted on reset—SDRAM contents are invalidated.
ADI_PWR_PC133_COMPLIANCE_ENABLED	SCKE is not asserted on reset—SDRAM contents are maintained.

ADI_PWR_PROC_KIND

This data type defines the processor variant, which governs the appropriate limits for speed selection. For ADSP-BF533 processors, these are:

`ADI_PWR_PROC_BF533SKBC750`	The ADSP-BF533SKBC750 processor.
`ADI_PWR_PROC_BF533SKBC600`	The ADSP-BF533SKBC600 processor.
`ADI_PWR_PROC_BF533SBBC500`	The ADSP-BF533SBBC500 processor.
`ADI_PWR_PROC_BF531_OR_BF532`	All package types for ADSP-BF531 and ADSP-BF532.
`ADI_PWR_PROC_BF537SKBC1600`	The ADSP-BF537SKBC1600 processor.
`ADI_PWR_PROC_BF537SBBC1500`	The ADSP-BF537SBBC1500 processor.
`ADI_PWR_PROC_BF536SBBC1400`	The ADSP-BF537SBBC1400 processor.
`ADI_PWR_PROC_BF536SBBC1300`	The ADSP-BF537SBBC1300 processor.
`ADI_PWR_PROC_BF534SBBC1500`	The ADSP-BF534SBBC1500 processor.
`ADI_PWR_PROC_BF534SBBC1400`	The ADSP-BF534SBBC1400 processor.

ADI_PWR_RESULT

The Power Management module functions return a result code of the enumeration type, `ADI_PWR_RESULT`. The PM module return values are:

`ADI_PWR_RESULT_SUCCESS`	This routine completed successfully.
`ADI_PWR_RESULT_FAILED`	A generic failure was encountered.
`ADI_PWR_RESULT_NO_MEMORY`	Insufficient memory for configuration values to be stored.
`ADI_PWR_RESULT_BAD_COMMAND`	The command is not recognized.
`ADI_PWR_RESULT_CALL_IGNORED`	A function call has been ignored with no action taken, due to the PM module not being initialized.
`ADI_PWR_RESULT_INVALID_VDDEXT`	An invalid external voltage level has been specified.
`ADI_PWR_RESULT_INVALID_PROCESSOR`	The processor type specified is invalid.
`ADI_PWR_RESULT_INVALID_IVG`	The IVG level supplied for PLL wakeup is invalid.
`ADI_PWR_RESULT_INVALID_INPUT_DELAY`	The input delay value is invalid.
`ADI_PWR_RESULT_INVALID_OUTPUT_DELAY`	The output delay value is invalid.
`ADI_PWR_RESULT_INVALID_LOCKCNT`	The PLL lock count value is invalid.
`ADI_PWR_RESULT_INVALID_EZKIT`	This is an invalid EZ-Kit type.
`ADI_PWR_RESULT_INVALID_MODE`	An invalid operating mode has been specified.
`ADI_PWR_RESULT_INVALID_CSEL`	An invalid value for `CSEL` has been specified.
`ADI_PWR_RESULT_INVALID_SSEL`	An invalid value for `SSEL` has been specified.

ADI_PWR_INVALID_CSEL_SSEL_COMBINATION	The core clock divider is greater that the system clock divider value, or both ADI_PWR_CSEL_NONE and ADI_PWR_SSEL_NONE are specified.
ADI_PWR_RESULT_VOLTAGE_REGULATOR_BYPASSED	Voltage regulator cannot be set since it is in bypass mode.
ADI_PWR_RESULT_INVALID_VR_VLEV	The VLEV argument is invalid or insufficient to sustain the current core and system clock frequencies.
ADI_PWR_RESULT_INVALID_VR_FREQ	The FREQ value is invalid.
ADI_PWR_RESULT_INVALID_VR_GAIN	The GAIN value is invalid.
ADI_PWR_RESULT_INVALID_VR_WAKE	The WAKE value is invalid.
ADI_PWR_RESULT_INVALID_VR_PHYWE	The PHYWE value is invalid.
ADI_PWR_RESULT_INVALID_VR_CANWE	The CANWE value is invalid.
ADI_PWR_RESULT_INVALID_VR_CLKBUFOE	The CLKBUFOE value is invalid.
ADI_PWR_RESULT_INVALID_VR_CKELOW	The CKELOW value is invalid.

ADI_PWR_SSEL

This data type defines the System Clock divider bit field in the `PLL_DIV` register. Valid values are:

`ADI_PWR_PWR_SSEL_1`	Divides Voltage Core Oscillator frequency by 1
`ADI_PWR_PWR_SSEL_2`	Divides Voltage Core Oscillator frequency by 2
`ADI_PWR_PWR_SSEL_3`	Divides Voltage Core Oscillator frequency by 3
`ADI_PWR_PWR_SSEL_4`	Divides Voltage Core Oscillator frequency by 4
`ADI_PWR_PWR_SSEL_5`	Divides Voltage Core Oscillator frequency by 5
`ADI_PWR_PWR_SSEL_6`	Divides Voltage Core Oscillator frequency by 6
`ADI_PWR_PWR_SSEL_7`	Divides Voltage Core Oscillator frequency by 7
`ADI_PWR_PWR_SSEL_8`	Divides Voltage Core Oscillator frequency by 8
`ADI_PWR_PWR_SSEL_9`	Divides Voltage Core Oscillator frequency by 9
`ADI_PWR_PWR_SSEL_10`	Divides Voltage Core Oscillator frequency by 10
`ADI_PWR_PWR_SSEL_11`	Divides Voltage Core Oscillator frequency by 11
`ADI_PWR_PWR_SSEL_12`	Divides Voltage Core Oscillator frequency by 12
`ADI_PWR_PWR_SSEL_13`	Divides Voltage Core Oscillator frequency by 13
`ADI_PWR_PWR_SSEL_14`	Divides Voltage Core Oscillator frequency by 14
`ADI_PWR_PWR_SSEL_15`	Divides Voltage Core Oscillator frequency by 15

ADI_PWR_VDDEXT

This data type defines the external voltage (`VDDEXT`) supplied to the voltage regulator

`ADI_PWR_VDDEXT_330`	3.3 Volts
`ADI_PWR_VDDEXT_250`	2.5 Volts

ADI_PWR_VLEV

This data type defines the acceptable voltage levels for the voltage regulator. The following table lists the values for ADSP-BF533 and ADSP-BF561 processors.

`ADI_PWR_VLEV_085`	0.85 V
`ADI_PWR_VLEV_090`	0.90 V
`ADI_PWR_VLEV_095`	0.95 V
`ADI_PWR_VLEV_100`	1.00 V
`ADI_PWR_VLEV_105`	1.05 V
`ADI_PWR_VLEV_110`	1.10 V
`ADI_PWR_VLEV_115`	1.15 V
`ADI_PWR_VLEV_120`	1.20 V (default)
`ADI_PWR_VLEV_125`	1.25 V
`ADI_PWR_VLEV_130`	1.30 V
`ADI_PWR_VLEV_135`	1.35 V
`ADI_PWR_VLEV_140`	1.40 V

ADI_PWR_VR_CANWE

This data type defines the valid values for the `CANWE` bit in the Voltage Regulator Control register. If enabled, the Voltage Regulator can be awakened from powerdown by activity on the Controller Area Network (CAN) interface.

`ADI_PWR_VR_CANWE_DISABLED`	Disable wake up by CAN activity.
`ADI_PWR_VR_CANWE_ENABLED`	Enable wake up by CAN activity.

ADI_PWR_VR_CKELOW

This data type defines the valid values for the `CKELOW` bit in the Voltage Regulator Control register. If enabled, the `SCKE` pin is driven low on system reset to enable the SDRAM to remain in self-refresh mode.

`ADI_PWR_VR_PHYWE_DISABLED`	Drive SCKE high on reset—SDRAM contents are invalidated.
`ADI_PWR_VR_PHYWE_ENABLED`	Drive SCKE low on reset—SDRAM contents are maintained.

ADI_PWR_VR_CLKBUFOE

This data type defines the valid values for the `CLKBUFOE` bit in the Voltage Regulator Control register. If enabled, the `CLKIN` signal can be shared with peripheral devices, especially the Ethernet PHY.

`ADI_PWR_VR_CLKBUFOE_DISABLED`	Disable CLKIN sharing.
`ADI_PWR_VR_CLKBUFOE_ENABLED`	Enable CLKIN sharing.

ADI_PWR_VR_FREQ

This data type defines the acceptable switching frequency values for the voltage regulator. Its value is linked to the switching capacitor and inductor values. The higher the frequency setting, the smaller the capacitor and inductor values. The following table lists the valid values for all Blackfin processors.

`ADI_PWR_VR_FREQ_POWERDOWN`	Powerdown/bypass onboard regulation
`ADI_PWR_VR_FREQ_333KHZ`	333 kHz
`ADI_PWR_VR_FREQ_667KHZ`	667 kHz
`ADI_PWR_VR_FREQ_1MHZ`	1 MHz (default)

ADI_PWR_VR_GAIN

This data type defines the acceptable values for the internal loop gain of the switching regulator loop. The gain controls how quickly the voltage output settles on its final value. The higher the gain, the quicker the settling time. High gain settings cause greater overshoot in the process.

`ADI_PWR_VR_GAIN_5`	5
`ADI_PWR_VR_GAIN_110`	10
`ADI_PWR_VR_GAIN_20`	20 (default)
`ADI_PWR_VR_GAIN_50`	50

ADI_PWR_VR_PHYWE

This data type defines the values for the `PHYWE` bit in the Voltage Regulator Control register. If enabled, the Voltage Regulator can be awakened from powerdown by activity on the PHY interface.

`ADI_PWR_VR_PHYWE_DISABLED`	Disable wake up by PHY activity.
`ADI_PWR_VR_PHYWE_ENABLED`	Enable wake up by PHY activity.

ADI_PWR_VR_WAKE

This data type defines the values for the WAKE bit in the Voltage Regulator Control register. If enabled (ADI_PWR_VR_WAKE_ENABLED), the voltage regulator can be awakened from powerdown (ADI_PWR_VR_FREQ_POWERDOWN) upon an RTC interrupt or a low-going edge on the RESET pin.

ADI_PWR_VR_WAKE_DISABLED	Disables wake up by RTC and RESET.
ADI_PWR_VR_WAKE_ENABLED	Enables wake up by RTC and RESET.

PM Module Macros

The following macros are provided for convenience.

ADI_PWR_VLEV_DEFAULT	The default/reset voltage level ADI_PWR_VLEV_130.
ADI_PWR_VLEV_MIN	The minimum voltage level ADI_PWR_VLEV_085.
ADI_PWR_VLEV_MAX	The maximum voltage level ADI_PWR_VLEV_120.
ADI_PWR_VOLTS(V)	Returns the voltage in volts as a float for the given level.
ADI_PWR_MILLIVOLTS(V)	Returns an integer value of the Voltage in millivolts for the given level.
ADI_PWR_VR_FREQ_DEFAULT	The default/reset switching frequency value, ADI_PWR_FREQ_1MHZ.
ADI_PWR_VR_FREQ_MIN	The minimum switching frequency value, ADI_PWR_FREQ_POWERDOWN.
ADI_PWR_VR_FREQ_MAX	The maximum switching frequency value, ADI_PWR_FREQ_1MHZ.
ADI_PWR_VR_GAIN_DEFAULT	The default/reset voltage regulator gain value, ADI_PWR_GAIN_20.
ADI_PWR_VR_GAIN_MIN	The minimum voltage regulator gain value, ADI_PWR_GAIN_5.
ADI_PWR_VR_GAIN_MAX	The default/reset voltage regulator gain value ADI_PWR_GAIN_20.

ADI_PWR_CLKIN_EZKIT_BF533	External clock frequency of EZ-KIT for ADSP-BF533.
ADI_PWR_CLKIN_EZKIT_BF537	External clock frequency of EZ-KIT for ADSP-BF537.
ADI_PWR_VDEXT_EZKIT_BF533	External voltage level of EZ-KIT for ADSP-BF533 (ADI_PWR_VDDEXT_330).
ADI_PWR_VDEXT_EZKIT_BF537	External voltage level of EZ-KIT for ADSP-BF537 (ADI_PWR_VDDEXT_330).
ADI_PWR_PROC_BF533SKBCZ600	Equivalent processor type to ADI_PWR_PROC_BF533SKBC600.
ADI_PWR_PROC_BF533SBBZ500	Equivalent processor type to ADI_PWR_PROC_BF533SBBC500.
ADI_PWR_PROC_BF532SBBC400	Equivalent processor type to ADI_PWR_PROC_BF531_OR_BF532.
ADI_PWR_PROC_BF532SBST400	Equivalent processor type to ADI_PWR_PROC_BF531_OR_BF532.
ADI_PWR_PROC_BF532SBBZ400	Equivalent processor type to ADI_PWR_PROC_BF531_OR_BF532.
ADI_PWR_PROC_BF531SBBC400	Equivalent processor type to ADI_PWR_PROC_BF531_OR_BF532.
ADI_PWR_PROC_BF531SBST400	Equivalent processor type to ADI_PWR_PROC_BF531_OR_BF532.
ADI_PWR_PROC_BF531SBSTZ400	Equivalent processor type to ADI_PWR_PROC_BF531_OR_BF532.
ADI_PWR_PROC_BF531SBBZ400	Equivalent processor type to ADI_PWR_PROC_BF531_OR_BF532.
ADI_PWR_PACKAGE_PBGA	Equivalent package type to ADI_PWR_PACKAGE_MBGA.
ADI_PWR_PROC_BF537SKBC600	Equivalent processor type to ADI_PWR_PROC_BF537SKBC1600.
ADI_PWR_PROC_BF537SKBC1Z600	Equivalent processor type to ADI_PWR_PROC_BF537SKBC1600.
ADI_PWR_PROC_BF537SKBC2Z600	Equivalent processor type to ADI_PWR_PROC_BF537SKBC1600.

ADI_PWR_PROC_BF537SBBC1Z500	Equivalent processor type to ADI_PWR_PROC_BF537SBBC1500.
ADI_PWR_PROC_BF537SBBC2Z500	Equivalent processor type to ADI_PWR_PROC_BF537SBBC1500.
ADI_PWR_PROC_BF536SBBC1Z400	Equivalent processor type to ADI_PWR_PROC_BF536SBBC1400.
ADI_PWR_PROC_BF536SBBC2Z400	Equivalent processor type to ADI_PWR_PROC_BF536SBBC1400.
ADI_PWR_PROC_BF536SBBC1Z300	Equivalent processor type to ADI_PWR_PROC_BF536SBBC1300.
ADI_PWR_PROC_BF536SBBC2Z300	Equivalent processor type to ADI_PWR_PROC_BF536SBBC1300.
ADI_PWR_PROC_BF534SBBC1Z400	Equivalent processor type to ADI_PWR_PROC_BF534SBBC1400.
ADI_PWR_PROC_BF534SBBC2Z400	Equivalent processor type to ADI_PWR_PROC_BF534SBBC1400.
ADI_PWR_PROC_BF534SBBC1Z500	Equivalent processor type to ADI_PWR_PROC_BF534SBBC1500.
ADI_PWR_PROC_BF534SBBC2Z500	Equivalent processor type to ADI_PWR_PROC_BF534SBBC1500.

4 EXTERNAL BUS INTERFACE UNIT MODULE

This chapter describes the External Bus Interface Unit (EBIU) Module that enables the dynamic configuration of the SDRAM Controller in response to changes in the System Clock frequency.

This chapter contains:

Introduction

The initial goal of the External Bus Interface Unit (EBIU) Module is to enable the Power Management module to adjust the SDRAM Controller (SDC) in accordance with changes made to the System Clock (SCLK) frequency. Calls to both `adi_pwr_SetFreq` and `adi_pwr_SetMaxFreqForVolt` adjust the SDC settings to the `SCLK` frequency selected, provided the EBIU module has been initialized. For more information on the PM module, refer to "Power Management Module" on page 3-1.

Using the module is straightforward. The `adi_ebiu_Init` function is called to set up the relevant values listed in the appropriate SDRAM data sheet. Thereafter, the refresh rate is adjusted automatically each time the Power Management module changes SCLK. "Using the EBIU Module" provides a step-by-step description of how to work with the EBIU module. Sample code is also included. A complete set of abbreviations for Micron SDRAM modules and EZ-Kits is supported. These simplify the initialization of the module. Refer to "ADI_EBIU_SDRAM_EZKIT" on page 4-28 and "ADI_EBIU_SDRAM_MODULE_TYPE" on page 4-30.

The EBIU Module uses an unambiguous naming convention to safeguard against conflicts with other software libraries provided by Analog Devices or other companies. All enumeration values and `typedefs` use the `ADI_EBIU_` prefix, while functions and global variables use the lower case equivalent, `adi_ebiu_`.

There are two versions of the library for each processor, corresponding to the debug and release configurations in VisualDSP++ Release 4.0. In addition to the usual defaults for the debug configuration, the API functions perform checks on the arguments passed and report appropriate error codes, as required. In the release version of the library, most functions return one of two result codes: `ADI_EBIU_RESULT_SUCCESS` on successful completion, or `ADI_EBIU_RESULT_CALL_IGNORED` if the EBIU module has not been initialized prior to the function call.

Using the EBIU Module

The first step to using the EBIU module involves setting up the module for the SDRAM to be used. In this step a table of command-value pairs passes to the `adi_ebiu_Init` function. The information required is described in detail in "adi_ebiu_Init" on page 4-12 in the Description section. The amount of information that must be passed depends on the individual board configuration. For example, only one command-value pair must passed to describe an EZ-Kit. For a production embedded board, all information may be required.

In the following example assume that the ADSP-BF533 EZ-KIT Lite (Rev 1.3) should be configured. Specify the EZ-KIT board as follows:

```
ADI_EBIU_COMMAND_PAIR ezkit_sdram [] = {
       // ADSP-BF533 EZ-KIT LITE
       { ADI_EBIU_CMD_SET_EZKIT,(void*)ADI_EBIU_EZKIT_BF533 },
       { ADI_EBIU_CMD_END, 0 } // table end marker
};
adi_ebiu_Init(ezkit_sdram,TRUE);
```

Calls to `adi_pwr_SetFreq` or `adi_pwr_SetMaxFreqForVolt` in the Power Management module prior to the call to `adi_ebiu_Init` have no effect on the SDC settings. The second argument in the call to `adi_ebiu_Init` instructs the module to either query the system clock frequency and adjust the SDRAM refresh rate accordingly, or return without altering the SDRAM refresh rate. Specify an argument value of TRUE if a call to `adi_pwr_SetFreq` (and others) precedes the call to `adi_ebiu_Init`. Specify a value of FALSE if the call order is reversed, for example:

```
adi_ebiu_Init(ezkit_sdram, FALSE);
adi_pwr_SetMaxFreqForVolt(ADI_PWR_VLEV_090);
```

To illustrate what is required for non EZ-Kit boards, the command table is an abbreviated form of the following code:

```
ADI_EBIU_SDRAM_BANK_VALUE bank_size;
ADI_EBIU_SDRAM_BANK_VALUE bank_width;
// set bank size to 32MB
// For BF533 the size and width settings apply to all banks
bank_size.value.size = ADI_EBIU_SDRAM_BANK_32MB;

// set column address width to 9-Bit
bank_width.value.width = ADI_EBIU_SDRAM_BANK_COL_9BIT;

// set min TWR to 1 SCLK cycle + 7.5ns
ADI_EBIU_TIMING_VALUE twrmin =
       {1,{7500, ADI_EBIU_TIMING_UNIT_PICOSEC}};

// set refresh period to 8192 cycles in 64ms
ADI_EBIU_TIMING_VALUE refresh =
       {8192,{64, ADI_EBIU_TIMING_UNIT_MILLISEC}};

// set min TRAS to 44ns
ADI_EBIU_TIME trasmin = {44, ADI_EBIU_TIMING_UNIT_NANOSEC};

// set min TRP to 20ns
ADI_EBIU_TIME trpmin  = {20, ADI_EBIU_TIMING_UNIT_NANOSEC};

// set min TRCD to 20ns
ADI_EBIU_TIME trcdmin = {20, ADI_EBIU_TIMING_UNIT_NANOSEC};


// set up command table
ADI_EBIU_COMMAND_PAIR ezkit_sdram[] = {
       { ADI_EBIU_CMD_SET_SDRAM_BANK_SIZE, (void*)&bank_size },
       { ADI_EBIU_CMD_SET_SDRAM_BANK_COL_WIDTH,
                     (void*)&bank_width},
       { ADI_EBIU_CMD_SET_SDRAM_CL_THRESHOLD, (void*)100 },// MHz
       { ADI_EBIU_CMD_SET_SDRAM_TRASMIN, (void*)&trasmin },
       { ADI_EBIU_CMD_SET_SDRAM_TRPMIN, (void*)&trpmin},
       { ADI_EBIU_CMD_SET_SDRAM_TRCDMIN, (void*)&trcdmin },
        { ADI_EBIU_CMD_SET_SDRAM_TWRMIN, (void*)&twrmin },
        { ADI_EBIU_CMD_SET_SDRAM_REFRESH, (void*)&refresh },
        { ADI_EBIU_CMD_END, 0}
};
```

The sample code shows that the minimum `TWR` value comprises two parts. This reflects the definition found in the appropriate Blackfin processor data sheet where the value is expressed as one cycle of SCLK plus 7.5 ns. Similarly, the SDRAM refresh period value is expressed as the time taken for the given number of refresh cycles. The sample code shows this value as 64ms for 8192 cycles.

For boards that use Micron memory modules, users can also specify the type and size of the bank:

```
ADI_EBIU_SDRAM_BANK_VALUE bank_size;
// set bank size to 32MB
bank_size.value.size = ADI_EBIU_SDRAM_BANK_32MB;
ADI_EBIU_COMMAND_PAIR ezkit_sdram[] = {
        // MT48LC16M16-75 module
        { ADI_EBIU_CMD_SET_SDRAM_MODULE,
        (void*)ADI_EBIU_SDRAM_MODULE_MT48LC16M16A2_75 },
        { ADI_EBIU_CMD_SET_SDRAM_BANK_SIZE, (void*)&bank_size },
        { ADI_EBIU_CMD_END, 0 }
};
adi_EBIU_Init(ezkit_sdram,FALSE);
```

Further changes can be made at any time by passing further command-value pairs or tables of pairs to `adi_ebiu_Control`. For example, to pass a single command-value pair to enable the SDRAM to self-refresh during inactivity, the following code could be used:

```
adi_ebiu_Control(
        ADI_EBIU_CMD_SET_SDRAM_SRFS,
        (void*)ADI_EBIU_SDRAM_SRFS_ENABLE
);
```

Since the SDRAM settings are closely tied to the system clock (SCLK) frequency, the direct use of the `adi_ebiu_AdjustSDRAM` function from within a client application is not required since it is called automatically by the appropriate functions in the Power Management Module when SCLK changes.

EBIU API Reference

This section provides descriptions of the EBIU module's API functions.

Notation Conventions

The reference pages for the API functions use the following format:

Name and purpose of the function

Description – Function specification

Prototype – Required header file and functional prototype

Arguments – Description of function arguments

Return Value – Description of function return values

adi_ebiu_AdjustSDRAM

Description

For the passed System Clock (`SCLK`) Frequency the `adi_ebiu_AdjustSDRAM` function calculates and sets the `TRAS`, `TRP`, `TRCD` and `TWR` values in the `EBIU_SDGCTL` register and the `RDIV` value in the `EBIU_SDRRC` register.

This function is primarily used by the Power Management module to ensure that SDRAM settings are optimal for the processor's current `SCLK` frequency.

The `adi_ebiu_AdjustSDRAM` function returns without making any changes if the SDRAM has not been successfully initialized with a call to `adi_ebiu_Init`.

Prototype

```
void adi_ebiu_AdjustSDRAM(
              const u32 fsclk
);
```

Arguments

`fsclk`	The System Clock, SCLK, Frequency in MHz.

Return Value

The function returns the following values in debug or release mode.

`ADI_EBIU_RESULT_SUCCESS`	This process completed successfully.
`ADI_EBIU_RESULT_CALL_IGNORED`	The SDRAM has not been successfully initialized, or SDRAM had not been enabled.

adi_ebiu_Control

Description

This function enables the EBIU SDRAM registers to be configured according to command-value pairs (see ADI_EBIU_COMMAND_PAIR on page 4-28), using one of the following options:

- A single command-value pair is passed:

```
adi_ebiu_Control(
         ADI_EBIU_CMD_SET_SDRAM_SRFS,
         (void*)ADI_EBIU_SDRAM_SRFS_ENABLE
);
```

- A single command-value pair structure is passed:

```
ADI_EBIU_COMMAND_PAIR cmd = {
         ADI_EBIU_CMD_SET_SDRAM_SRFS,
         (void*)ADI_EBIU_SDRAM_SRFS_ENABLE
};
adi_ebiu_Control(ADI_EBIU_CMD_PAIR, (void*)&cmd);
```

- A table of ADI_EBIU_COMMAND_PAIR structures is passed. The last command-value entry in the table must be {ADI_EBIU_CMD_END, 0}:

```
ADI_EBIU_COMMAND_PAIR table[] = {
    { ADI_EBIU_CMD_SET_SDRAM_FBBRW, (void*)ADI_EBIU_SDRAM_
FBBRW_ENABLE },
    { ADI_EBIU_CMD_SET_SDRAM_CDDBG, (void*)ADI_EBIU_CDDBG_
ENABLE },
    { ADI_EBIU_CMD_END, 0 }
};
adi_ebiu_Control(
                ADI_EBIU_CMD_TABLE,
                (void*)table
        );
```

Refer to `ADI_EBIU_COMMAND` on page 4-24 and “Command Value Enumerations” on page 4-28 for the complete list of commands and associated values.

Prototype

```
adi_EBIU_RESULT adi_ebiu_Control(
        ADI_EBIU_COMMAND command,
        void *value
);
```

Arguments

`Command`	An `ADI_EBIU_COMMAND` enumeration value specifying the meaning of the associated value argument.
`Value`	This is the required value, (see in Description above).

Return Value

In Debug mode one of the following values is returned, otherwise `ADI_EBIU_RESULT_SUCCESS` or `ADI_EBIU_RESULT_CALL_IGNORED` is returned, depending on whether or not the EBIU module has been successfully initialized.

`ADI_EBIU_RESULT_BAD_COMMAND`	The command is not recognized.
`ADI_EBIU_RESULT_SUCCESS`	This function completed successfully.
`ADI_EBIU_RESULT_CALL_IGNORED`	The EBIU module is not initialized.
`ADI_EBIU_RESULT_INVALID_SDRAM_SRFS`	An invalid Self-refresh value is specified (see “ADI_EBIU_SDRAM_TCSR” on page 4-32).

ADI_EBIU_RESULT_INVALID_ SDRAM_PUPSD	An invalid Power Up Start Delay bit value is specified (see "ADI_EBIU_SDRAM_EBUFE" on page 4-33).
ADI_EBIU_RESULT_INVALID_ SDRAM_PSM	An invalid SDRAM Power Up Sequence bit value is specified (see "ADI_EBIU_SDRAM_PUPSD" on page 4-33).
ADI_EBIU_RESULT_INVALID_ SDRAM_EBUFE	An invalid External Buffering bit value is specified (see "ADI_EBIU_SDRAM_SRFS" on page 4-33).
ADI_EBIU_RESULT_INVALID_ SDRAM_FBBRW	An invalid Fast back-to-back read to write bit value is specified (See "ADI_EBIU_SDRAM_FBBRW" on page 4-34).
ADI_EBIU_RESULT_INVALID_ SDRAM_CDDBG	An invalid Control Disable during Bus Grant bit value is specified (See s"ADI_EBIU_SDRAM_CDDBG" on page 4-35).

adi_ebiu_GetConfigSize

Description

This function returns the number of bytes required to save the current configuration data. This value is also available via the `ADI_EBIU_SIZEOF_CONFIG` macro.

Prototype

```
size_t adi_ebiu_GetConfigSize(void);
```

Return Value

The size of the configuration structure.

adi_ebiu_Init

Description

This function initializes the EBIU module. Currently, the module is configured to handle only the SDRAM Controller. Thus, the `adi_ebiu_Init` function sets up the `EBIU_SDGCTL`, `EBIU_SDBCTL`, and `EBIU_SDRRC` registers to reflect the correct SDRAM configuration attached to the processor.

The following values are required to be set for successful initialization:

Description	Command	Value Type
Bank Size	`ADI_EBIU_CMD_SET_SDRAM_BANK_SIZE`	`ADI_EBIU_SDRAM_BANK_VALUE`
Bank column address width	`ADI_EBIU_CMD_SET_SDRAM_BANK_COLUMN_WIDTH`	`ADI_EBIU_SDRAM_BANK_VALUE`
CAS[1] latency threshold (MHz)	`ADI_EBIU_CMD_SET_SDRAM_CL_THRESHOLD`	`u16`
Minimum TRAS[2] (ns)	`ADI_EBIU_CMD_SET_SDRAM_TRASMIN`	`u16`
Min. TRP[3] (ns)	`ADI_EBIU_EBIU_CMD_SET_SDRAM_TRPMIN`	`u16`
Min. TRCD[4] (ns)	`ADI_EBIU_CMD_SET_SDRAM_TRCDMIN`	`u16`
Min. TWR[5] (cycles, ns)	`ADI_EBIU_CMD_SET_SDRAM_TWRMIN`	`ADI_EBIU_TIMING_VALUE`
Refresh period (cycles, ms)	`ADI_EBIU_CMD_SET_SDRAM_REFRESH`	`ADI_EBIU_TIMING_VALUE`

1 Column Address Strobe

2 Required delay between issuing a `Bank Activate` command and a `Precharge` command, and between the `Self-Refresh` command and the exit from Self-Refresh.

3 Required delay between issuing a `Precharge` command and the `Bank Activate`, `Auto-Refresh`, or `Self-Refresh` commands.

4 Required delay between issuing a `Bank Activate` command and the start of the first read/write command.

5 Required delay between a `Write` command and a `Precharge` command.

Upon successful initialization of the module, subsequent calls to adi_ebiu_AdjustSDRAM adjust the SDRAM refresh rate in the EBIU_SDRRC register to correspond with the given system clock frequency. If the power management module has been initialized prior to calling adi_ebiu_Init, then the AdjustRefreshRate flag can be set to TRUE to instruct the function to initialize the SDRAM refresh rate to correspond to the current value of SCLK. If not then, the SDRAM Controller is returned to reset values.

When multiple banks are used, the ADI_EBIU_CMD_SET_SDRAM_BANK_SIZE and ADI_EBIU_CMD_SET_SDRAM_BANK_COL_WIDTH command-value pairs must be specified for each bank.

The criteria described above are also met if either the Micron memory module and total size per bank or a particular EZ-Kit is specified. (See "ADI_EBIU_SDRAM_EZKIT" on page 4-28 for further details).

If the system configuration makes use of low power (2.5V) SDRAM, the following values also need to be initialized:

Description	Command	Value Type
Extended Mode Register Enable	ADI_EBIU_CMD_SET_SDRAM_EMREN	ADI_EBIU_SDRAM_SDRAM_EMREN
Partial Array Self-Refresh	ADI_EBIU_CMD_SET_SDRAM_PASR	ADI_EBIU_SDRAM_PASR
Temperature Compensated Self-Refresh	ADI_EBIU_CMD_SET_SDRAM_TCSR	ADI_EBIU_SDRAM_TCSR

Additional command-value pairs can be passed to the adi_ebiu_Init function. Alternatively, they can be set with a call to the adi_ebiu_Control function.

The adi_ebiu_Init function should only be called once, prior to adjusting the power management settings, so that the SDRAM is adjusted according to changes in SCLK. Subsequent calls to the function are ignored.

Prototype

```
ADI_EBIU_RESULT adi_ebiu_Init(
        const ADI_EBIU_COMMAND_PAIR *ConfigData,
        const u16 AdjustRefreshRate
);
```

Arguments

ConfigData	The address of a table of command-value pairs as defined by "ADI_EBIU_COMMAND" on page 4-24 and "Command Value Enumerations" on page 4-28. The last command in the table must be the ADI_EBIU_CMD_END command.
AdjustRefreshRate	A u16 value to determine whether the SDRAM refresh rate is to be updated according to the current value of the SCLK frequency.

Return Value

In debug mode, the returned values are:

ADI_EBIU_RESULT_BAD_COMMAND	A command-value pair is invalid.
ADI_EBIU_RESULT_FAILED	Not all required items are initialized.
ADI_EBIU_RESULT_CALL_IGNORED	This EBIU module is already initialized.
ADI_EBIU_RESULT_INVALID_SDRAM_SCTLE	An invalid SCTLE value specified.
ADI_EBIU_RESULT_INVALID_EZKIT	An invalid EZ-KIT type is specified.

`ADI_EBIU_RESULT_INVALID_SDRAM_MODULE`	An invalid memory module type is specified.
`ADI_EBIU_RESULT_INVALID_SDRAM_BANK_SIZE`	An invalid bank size is specified.
`ADI_EBIU_RESULT_INVALID_SDRAM_COL_WIDTH`	An invalid Column Address width is specified.
`ADI_EBIU_RESULT_INVALID_SDRAM_TWRMIN`	An invalid `TWRMIN` value is specified.
`ADI_EBIU_RESULT_INVALID_SDRAM_EMREN`	An invalid `EMREN` value is specified.
`ADI_EBIU_RESULT_INVALID_SDRAM_PASR`	An invalid `PASR` value is specified.
`ADI_EBIU_RESULT_INVALID_SDRAM_TCSR`	An invalid `TCSR` value is specified.

adi_ebiu_LoadConfig

Description

This function restores the current configuration values from the memory location pointed to by the `hConfig` argument. The SDRAM controller is reset.

Prototype

```
ADI_EBIU_RESULT adi_ebiu_LoadConfig(
                ADI_EBIU_CONFIG_HANDLE hConfig,
                size_t szConfig
);
```

Argument

`hConfig`	The address of the memory area from which the current configuration is to be stored.
`szConfig`	Number of bytes available at the given address. Must be greater than or equal to the `adi_ebiu_GetConfigSize()` return value.

Return Value

In the debug variant of the library, one of the following values is returned. Otherwise `ADI_EBIU_RESULT_SUCCESS` is returned.

`ADI_EBIU_RESULT_SUCCESS`	This process completed successfully.
`ADI_EBIU_RESULT_NO_MEMORY`	The `szConfig` value is too small.
`ADI_EBIU_RESULT_CALL_IGNORED`	The SDRAM has not been successfully initialized.

adi_ebiu_SaveConfig

Description

This function stores the current settings into the memory area pointed to by the `hConfig` argument. Currently, only the SDRAM configuration is saved.

Prototype

```
ADI_EBIU_RESULT adi_ebiu_SaveConfig(
ADI_EBIU_CONFIG_HANDLE hConfig,
size_t szConfig
);
```

Argument

`hConfig`	The address of the memory location into which the current configuration is to be stored.
`szConfig`	Number of bytes available at the given address. Must be greater than or equal to the `adi_ebiu_GetConfigSize()` return value.

Return Value

In the debug variant of the library, one of the following values is returned. Otherwise `ADI_EBIU_RESULT_SUCCESS` is returned.

`ADI_EBIU_RESULT_SUCCESS`	This process completed successfully.
`ADI_EBIU_RESULT_NO_MEMORY`	The `szConfig` value is too small.
`ADI_EBIU_RESULT_CALL_IGNORED`	The SDRAM has not been successfully initialized.

Public Data Types and Enumerations

This section provides descriptions of the public data types and enumerations.

ADI_EBIU_RESULT

All public EBIU module functions return a result code of the enumeration type, `ADI_EBIU_RESULT`. Possible values are:

`ADI_EBIU_RESULT_SUCCESS`	Generic success.
`ADI_EBIU_RESULT_FAILED`	Generic failure.
`ADI_EBIU_RESULT_BAD_COMMAND`	Invalid control command is specified.
`ADI_EBIU_RESULT_CALL_IGNORED`	A function call has been ignored with no action taken, as the module has not been initialized.
`ADI_EBIU_RESULT_INVALID_SDRAM_EBE`	Invalid value for the `EBE` field of the `EBIU_SDBCTL` register is specified.
`ADI_EBIU_RESULT_INVALID_SDRAM_BANK_SIZE`	Invalid value for the `EBSZ` field of the `EBIU_SDBCTL` register is specified.
`ADI_EBIU_RESULT_INVALID_SDRAM_COL_WIDTH`	Invalid value for the `EBCAW` field of the `EBIU_SDBCTL` register is specified.
`ADI_EBIU_RESULT_INVALID_SDRAM_CDDBG`	Invalid value for the `CDDBG` field of the `EBIU_SDGCTL` register is specified.
`ADI_EBIU_RESULT_INVALID_SDRAM_EBUFE`	Invalid value for the `EBUFE` field of the `EBIU_SDGCTL` register is specified.
`ADI_EBIU_RESULT_INVALID_SDRAM_EMREN`	Invalid value for the `EMREN` field of the `EBIU_SDGCTL` register is specified.
`ADI_EBIU_RESULT_INVALID_SDRAM_FBBRW`	Invalid value for the `FBBRW` field of the `EBIU_SDGCTL` register is specified.
`ADI_EBIU_RESULT_INVALID_SDRAM_PASR`	Invalid value for the `PASR` field of the `EBIU_SDGCTL` register is specified.
`ADI_EBIU_RESULT_INVALID_SDRAM_PSM`	Invalid value for the `PSM` field of the `EBIU_SDGCTL` register is specified.
`ADI_EBIU_RESULT_INVALID_SDRAM_PUPSD`	Invalid value for the `PUPSD` field of the `EBIU_SDGCTL` register is specified.
`ADI_EBIU_RESULT_INVALID_SDRAM_SRFS`	Invalid value for the `SRFS` field of the `EBIU_SDGCTL` register. is specified

ADI_EBIU_RESULT_INVALID_ SDRAM_TCSR	Invalid value for the TCSR field of the EBIU_SDGCTL register is specified.
ADI_EBIU_RESULT_INVALID_ SDRAM_TWRMIN	An invalid value for TWRMIN is specified and would cause TWR to be greater than 3.
ADI_EBIU_RESULT_NO_MEMORY	Insufficient memory to load/save configuration.

ADI_EBIU_SDRAM_BANK_VALUE

The ADI_EBIU_SDRAM_BANK_VALUE structure specifies the settings to be applied to a specific bank.

```
typedef struct ADI_EBIU_SDRAM_BANK_VALUE (
        u16 bank;
        Union {
              ADI_EBIU_SDRAM_BANK_SIZE size;
              ADI_EBIU_SDRAM_BANK_COL_WIDTH width;
        } u;
} ADI_EBIU_SDRAM_BANK_VALUE;
```

See “ADI_EBIU_SDRAM_BANK_SIZE” on page 4-29 and “ADI_EBIU_SDRAM_BANK_COL_WIDTH” on page 4-29 for details of the size and width fields. The bank field is intended for future use and has no meaning for the ADSP-BF531/ADSP-BF532/ADSP-BF533 and ADSP-BF534/ADSP-BF536/ADSP-BF537 Blackfin processors.

ADI_EBIU_TIME

The ADI_EBIU_TIME structure enables users to specify a timing value as an integral number of a given unit. It is defined as:

```
typedef struct ADI_EBIU_TIME {
      u32      value;
      ADI_EBIU_TIMING_UNIT units;
} ADI_EBIU_TIME;
```

Where ADI_EBIU_TIMING_UNIT is an enumeration type defined in the following table:

ADI_EBIU_TIMING_UNIT_MILLISEC	The time value specified by the associated value in the ADI_EBIU_TIME structure is in milliseconds (ms).
ADI_EBIU_TIMING_UNIT_MICROSEC	The time value specified by the associated value in the ADI_EBIU_TIME structure is in microseconds (s).
ADI_EBIU_TIMING_UNIT_NANOSEC	The time value specified by the associated value in the ADI_EBIU_TIME structure is in nanoseconds (ns).
ADI_EBIU_TIMING_UNIT_PICOSEC	The time value specified by the associated value in the ADI_EBIU_TIME structure is in picoseconds (ps).
ADI_EBIU_TIMING_UNIT_FEMTOSEC	The time value specified by the associated value in the ADI_EBIU_TIME structure is in femtoseconds (fs).

The actual values of the enumeration fields are used as factors in the integer arithmetic within the module. The millisecond value, which is used as a logic control value, is an exception, since it is not used as a factor.

Developers can use the complete range of units to enable timing values to be expressed as an unsigned 32-bit integer. For example, the SDRAM on the ADSP-BF533 EZ-Kit Lite (Rev 1.3) board has a minimum TWR value of one SCLK cycle and 7.5ns. The time value must be passed as 7500ps. Thus the ADI_EBIU_TIME value must be specified as:

```
ADI_EBIU_TIME time = {7500, ADI_EBIU_TIMING_UNIT_PICOSEC};
```

ADI_EBIU_TIMING_VALUE

Certain timing values required for the correct setting of the SDRAM control registers are specified on the appropriate data sheet as a number of SCLK cycles combined with a value expressed in one of several units (for example nanoseconds or milliseconds).

To facilitate the passing of such values to the `adi_ebiu_Init` function, the `ADI_EBIU_TIMING_VALUE` structure is defined:

```
typedef struct ADI_EBIU_TIMING_VALUE {
        u32    cycles;
        ADI_EBIU_TIME    time;
} ADI_EBIU_TIMING_VALUE;
```

Where `ADI_EBIU_TIME` is defined in "ADI_EBIU_TIME" on page 4-22.

For example, the SDRAM on the ADSP-BF533 EZ-Kit Lite (Rev 1.3) board has a minimum `TWR` value of one SCLK cycle and 7.5ns. Using the above structure, this value is expressed as:

```
ADI_EBIU_TIMING_VALUE twrmin
         = { 1, {7500, ADI_EBIU_TIMING_UNIT_PICOSEC}};
```

Setting Control Values in the EBIU Module

To set control values in the EBIU module, the user passes command-value pairs (type `ADI_EBIU_COMMAND_PAIR`) to the `adi_ebiu_Init` and `adi_ebiu_Control` functions (either individually or as a table). Note that `adi_ebiu_Init` only allows a table to be supplied. This section describes the command-value pair structure and valid commands.

ADI_EBIU_COMMAND

The `ADI_EBIU_COMMAND` is used to control/access the configuration of the EBIU Module. It is to be used in an `ADI_EBIU_COMMAND_PAIR` couplet to set a configuration value in calls to `adi_ebiu_Init` and `adi_ebiu_Control`.

Table 4-1. ADI_EBIU_COMMAND Data Values

Command	**Associated data value**
General commands used with both the adi_ebiu_Control and adi_ebiu_Init functions:	
`ADI_EBIU_CMD_END`	Defines the end of a table of command pairs.
`ADI_EBIU_CMD_SET_SDRAM_EBUFE`	An `ADI_EBIU_SDRAM_EBUFE` value to specify whether external buffers are to be used when several SDRAM devices are used. See "ADI_EBIU_SDRAM_SRFS" on page 4-33.
`ADI_EBIU_CMD_SET_SDRAM_FBBRW`	An `ADI_EBIU_SDRAM_FBBRW` value to specify whether to enable/disable fast back-to-back read/write operations. See "ADI_EBIU_SDRAM_FBBRW" on page 4-34.
`ADI_EBIU_CMD_SET_SDRAM_CDDBG`	An `ADI_EBIU_SDRAM_CDDBG` value to specify whether to enable/disable SDRAM control signals when the external memory interface is granted to an external controller. See "ADI_EBIU_SDRAM_CDDBG" on page 4-35.

Table 4-1. ADI_EBIU_COMMAND Data Values (Cont'd)

Command	Associated data value
`ADI_EBIU_CMD_SET_SDRAM_PUPSD`	An `ADI_EBIU_SDRAM_PUPSD` value specifying whether or not the power up start sequence is delayed by 15 SCLK cycles. See "ADI_EBIU_SDRAM_PUPSD" on page 4-33.
`ADI_EBIU_CMD_SET_SDRAM_PSM`	An `ADI_EBIU_SDRAM_PSM` value specifying the order of events in the power up start sequence. See "ADI_EBIU_SDRAM_PSM" on page 4-34.
Commands valid only when passed to the adi_ebiu_Init function:	
`ADI_EBIU_CMD_SET_EZKIT`	An `ADI_EBIU_EZKIT` value to identify the EZ-KIT for which the EBIU module is to be configured. See "ADI_EBIU_SDRAM_EZKIT" on page 4-28.
`ADI_EBIU_CMD_SET_SDRAM_MODULE`	An `ADI_EBIU_SDRAM_MODULE_TYPE` value containing the Micron Memory module to be configured. This value applies to all banks in use. See "ADI_EBIU_SDRAM_MODULE_TYPE" on page 4-30.
`ADI_EBIU_CMD_SET_SDRAM_BANK_SIZE`	The address of an `ADI_EBIU_SDRAM_BANK_VALUE` structure containing the bank number and the external bank size. Refer to "ADI_EBIU_SDRAM_BANK_VALUE" on page 4-21 and "ADI_EBIU_SDRAM_BANK_SIZE" on page 4-29.
`ADI_EBIU_CMD_SET_SDRAM_BANK_COL_WIDTH`	The address of an `ADI_EBIU_SDRAM_BANK_VALUE` structure containing the bank number and the external bank column address width. See "ADI_EBIU_SDRAM_BANK_VALUE" on page 4-21 and "ADI_EBIU_SDRAM_BANK_COL_WIDTH" on page 4-29.
`ADI_EBIU_CMD_SET_SDRAM_CL_THRESHOLD`	An `u32` value to specify the SCLK frequency threshold, which determines the CAS latency value to be used.

Table 4-1. ADI_EBIU_COMMAND Data Values (Cont'd)

Command	Associated data value
ADI_EBIU_CMD_SET_SDRAM_TRASMIN	An ADI_EBIU_TIME value to set the minimum TRAS value as given on the data sheet of the appropriate SDRAM. See "ADI_EBIU_TIME" on page 4-22.
ADI_EBIU_CMD_SET_SDRAM_TRPMIN	An ADI_EBIU_TIME value to set the minimum TRP value as described in the appropriate Blackfin processor data sheet of the appropriate SDRAM. See "ADI_EBIU_TIME" on page 4-22.
ADI_EBIU_CMD_SET_SDRAM_TRCDMIN	An ADI_EBIU_TIME value to set the minimum TRCD value as described in the appropriate Blackfin processor data sheet of the appropriate SDRAM. See "ADI_EBIU_TIME" on page 4-22.
ADI_EBIU_CMD_SET_SDRAM_TWRMIN	The address of an ADI_EBIU_TIMING_VALUE structure containing the minimum TWR value as described in the appropriate Blackfin processor data sheet of the appropriate SDRAM. See "ADI_EBIU_TIMING_VALUE" on page 4-23.
ADI_EBIU_CMD_SET_SDRAM_REFRESH	The address of an ADI_EBIU_TIMING_VALUE structure containing the maximum t_{REF} value as given on the data sheet of the appropriate SDRAM. See "ADI_EBIU_TIME" on page 4-22.
ADI_EBIU_CMD_SET_SDGCTL_REG	An u32 word containing the entire contents of the EBIU_SDGCTL register.
ADI_EBIU_CMD_SET_SDBCTL_REG	An u16 word containing the entire contents of the EBIU_SDBCTL register.
ADI_EBIU_CMD_SET_SDRAM_EMREN	An ADI_EBIU_SDRAM_EMREN value to specify whether low power (2.5V) SDRAM is being used. See "ADI_EBIU_SDRAM_MODULE_TYPE" on page 4-30.

Table 4-1. ADI_EBIU_COMMAND Data Values (Cont'd)

Command	Associated data value
ADI_EBIU_CMD_SET_SDRAM_PASR	An ADI_EBIU_SDRAM_PASR value to specify which banks are to be refreshed. Applicable only to low power SDRAM. See "ADI_EBIU_CMD_SET_SDRAM_SCTLE" on page 4-31.
ADI_EBIU_CMD_SET_SDRAM_TCSR	An ADI_EBIU_SDRAM_TCSR value to specify the temperature compensated self-refresh value. This command can only be used for low power SDRAM. See "ADI_EBIU_SDRAM_PASR" on page 4-32.
ADI_EBIU_CMD_SET_SDRAM_SCTLE	An ADI_EBIU_SDRAM_SCTLE value to specify whether the SDC is enabled or not. See "ADI_EBIU_CMD_SET_SDRAM_SCTLE" on page 4-31.
Commands valid only when passed to the adi_ebiu_Init function:	
ADI_EBIU_CMD_PAIR	Used to tell adi_ebiu_control that a single command pair is being passed.
ADI_EBIU_CMD_TABLE	Used to tell adi_ebiu_control that a table of command pairs is being passed.
ADI_EBIU_CMD_SET_SDRAM_ENABLE	An ADI_EBIU_SDRAM_ENABLE value to enable/disable external SDRAM. Automatically set upon initialization. See "ADI_EBIU_SDRAM_EZKIT" on page 4-28.
ADI_EBIU_CMD_SET_SDRAM_SRFS	An ADI_EBIU_SDRAM_SRFS value to enable/disable self-refresh of SDRAM during inactivity. See "ADI_EBIU_SDRAM_TCSR" on page 4-32.

ADI_EBIU_COMMAND_PAIR

The ADI_EBIU_COMMAND_PAIR data type enables developers to generate a table of control commands to pass to the EBIU via the adi_ebiu_Init and adi_ebiu_Control functions:

```
typedef struct ADI_EBIU_COMMAND_PAIR (
        ADI_EBIU_COMMAND kind;
        void *value;
} ADI_EBIU_COMMAND_PAIR;
```

Command Value Enumerations

The following enumerations are used to specify the required information to set up the SDRAM controller. For further information on the values required, refer to the *Engineer-to-Engineer Note EE-210*[*].

ADI_EBIU_SDRAM_EZKIT

This enumeration defines the EZ-KIT board for which the EBIU module is to be configured. For Blackfin processors these are:

ADI_EBIU_EZKIT_BF533	An ADSP-BF533 EZ-KIT LITE board.
ADI_EBIU_EZKIT_BF537	An ADSP-BF537 EZ-KIT LITE board.

ADI_EBIU_SDRAM_ENABLE

This enumeration specifies if SDRAM is enabled or disabled. This enumeration corresponds to the EBE bit in the EBIU_SDBCTL register.

The default value is specified by the following macro:

[*] Refer to SDRAM Selection Guidelines and Configuration for ADI Processors, EE-210, October 27, 2003.

`ADI_EBIU_SDRAM_EBE_DISABLE`	Disables SDRAM.
`ADI_EBIU_SDRAM_EBE_ENABLE`	Enables SDRAM.

```
#define ADI_EBIU_SDRAM_EBE_DEFAULT ADI_EBIU_SDRAM_EBE_DISABLE
```

ADI_EBIU_SDRAM_BANK_SIZE

This enumeration specifies the SDRAM external bank size. This enumeration corresponds to the `EBSZ` bits in the `EBIU_SDBCTL` register.

`ADI_EBIU_SDRAM_BANK_16MB`	16MB external SDRAM
`ADI_EBIU_SDRAM_BANK_32MB`	32MB external SDRAM
`ADI_EBIU_SDRAM_BANK_64MB`	64MB external SDRAM
`ADI_EBIU_SDRAM_BANK_128MB`	128MB external SDRAM

The default value is specified by the following macro:

```
#define ADI_EBIU_SDRAM_BANK_SIZE_DEFAULT ADI_EBIU_SDRAM_BANK_32MB
```

ADI_EBIU_SDRAM_BANK_COL_WIDTH

This enumeration specifies the SDRAM external bank column address width and corresponds to the `EBCAW` bits in the `EBIU_SDBCTL` register.

`ADI_EBIU_SDRAM_BANK_COL_8BIT`	8-bit external bank column address width
`ADI_EBIU_SDRAM_BANK_COL_9BIT`	9-bit external bank column address width
`ADI_EBIU_SDRAM_BANK_COL_10BIT`	10-bit external bank column address width
`ADI_EBIU_SDRAM_BANK_COL_11BIT`	11-bit external bank column address width

The default value is specified by the following macro:

```
#define ADI_EBIU_SDRAM_BANK_COL_WIDTH_DEFAULT
                      ADI_EBIU_SDRAM_BANK_COL_9BIT
```

ADI_EBIU_SDRAM_MODULE_TYPE

This enumeration specifies the SDRAM module type. The enumerator values contain the relevant information, for example speed grade and configuration settings required to initialize the SDRAM controller. Since Analog Devices EZ-Kits include SDRAM supplied by Micron, this information applies only to Micron parts. Valid values are:

`ADI_EBIU_SDRAM_MODULE_MT48LC16M4A2_6`	64Mbit, 4Meg x 4 x 4, speed grade: -6
`ADI_EBIU_SDRAM_MODULE_MT48LC8M8A2_6`	64Mbit, 2Meg x 8 x 4, speed grade: -6
`ADI_EBIU_SDRAM_MODULE_MT48LC4M16A2_6`	64Mbit, 1Meg x 16 x 4, speed grade: -6
`ADI_EBIU_SDRAM_MODULE_MT48LC16M4A2_7E`	64Mbit, 4Meg x 4 x 4, speed grade: -7E
`ADI_EBIU_SDRAM_MODULE_MT48LC8M8A2_7E`	64Mbit, 2Meg x 8 x 4, speed grade: -7E
`ADI_EBIU_SDRAM_MODULE_MT48LC4M16A2_7E`	64Mbit, 1Meg x 16 x 4, speed grade: -7E
`ADI_EBIU_SDRAM_MODULE_MT48LC16M4A2_75`	64Mbit, 4Meg x 4 x 4, speed grade: -75
`ADI_EBIU_SDRAM_MODULE_MT48LC8M8A2_75`	64Mbit, 2Meg x 8 x 4, speed grade: -75
`ADI_EBIU_SDRAM_MODULE_MT48LC4M16A2_75`	64Mbit, 1Meg x 16 x 4, speed grade: -75
`ADI_EBIU_SDRAM_MODULE_MT48LC16M4A2_8E`	64Mbit, 4Meg x 4 x 4, speed grade: -8E
`ADI_EBIU_SDRAM_MODULE_MT48LC8M8A2_8E`	64Mbit, 2Meg x 8 x 4, speed grade: -8E
`ADI_EBIU_SDRAM_MODULE_MT48LC4M16A2_8E`	64Mbit, 1Meg x 16 x 4, speed grade: -8E
`ADI_EBIU_SDRAM_MODULE_MT48LC32M4A2_6A`	128Mbit, 8Meg x 4 x 4, speed grade: -6A
`ADI_EBIU_SDRAM_MODULE_MT48LC16M8A2_6A`	128Mbit, 4Meg x 8 x 4, speed grade: -6A
`ADI_EBIU_SDRAM_MODULE_MT48LC8M16A2_6A`	128Mbit, 2Meg x 16 x 4, speed grade: -6A
`ADI_EBIU_SDRAM_MODULE_MT48LC32M4A2_7E`	128Mbit, 8Meg x 4 x 4, speed grade: -7E
`ADI_EBIU_SDRAM_MODULE_MT48LC16M8A2_7E`	128Mbit, 4Meg x 8 x 4, speed grade: -7E
`ADI_EBIU_SDRAM_MODULE_MT48LC8M16A2_7E`	128Mbit, 2Meg x 16 x 4, speed grade: -7E
`ADI_EBIU_SDRAM_MODULE_MT48LC32M4A2_75`	128Mbit, 8Meg x 4 x 4, speed grade: -75
`ADI_EBIU_SDRAM_MODULE_MT48LC16M8A2_75`	128Mbit, 4Meg x 8 x 4, speed grade: -75
`ADI_EBIU_SDRAM_MODULE_MT48LC8M16A2_75`	128Mbit, 2Meg x 16 x 4, speed grade: -75
`ADI_EBIU_SDRAM_MODULE_MT48LC32M4A2_8E`	128Mbit, 8Meg x 4 x 4, speed grade: -8E

ADI_EBIU_SDRAM_MODULE_MT48LC16M8A2_8E	128Mbit, 4Meg x 8 x 4, speed grade: -8E
ADI_EBIU_SDRAM_MODULE_MT48LC8M16A2_8E	128Mbit, 2Meg x 16 x 4, speed grade: -8E
ADI_EBIU_SDRAM_MODULE_MT48LC64M4A2_7E	256Mbit, 16Meg x 4 x 4, speed grade: -7E
ADI_EBIU_SDRAM_MODULE_MT48LC32M8A2_7E	256Mbit, 8Meg x 8 x 4, speed grade: -7E
ADI_EBIU_SDRAM_MODULE_MT48LC16M16A2_7E	256Mbit, 4Meg x 16 x 4, speed grade: -7E
ADI_EBIU_SDRAM_MODULE_MT48LC64M4A2_75	256Mbit, 16Meg x 4 x 4, speed grade: -75
ADI_EBIU_SDRAM_MODULE_MT48LC32M8A2_75	256Mbit, 8Meg x 8 x 4, speed grade: -75
ADI_EBIU_SDRAM_MODULE_MT48LC16M16A2_75	256Mbit, 4Meg x 16 x 4, speed grade: -75
ADI_EBIU_SDRAM_MODULE_MT48LC64M8A2_7E	512Mbit, 16Meg x 8 x 4, speed grade: -7E
ADI_EBIU_SDRAM_MODULE_MT48LC32M16A2_7E	512Mbit, 8Meg x 16 x 4, speed grade: -7E
ADI_EBIU_SDRAM_MODULE_MT48LC64M8A2_75	512Mbit, 16Meg x 8 x 4, speed grade: -75
ADI_EBIU_SDRAM_MODULE_MT48LC32M16A2_75	512Mbit, 8Meg x 16 x 4, speed grade: -75

ADI_EBIU_CMD_SET_SDRAM_SCTLE

This enumeration specifies if the SDRAM controller is enabled or disabled and corresponds to the `SCTLE` bit in the `EBIU_SDGCTL` register.

ADI_EBIU_SDRAM_SCTLE_DISABLE	Disable SDRAM Controller.
ADI_EBIU_SDRAM_SCTLE_ENABLE	Enable SDRAM Controller.

ADI_EBIU_SDRAM_EMREN

This enumeration specifies that low power (2.5V) SDRAM is to be used and corresponds to the `EMREN` bit in the `EBIU_SDGCTL` register:

ADI_EBIU_SDRAM_EMREN_DISABLE	Mobile low power SDRAM is not present.
ADI_EBIU_SDRAM_EMREN_ENABLE	Mobile low power SDRAM is present.

The default value is specified by the following macro:

```
#define ADI_EBIU_SDRAM_EMREN_DEFAULT ADI_EBIU_SDRAM_EMREN_DISABLE
```

ADI_EBIU_SDRAM_PASR

When low power (2.5V) SDRAM is used, this enumeration specifies which banks are to be refreshed. This enumeration corresponds to the `PASR` bits in the `EBIU_SDGCTL` register:

`ADI_EBIU_SDRAM_PASR_ALL`	All four SDRAM banks are to be refreshed.
`ADI_EBIU_SDRAM_PASR_INT01`	Internal SDRAM banks 0 and 1 are to be refreshed.
`ADI_EBIU_SDRAM_PASR_INT01_ONLY`	Internal SDRAM banks 0 and 1 only to be refreshed.

The default value is specified by the following macro:

```
#define ADI_EBIU_SDRAM_PASR_DEFAULT ADI_EBIU_SDRAM_PASR_ALL
```

ADI_EBIU_SDRAM_TCSR

When low power (2.5V) SDRAM is used, this enumeration specifies the temperature compensated self-refresh value and corresponds to the `TCSR` bits in the `EBIU_SDGCTL` register.

`ADI_EBIU_SDRAM_TCSR_45DEG`	The SDRAM banks are to be refreshed if the temperature exceeds 45 degrees Celsius.
`ADI_EBIU_SDRAM_TCSR_85DEG`	The SDRAM banks are to be refreshed if the temperature exceeds 85 degrees Celsius.

The default value is specified by the following macro:

```
#define ADI_EBIU_SDRAM_TCSR_DEFAULT ADI_EBIU_SDRAM_TCSR_45DEG
```

ADI_EBIU_SDRAM_SRFS

This enumeration specifies if the EBIU is to enable/disable SDRAM self-refresh during periods of inactivity. This enumeration corresponds to the `SRFS` bit in the `EBIU_SDGCTL` register. For example, SDRAM self-refresh is enabled when the Processor mode is put into the "deep sleep" via the Power Management Module. For more information, see "Power Management Module" on page 3-1.

`ADI_EBIU_SDRAM_SRFS_DISABLE`	Disable SDRAM self-refresh on inactivity.
`ADI_EBIU_SDRAM_SRFS_ENABLE`	Enable SDRAM self-refresh on inactivity.

The default value is specified by the following macro:

```
#define ADI_EBIU_SDRAM_SRFS_DEFAULT ADI_EBIU_SDRAM_SRFS_DISABLE
```

ADI_EBIU_SDRAM_EBUFE

This enumeration specifies whether or not the EBIU uses external buffers when several SDRAM devices are being used in parallel. This enumeration corresponds to the `EBUFE` bit in the `EBIU_SDGCTL` register.

`ADI_EBIU_SDRAM_EBUFE_DISABLE`	Disable the use of external buffers when several SDRAM devices are being used in parallel.
`ADI_EBIU_SDRAM_EBUFE_ENABLE`	Enable the use of external buffers when several SDRAM devices are being used in parallel.

The default value is specified by the following macro:

```
#define ADI_EBIU_SDRAM_EBUFE_DEFAULT ADI_EBIU_SDRAM_EBUFE_DISABLE
```

ADI_EBIU_SDRAM_PUPSD

This enumeration specifies whether or not the power-up start sequence is to be delayed by 15 SCLK cycles. This enumeration corresponds to the `PUPSD` bit in the `EBIU_SDGCTL` register.

ADI_EBIU_SDRAM_PUPSD_NODELAY	No delay to the power-up start sequence.
ADI_EBIU_SDRAM_PUPSD_15CYCLES	The power-up start sequence is to be delayed by 15 SCLK cycles.

The default value is specified by the following macro:

```
#define ADI_EBIU_SDRAM_PUPSD_DEFAULT ADI_EBIU_SDRAM_PUPSD_NODELAY
```

ADI_EBIU_SDRAM_PSM

This enumeration specifies the SDRAM power-up sequence. This enumeration corresponds to the PSM bit in the EBIU_SDGCTL register.

ADI_EBIU_SDRAM_PSM_REFRESH_FIRST	The SDC is to perform a Precharge All command, followed by eight Auto-Refresh cycles, and then a Load Mode Register command.
ADI_EBIU_SDRAM_PSM_REFRESH_LAST	The SDC performs a Precharge All command, followed by a Load Mode Register command, and then completes eight Auto-Refresh cycles.

The default value is specified by the following macro:

```
#define ADI_EBIU_SDRAM_PSM_DEFAULT ADI_EBIU_SDRAM_PSM_REFRESH_FIRST
```

ADI_EBIU_SDRAM_FBBRW

This enumeration specifies whether or not the EBIU uses fast back-to-back read-write access to allow SDRAM read and write operations on consecutive cycles. This enumeration corresponds to the FBBRW bit in the EBIU_SDGCTL register.

ADI_EBIU_SDRAM_FBBRW_DISABLE	Fast back-to-back read-write access disabled.
ADI_EBIU_SDRAM_FBBRW_ENABLE	SDRAM read and write operations to occur on consecutive cycles.

The default value is specified by the following macro:

```
#define ADI_EBIU_SDRAM_FBBRW_DEFAULT ADI_EBIU_SDRAM_FBBRW_DISABLE
```

ADI_EBIU_SDRAM_CDDBG

This enumeration enables or disables the SDRAM control signals when the external memory interface is granted to an external controller. This enumeration corresponds to the `CDDBG` bit in the `EBIU_SDGCTL` register.

`ADI_EBIU_SDRAM_CDDBG_DISABLE`	Disable the SDRAM control signals when the external memory interface is granted to an external controller.
`ADI_EBIU_SDRAM_CDDBG_ENABLE`	Enable the SDRAM control signals when the external memory interface is granted to an external controller.

The default value is specified by the following macro:

```
#define ADI_EBIU_SDRAM_CDDBG_DEFAULT ADI_EBIU_SDRAM_CDDBG_DISABLE
```

5 DEFERRED CALLBACK MANAGER

This chapter describes the Deferred Callback Manager that is used by the application developer to manage the deferred execution of function calls. Included is a detailed description of the Application Programming Interface (API) provided by the Deferred Callback Manager.

This chapter contains:

Introduction

Callback functions are commonly used in event driven applications where the client application requests that a Service Manager, such as the System Services Library's (SSL) DMA Manager, notifies it upon completion of a requested task, for example the completion of DMA transfer, by means of a Client Callback Function specified by the client application upon initialization of the required service.

The need to execute a client callback function normally occurs while executing an Interrupt Service Routine (ISR) at relatively high priority. The general rule for such ISR's is to keep the amount of time spent in them as deterministic as possible and to a minimum. Callbacks on the other hand may be both lengthy and non-deterministic. In most cases, users may prefer to defer the execution of such callbacks to a scheduler running at a lower priority, which can be preempted by higher priority interrupts. In doing so, the requesting ISR can complete with minimal delay.

The System Services Library's Deferred Callback (DCB) Manager provides such a service by managing one or more queues of deferred callbacks such that (typically) their invocation occurs within a dispatch function operating at a lower interrupt priority than the rest of the application's interrupt services. Each callback entry posted to a queue comprises the address of the required callback function along with three values (two pointers and one 32-bit unsigned integer), which are passed to the callback function upon its (deferred) execution.

The DCB Manager is designed to operate either as a standalone module or in conjunction with a real-time operating system (RTOS). Implementations of the module exist for Express Logic's ThreadX, Green Hills Software' Integrity, as well as Analog Devices VDK. The number of queues available and their length is determined by the client application upon module and queue initialization. Whether or not the DCB Manager is implemented in standalone mode or in conjunction with one of the above RTOS also impacts the number and size of queues. For instance when implemented in conjunction with VDK, the DCB Manager can only support one queue at a fixed priority level of IVG 14.

While only one queue is permitted per IVG level, engineers can set priorities for individual callback entries by supplying a software priority level upon posting. There is no limit to the number of software priority levels that can be used (except for practical implications within the limits of an unsigned short values) The dispatch function attempts to execute all higher priority callbacks before those with lower priorities at the same IVG level.

A detailed description of how the DCB Manager operates is provided in "Using the Deferred Callback Manager" on page 5-3, along with code segments illustrating its use in standalone mode, and the implications for its use in conjunction with an RTOS are given in Interoperability With an RTOS.

The DCB manager uses an unambiguous naming convention to safeguard against conflicts with other software libraries provided by ADI or other companies. As a result, all enumeration values and `typedefs` use the `ADI_DCB_` prefix, while functions and global variables use the lower case, `adi_dcb_`, equivalent.

Using the Deferred Callback Manager

The operation of the DCB Manager comprises the following operations.

- Setting up the DCB Manager
    - Initializing the DCB Manager
    - Opening a queue

- Managing the queue
    - Posting callbacks to the required queue
    - Dispatching callbacks according to the priority level determined upon posting.
- Performing housekeeping functions
    - Closing the queue
    - Terminating the DCB Manager

Exactly how this is implemented depends on whether the DCB manager is used in standalone mode or in conjunction with the deferred calling mechanism supplied by an RTOS. In all cases the API calls to the DCB Manager are the same: A queue is initialized with a call to `adi_dcb_Open`, and callbacks added to the queue via a call to the `adi_dcb_Post` function.

The deferred execution of the callbacks is scheduled according to software priority by the `adi_dcb_Dispatch_Callbacks` function. In the standalone environment, the DCB Manager registers this function as an Interrupt Handler routine against the desired IVG level, using the System Services Library's Interrupt Manager module, when the queue is initialized, and an interrupt raised each time a callback is posted. Since the standalone version uses the Interrupt Manager, the Interrupt Manager must be initialized before the DCB Manager is initialized.

The following code sample demonstrates the standalone use of one queue initialized at IVG level 14, which is the lowest IVG level available at application level.

As mentioned above, for standalone operation we need to initialize the interrupt manager prior to initializing the DCB Manager. On the assumption that the sample application requires only one interrupt handler to be defined per IVG level, initialize the Interrupt Manager using the following code:

```
u32 ne;
adi_int_Init(NULL,0,&ne,NULL);
```

Initialize the DCB manager with sufficient memory for one queue as follows:

```
static char mjk_dcb_Data[ADI_DCB_QUEUE_SIZE];
:
u32 ns;
:
adi_dcb_Init(
    (void*)mjk_dcb_Data,    // Address of memory to be used
    ADI_DCB_QUEUE_SIZE,     // Number of bytes required for the
                            // required number of queue servers.
    &ns                     // on return this should be the same
                            // as the required number of queues.
    NULL                    // No special data area for critical
                            // region required
);
```

Next, open the queue server for use by passing sufficient memory for the length of queue required (five entries in this case), and the desired IVG level at which the queue operates. This level is ignored if it is used within a VDK-based application. A handle, `p_DCB_handle`, to the queue server is returned:

```
static char mjk_dcb_QueueData[5*ADI_DCB_ENTRY_SIZE];
ADI_DCB_HANDLE p_DCB_handle;
:
```

```
u32 nqe;
:
adi_dcb_Open(
    14,                         // required IVG level
    (void*) mjk_dcb_QueueData, // Address of memory to be used
    5*ADI_DCB_ENTRY_SIZE,      // for a queue 5 deep.
    &nqe;                      // on return this should be the
                               // same as the required number of
                               // entries (5 in this case).
    &p_DCB_handle              // returned handle to queue server
    );
```

The DCB Manager is now ready to accept callback postings to the queue server. Note that this function is normally performed in an ISR of another Service. The DCB Manager passes the address of the client callback function and its associated argument values to the queue server identified by the handle obtained:

```
adi_dcb_Post(
    p_DCB_handle,         // handle to required queue server.
    0,                    // Priority level.
    ClientCallback,       // Address of callback function.
    pService,             // Address of the service instance
                          // that is posting the callback.
    event,                // Flag identifying the event that
                          // has precipitated the interrupt.
    (void*)data           // Address of data relevant to the
                          // callback.
);
```

Where `event` typically defines some event, for example DMA completion, and `data` typically points to an appropriate location in memory meaningful within the context of the callback function. Within the DMA Manager context, this argument is either the address of an appropriate descriptor or data buffer.

If, for any reason, users want to flush the queue of entries for the above callback, this can be achieved in one of two ways. Either users can call the `adi_dcb_Remove` function directly or call it indirectly using the `adi_dcb_Control` function. See `adi_dcb_Terminate` for further details and an example of its use), along with any other requests. The following code describes the direct approach:

```
adi_dcb_Remove(
    p_DCB_handle,            // handle to required queue server
    ClientCallback           // Address of callback function to
                             // flush
);
```

Finally, if required, the queue can be closed and the DCB Manager terminated:

```
adi_dcb_Close(
    p_DCB_handle,        // handle to required queue server
);
adi_dcb_Terminate();
```

Interoperability With an RTOS

The DCB manager employs two functions, `adi_dcb_RegisterISR` and `adi_dcb_Forward`, to interface with the different RTOS environments, including standalone mode.

These functions are supplied in a separate source file, `adi_dcb_xxxx.c` for each implementation where `xxxx` describes either the required RTOS (for example, `threadx` for Express Logic's `ThreadX`, and `integrity` for Green Hill Software's Integrity), or `standalone` for standalone use. VDK support is achieved with the functions described above supplied directly by VDK. As a result, there is no equivalent `adi_dcb_vdk.c` file. The relevant `adi_` dcb_xxxx.c file is incorporated (or not) into the main adi_dcb.c source file via conditional compilation governed by a macro, ADI_SSL_XXXX, where XXXX is STANDALONE, THREADX, INTEGRITY or VDK.

Currently implementations of the DCB Manager are only provided for the environments previously described. To implement these functions under an alternative RTOS (for example, Linux), developers must provide replacement definitions in equivalent files.

These functions are now described in more detail:

adi_dcb_Forward

The adi_dcb_Forward function takes two arguments. The first is a pointer to the DCB entry header structure, ADI_DCB_ENTRY_HDR, and the second is to the IVG level of the appropriate queue.

The adi_dcb_Forward function is invoked from within the adi_dcb_Post function and has the following prototype:

```
void adi_dcb_Forward(
     ADI_DCB_ENTRY_HDR *Entry,
     u16 IvgLevel
     );
```

The arguments are as follows:

Entry	Pointer to the ADI_DCB_ENTRY_HDR structure. This is coincident with the address of the queue server structure to which the callback is to be posted. Ignored in standalone mode.
IvgLevel	The IVG level of the appropriate queue. This argument is ignored by VDK.

The ADI_DCB_ENTRY_HDR structure used to pass information to the underlying RTOS is defined as:

```
typedef struct ADI_DCB_ENTRY_HDR {
        struct ADI_DCB_ENTRY_HDR *pNext;
        ADI_DCB_DEFERRED_FNpDeferredFunction;
} ADI_DCB_ENTRY_HDR;
```

The first word in this structure, pNext, is NULL on entry to the adi_dcb_Forward function. While this value is typically used to point to the next item in the queue, its interpretation within the adi_dcb_Forward function is wholly dependent on the specific RTOS implementation required. The second word, pDeferredFunction, is set to point to the adi_dcb_DispatchCallbacks function when the queue is initialized. The Deferred Procedure Call server within the appropriate RTOS must pass the pointer to this structure to the adi_dcb_DispatchCallbacks function upon its deferred execution.

adi_dcb_RegisterISR

The adi_dcb_RegisterISR function is invoked from within the adi_dcb_Open function and has the following prototype:

```
   void adi_dcb_RegisterISR(
        u16 IvgLevel,
        ADI_INT_HANDLER_FN Dispatcher,
        ADI_DCB_HANDLE *hServer
        );
```

The data types are defined in the <services/services.h> header file and the arguments are as follows:

IvgLevel	The interrupt level at which callbacks are to be dispatched.

`Dispatcher`	This must be the address of the `adi_dcb_DispatchCallbacks` function.
`hServer`	The address of the queue server structure.

In the standalone implementation this function registers the `adi_dcb_DispatchCallbacks` function with the interrupt manager at the specified interrupt level. In the VDK implementation, it returns with no effect.

Handling Critical Regions within Callbacks

If within a callback function critical regions are required, users should be aware of any restrictions the underlying RTOS imposes. For example, VDK based applications are prohibited from calling `PushCriticalRegion/PopCriticalRegion` functions from within interrupt level. If the VDK version of the DCB Manager is used these kinds of calls can be used, as the callback is executed at kernel level. However, if the standalone version of the library is used to run a DCB queue at a higher priority than the VDK DPC queue, such calls are illegal since the callback executes at the interrupt level. In these cases, effect critical regions directly, for example, by using the `cli()`, `sti()` built-ins.

DCB Manager API Reference

This section provides descriptions of the DCB Manager API functions.

Notation Conventions

The reference pages for the API functions use the following format:

Name and purpose of the function

Description – Function specification

Prototype – Required header file and functional prototype

Arguments – Description of function arguments

Return Value – Description of function return values

adi_dcb_Close

Description

This function closes the DCB queue server identified by the single handle argument, freeing up the slot for subsequent use. In standalone mode, the DCB Manager's `adi_dcb_DispatchCallbacks` function is unhooked from the interrupt handler chain for the given IVG Level.

Prototype

```
ADI_DMA_RESULT
adi_dcb_Close(
      ADI_DCB_HANDLE hServer
);
```

Arguments

`hServer`	The handle of the required queue server to be closed.

Return Value

In debug mode this routine returns `ADI_DCB_RESULT_NO_SUCH_QUEUE` if the handle provided does not represent a valid queue server registered with the DCB Manager. Otherwise `ADI_DCB_RESULT_SUCCESS` is returned.

`ADI_DCB_RESULT_SUCCESS`	Queue successfully closed.
`ADI_DCB_RESULT_NO_SUCH_QUEUE`	The handle provided does not represent a valid queue server.
`ADI_DCB_RESULT_QUEUE_IN_USE`	Callbacks are on the queue awaiting dispatch. If this does not matter, then flush the queue first before closing.

adi_dcb_Control

Description

This function is used to configure/control a deferred callback queue server according to command-value pairs (see "ADI_DCB_COMMAND_PAIR" on page 5-25). Currently, only one command is relevant, `ADI_DCB_CMD_FLUSH_QUEUE`, though others may be added in the future. The command-value pairs can be specified in one of three ways:

- A single command-value pair is passed:

```
adi_dcb_Control(
        hServer,
        ADI_DCB_CMD_FLUSH_QUEUE,
        (void*)ClientCallback
);
```

- A single command-value pair structure is passed, for example:

```
ADI_DCB_COMMAND_PAIR cmd=
        {ADI_DCB_CMD_FLUSH_QUEUE, (void *)ClientCallback};
adi_dcb_Control(
        hServer,
        ADI_DCB_CMD_PAIR,
        (void*)&cmd
);
```

- A table of `ADI_DCB_COMMAND_PAIR` structures is passed. The last entry in the table must be `ADI_DCB_CMD_END`:

  ```
  ADI_DCB_COMMAND_PAIR table[2] = {
          {ADI_DCB_CMD_FLUSH_QUEUE, (void*)ClientCallback,
          {ADI_DCB_CMD_END, 0}
  );

  adi_dcb_Control(
          hServer,
          ADI_DCB_CMD_TABLE,
          (void*)table

  );
  ```

Refer to "ADI_DCB_COMMAND" on page 5-26 for the complete list of commands and associated values.

Prototype

```
ADI_DMA_RESULT
adi_dcb_Control(
        ADI_DCB_HANDLE hServer,
        ADI_DCB_COMMAND command,
        void *value
);
```

Arguments

`hServer`	This is the handle of the required queue server to be closed.
`command`	This is an `ADI_DCB_COMMAND` enumeration value specifying the meaning of the associated value argument (see "ADI_DCB_COMMAND" on page 5-26).
`value`	This is the required value. A single value or a command-value pair or a table of command-value pairs.

Return Value

In debug mode, this routine returns one of the following result codes, otherwise `ADI_DCB_RESULT_SUCCESS` is always returned:

`ADI_DCB_RESULT_NO_SUCH_QUEUE`	This is the handle of the required queue server is invalid.
`ADI_DCB_RESULT_BAD_COMMAND`	Either the command kind or the value specified is invalid.

adi_dcb_Init

Description

The `adi_dcb_Init` function initializes the DCB Manager with sufficient memory for the required number of Deferred Callback Queues (referred to as Queue Servers).

This function can be called once per processor core.

Prototype

```
ADI_DCB_RESULT
adi_dcb_Init(
        void            *ServerMemData,
        size_t          szServer,
        unsigned int    *NumServers
        void            *hCriticalRegionData
);
```

Arguments

`ServerMemData`	This is the pointer to an area of memory, which is used to hold the data associated with each registered queue server.
`szServer`	This is the length in bytes of memory being supplied for the queue server data
`NumServers`	On return, this argument holds the maximum number of simultaneously open queue servers that the supplied memory can support.
`hCriticalRegionData`	This is the handle to data area containing critical region data. This will be passed to `adi_int_EnterCriticalRegion` where used internally of the module. See "Interrupt Manager" on page 2-1 for further details.

Return Value

In debug mode this routine returns one of the following result codes, otherwise `ADI_DCB_RESULT_SUCCESS` is always returned:

`ADI_DCB_RESULT_SUCCESS`	Successfully initialized the queue server
`ADI_DCB_RESULT_NO_MEMORY`	Insufficient memory for one queue entry was encountered.
`ADI_DCB_RESULT_CALL_IGNORED`	The DBG Manager has already been initialized for this processor core.

adi_dcb_Open

Description

The `adi_dcb_Open` function opens a queue server for use by assigning memory for its callback queue. Additionally, in standalone mode, the queue is assigned to the requested IVG priority level and the DCB Manager's `adi_dcb_DispatchCallbacks` function is hooked to the interrupt handler chain with the Interrupt Manager for the given IVG Level.

ⓘ Users must initialize the Interrupt Manager prior to opening a queue server.

Prototype

```
ADI_DCB_RESULT adi_dcb_Open (
      u32               IvgLevel,
      void              *QueueMemData,
      size_t            szQueue,
      u32               *NumEntries,
      ADI_DCB_HANDLE    *hServer
);
```

Arguments

`IvgLevel`	The IVG level at which the DCB Manager's dispatcher function is to operate. This value is ignored in the VDK version of the library.
`QueueMemData`	This is the pointer to an area of memory, which is used to hold the data associated with the server's entry queue.
`szQueue`	This is the length in bytes of memory being supplied for the queue.
`NumEntries`	On return, this argument holds the maximum number of queue entries that the supplied memory can support.
`hServer`	On return, this argument contains a handle to the queue server opened. This should be used to uniquely identify the queue server in calls to other API functions within the SSL.

Return Value

In debug mode, this routine returns one of the following result codes, otherwise `ADI_DCB_RESULT_SUCCESS` is always returned.

`ADI_DCB_RESULT_SUCCESS`	The queue server was successfully initialized.
`ADI_DCB_RESULT_NO_MEMORY`	Insufficient memory for one queue entry was encountered.
`ADI_DCB_RESULT_QUEUE_IN_USE`	A queue server has already been opened for use by the specified IVG.

adi_dcb_Post

Description

This function posts a callback function and associated argument values to the queue server, identified by the handle argument for further processing.

A callback is associated with a priority level, so that higher priority callbacks run before lower priority ones. To run all callbacks at the same priority level, assign the same priority to each callback posted.

Prototype

```
ADI_DCB_RESULT adi_dcb_Post(
      ADI_DCB_HANDLE       *hServer,
      u16                  Priority;
      ADI_DCB_CALLBACK_FN  Callback,
      void                 *pHandle,
      u32                  u32Arg,
      void                 *pArg
);
```

Arguments

`hServer`	This is the handle of the required queue server.
`Priority`	This is the priority level at which the callback is to run, the lower the number the higher the priority. There is no real limit on the value supplied.
`Callback`	This is the address of the client callback function to be queued.

pHandle	This is the a void* address which is to be passed as the first argument to the callback function upon its deferred execution. Typically it will be a handle address that is meaningful within the context of the callback function. For example, when used within the interrupt handler of the DMA manger this argument is the ClientHandle value defined when the DMA channel was opened.
u32Arg	A u32 value which is to be passed as the second argument to the callback function upon its deferred execution (see "ADI_DCB_CALLBACK_FN" on page 5-24). Typically, it will be a value that is meaningful within the context of the callback function. For example, when used within the interrupt handler of the DMA manger this argument describes the nature of the event that has occurred.
pArg	The void* value which is to be passed as the third argument to the callback function (see "ADI_DCB_CALLBACK_FN" on page 5-24) upon its deferred execution. Typically, it will be an address of a block of data. For example, when called within the interrupt handler of the DMA Manager this argument points to the start of the buffer for which the DMA transfer has completed.

Return Value

In debug mode, this routine returns one of the following result codes; otherwise, ADI_DCB_RESULT_SUCCESS is always returned.

ADI_DCB_RESULT_SUCCESS	The entry was successfully queued.
ADI_DCB_RESULT_NO_MEMORY	There is no vacant queue entry available.
ADI_DCB_RESULT_NO_SUCH_QUEUE	The handle provided does not represent a valid queue server.

adi_dcb_Remove

Description

This function removes entries in the given queue that matches the address of the given callback function. Alternatively, passing a NULL value for the callback function address, instructs the callback manager to remove all entries in the queue.

Prototype

```
ADI_DCB_RESULT adi_dcb_Remove(
      ADI_DCB_HANDLE        hServer,
      ADI_DCB_CALLBACK_FN   Callback
);
```

Arguments

`hServer`	This is the handle of the required queue server.
`Callback`	This is the address of the client callback function to be removed. If NULL then all entries in the queue will be removed, otherwise all entries matching the given callback function address is removed.

Return Value

In debug mode this routine returns one of the following result codes, otherwise `ADI_DCB_RESULT_FLUSHED_OK` is always returned:

`ADI_DCB_RESULT_FLUSHED_OK`	Entries were successfully removed.
`ADI_DCB_RESULT_NONE_FLUSHED`	The routine found no entries to be removed.
`ADI_DCB_RESULT_NO_SUCH_QUEUE`	The handle provided does not represent a valid queue server.

adi_dcb_Terminate

Description

This function terminates the DCB Manager by dissociating the supplied memory (see `adi_dcb_Init`) and Critical region data.

Prototype

```
ADI_DCB_RESULT
adi_dcb_Terminate ( void );
```

Return Value

`ADI_DCB_RESULT_SUCCESS` is always returned.

Public Data Types and Macros

This section provides descriptions of the public data types and macros.

ADI_DCB_CALLBACK_FN

The `ADI_DCB_CALLBACK_FN typedef` defines the prototype for the callback functions to be posted:

```
typedef void (*ADI_DCB_CALLBACK_FN) (void* pHandle, u32 u32Arg,
void* pArg);
```

Where the values of the arguments are those passed to the `adi_dcb_Post` function when the callback is queued for deferred execution.

ADI_DCB_COMMAND_PAIR

This data type is used to enable the generation of a table of control commands to be sent to the DCB Manager via the `adi_dcb_Control` function.

```
typedef struct ADI_DCB_COMMAND_PAIR {
   ADI_DCB_COMMAND kind;
   void *value;
} ADI_DCB_COMMAND_PAIR;
```

For valid values for the `kind` field refer to "ADI_DCB_COMMAND" on page 5-26. For example, the following command could be sent to the DCB Manager to flush all callbacks in the queue:

```
ADI_DCB_COMMAND_PAIR CMD = { ADI_DCB_CMD_FLUSH_QUEUE, NULL };
```

ADI_DCB_COMMAND

The `ADI_DCB_COMMAND` is used to control the DCB Manager's queue server. This data type is used in an `ADI_DCB_COMMAND_PAIR` couplet to change a configuration value in calls to `adi_dcb_Control`.

Command	Associated Data Value
`ADI_DCB_CMD_END`	This command defines the end of a table of command pairs.
`ADI_DCB_CMD_PAIR`	This command is used to tell `adi_dcb_Control` that a single command pair is being passed.
`ADI_DCB_CMD_TABLE`	This command is used to tell `adi_dcb_Control` that a table of command pairs is being passed.
`ADI_DCB_CMD_FLUSH_QUEUE`	The address of the callback function for which all matching queue entries are cleared from the queue regardless of priority.

ADI_DCB_ENTRY_HDR

The `ADI_DCB_ENTRY_HDR` structure is provided to interface with the underlying RTOS through the `adi_dcb_Forward` function (see "adi_dcb_Forward" on page 5-8):

```
typedef struct ADI_DCB_ENTRY_HDR (
   struct ADI_DCB_ENTRY *pNext;           // Next item in queue
   ADI_DCB_DEFERRED_FN pDeferredFunction; // Deferred Callback
                                          // Function pointer,
} ADI_DCB_ENTRY_HDR;
```

Where `pNext` points to the next item in the queue and `pDeferredFunction` is the address of the deferred function, which is always the address of `adi_dcb_DispatchCallbacks`.

The `ADI_DCB_DEFERRED_FN typedef` defines the prototype for this function:

```
typedef void (*ADI_DCB_DEFERRED_FN) (ADI_DCB_ENTRY *);
```

ADI_DCB_RESULT

All public DCB Manager functions return a result code of the `ADI_DCB_RESULT` data type. Possible values are:

`ADI_DCB_RESULT_SUCCESS`	The queue server was successfully initialized.
`ADI_DCB_RESULT_NO_MEMORY`	Insufficient memory for one queue entry was present.
`ADI_DCB_RESULT_QUEUE_IN_USE`	A queue server has already been opened for use by the (see "ADI_DCB_COMMAND" on page 5-26) specified IVG.
`ADI_DCB_RESULT_CALL_IGNORED`	The DBG Manager has already been initialized for this (see "ADI_DCB_COMMAND" on page 5-26) processor core.
`ADI_DCB_RESULT_NO_SUCH_QUEUE`	The handle provided does not represent a valid queue server registered with the DCB Manager.
`ADI_DCB_RESULT_BAD_COMMAND`	Either the command kind or the value specified is invalid.

6 DMA MANAGER

This chapter describes Direct Memory Access (DMA) Manager features and Application Programming Interface (API).

This chapter contains:

Introduction

The DMA Manager provides the application developer with the means to manage DMA traffic on as many channels as required across the spectrum—from setting up the DMA channels for their intended purpose, to providing callbacks, to the client application on transfer completion.

As part of the System Services, the DMA Manager provides a complete and easy-to-use interface to the DMA controller. To this end, the DMA Manager is designed to:

- Remove the need for direct client access to memory mapped registers (MMR) through the implementation of API function calls.
- Place no limitations on the type of data transfer—all descriptor types are supported as well as both single and circular buffers. Both one-dimensional (1D) or two-dimensional (2D) DMA can be used.

- Provide a simple interface to perform block copies of data between different memory locations using both 1D and 2D Memory DMA, such that blocks of data can be copied between internal and external memory with one function call in an equivalent manner to the C library `memcpy` function.
- Interpret interrupts raised on DMA transfer completion and pass higher-level event information to the user-supplied callback functions. For example, if an interrupt is raised on each inner loop of a circular 2D DMA transfer, an event can be passed to the callback function at the completion of each inner loop.
- Minimize the memory used by the module. No static memory space is set aside within the API framework to hold the configuration details for each channel. Instead, a mechanism is provided to enable client applications to set-aside sufficient memory for as many DMA channels as required by the application.
- Be as portable as possible by providing a consistent interface across all processor families and variants. Additionally, the DMA Manager uses an unambiguous naming convention to safeguard against conflicts with other software libraries provided by Analog Devices, Inc. or elsewhere.

  To this end, all enumeration values and `typedefs` use the `ADI_DMA_` prefix, while functions and global variables use the lower case, `adi_dma_`, equivalent.

Theory of Operation

This section describes the internal operation of the DMA Manager.

Overview

The DMA Manager is used to control the Blackfin DMA controller. The DMA Manager supports both peripheral DMA for moving data to and from the various on-board peripherals, and memory DMA for moving data between the various memory spaces of the Blackfin processor.

The DMA Manager is capable of controlling any number of DMA channels. The user can specify which channels the DMA Manager should control. Remaining channels not under control of the DMA Manager can be used for any purpose and can be controlled independently of the DMA Manager by the application.

Various data transfer modes of the Blackfin DMA controller are supported including descriptor chains, circular buffers (utilizing the autobuffer capability of the Blackfin processor), and one-shot transfers. Both one-dimensional or linear transfers are supported as are two-dimensional or matrix transfers.

The DMA Manager can be directed to notify the client, via the client's callback function, when data transfers complete. Additionally the client's callback function is invoked when unexpected events, such as DMA errors, occur. As with all system services, the DMA Manager allows the client to specify callbacks to be "live", meaning the client's callback function is invoked at hardware interrupt time, or "deferred", meaning the client's callback function is invoked outside the context of the hardware interrupt.

Initialization

In order to use the DMA Manager, the client must first initialize it. The DMA Manager does not use any static data, so the initialization step is used to give the DMA Manager memory that it can use to manage the DMA controller.

The DMA Manager requires a small fixed amount of base memory and then a variable amount of memory, depending on how many simultaneously open DMA channels the system requires. Note that memory DMA requires two DMA channels—one channel for the source and another channel for the destination for each memory DMA stream. Macros are provided to define the amount of memory (in bytes) that are required for the base and channel memory. These macros are `ADI_DMA_BASE_MEMORY` and `ADI_DMA_CHANNEL_MEMORY`. For instance, if the client wanted to initialize the DMA Manager and would have at most four DMA channels and one memory DMA stream open simultaneously, the amount of memory that would be required is:

```
(ADI_DMA_BASE_MEMORY + (ADI_DMA_CHANNEL_MEMORY * 6)).
```

When called, the initialization function, `adi_dma_Init()`, initializes the memory that was passed in. Like all functions within the DMA Manager, the initialization function returns a return code that indicates success or the specific error that occurred during the function call. All DMA API functions return the `ADI_DMA_RESULT_SUCCESS` value to indicate success. All error codes are of the form `ADI_DMA_RESULT_XXXX`.

In addition to the return code, the `adi_dma_Init()` function returns a count of the number of channels it can manage simultaneously, and a handle into the DMA Manager. The channel count can be tested to ensure the DMA Manager can control the requested number of channels. The DMA Manager handle value that is returned is later passed into the `adi_dma_Open` and `adi_dma_MemoryOpen` functions. These functions use the manager handle to identify the DMA Manager that is to control the channel. Passing in this handle allows these functions to quickly identify the memory that will be used to manage the channel(s) being opened. After the DMA Manager has been initialized, DMA channels and memory streams can be opened for use.

While it is possible to create multiple DMA Managers in a unicore Blackfin system, there is no practical advantage in doing so.

Termination

When the DMA Manager is no longer needed, the client can terminate the DMA Manager with the `adi_dma_Terminate` function. This function is passed the DMA Manager handle given to the client in the `adi_dma_Init` function. The DMA Manager closes any open channels and streams, and then returns to the caller. After the return from the `adi_dma_Terminate()` function, the memory that was supplied to the DMA Manager via the `adi_dma_Init()` function can be reused by the client.

Note that in many embedded systems, the DMA Manager is never terminated.

Memory DMA and Peripheral DMA

As described in the *Blackfin Processor Hardware Reference*, the Blackfin DMA controller supports both peripheral DMA and memory DMA. Regardless of if peripheral DMA or memory DMA is being used, the client schedules DMA Manager activity on a block by block basis rather than on a sample by sample basis. While a block of data can be defined to be a single sample of data, this is seldom the case. Most often, data is blocked in quantities relevant to the processing that is to be performed. The term "buffer" is used throughout this document to represent the block of data.

Peripheral DMA is used to move blocks of data between on-chip peripherals and one of the memory spaces of the Blackfin processor, most commonly within the context of a device driver. For example, an on-chip peripheral such as a PPI uses DMA to move blocks of data into or out from the PPI device. As such, the device driver for the PPI typically uses the DMA Manager to control dataflow through the PPI.

Memory DMA describes the movement of data between any of the various Blackfin memory spaces. For example, due to the huge amounts of data used for video processing, video frames may be stored in external

SDRAM, and then DMA-ed piecemeal into internal L1 memory for processing.

The DMA Manager fully supports both peripheral DMA and memory DMA. When using peripheral DMA, clients leverage the capabilities of the DMA Manager on a channel by channel basis. When using memory DMA, clients can choose to control memory streams as individual source and destination channels using the same techniques and functions provided for peripheral DMA, or alternatively can control memory DMA as a single memory stream using the higher level `adi_dma_MemoryXXXX()` functions.

Controlling Memory Streams

When memory DMA is needed, controlling and scheduling memory DMA is most easily accomplished using higher level memory streams. The `adi_dma_MemoryXXXX()` functions provide a simple, efficient method to cause the Blackfin DMA controller to transfer data between the various memory spaces.

The overall sequence for using memory streams is to first open the memory stream, schedule transfers as needed, then close the memory stream when it is no longer needed. In many embedded systems, the memory stream is never closed, but remains open at all times.

Opening Memory Streams

To open the memory stream, the client calls the `adi_dma_MemoryOpen` function. The client passes the following parameters into the function:

- A handle to the DMA Manager that is to control the stream.
- The stream ID (of type `ADI_DMA_STREAM_ID`) that identifies which memory DMA stream to use.

- A client handle that is passed back to the client's callback function. This is a client supplied value, supposedly of some meaning to the client, which is passed back to the client's callback function so that the client can associate this value with the stream that is causing the callback.
- A pointer to a location into which the DMA Manager stores the stream handle. The stream handle is a DMA Manager defined value that uniquely identifies the stream to the DMA Manager.
- A handle to a deferred callback service (typically from the *Deferred Callback Service*) or a NULL value. If a NULL value is supplied, the DMA Manager will make "live" callbacks to the application. "Live" callbacks are made during hardware interrupt time. If a deferred callback service handle is provided, all callbacks for the stream use the deferred callback service to defer callback processing until after hardware interrupt time.

Memory Transfers

Once a memory stream has been opened, the client can submit jobs to the stream using the `adi_dma_MemoryCopy` and/or `adi_dma_MemoryCopy2D` functions. Linear, one-dimensional, memory transfers use the former function, two-dimensional transfers use the latter function. The same stream can be used for both one-dimensional and two-dimensional transfers, so a client can schedule a one-dimensional transfer on a given stream, then schedule a two-dimensional transfer on that same stream.

Note that a memory stream can support only one transfer at a time. If a transfer is in progress and another transfer is requested, these functions return an error code indicating the stream is in use. If queuing of memory transfers is required, this can be accomplished by using the channel-based method of controlling DMA.

One-Dimensional Transfers (Linear Transfers)

One-dimensional linear transfers are handled by calling the `adi_dma_MemoryCopy()` function. When calling the `adi_dma_MemoryCopy()` function, the client provides the following parameters:

- The stream handle. This is the value that was provided to the client during the `adi_dma_MemoryOpen()` function.
- The destination starting address into which the data is copied.
- The source starting address from which data is copied.
- The width of each element, in bytes, that is to be copied. The DMA Manager uses this value to schedule 8-, 16- or 32-bit transfers.
- A count of the number of elements to be copied.
- The address of the callback function that is called when the transfer is complete. When the callback function is actually invoked depends on the callback service handle value that was supplied to the stream when it was opened, either deferred or "live". If the `adi_dma_MemoryCopy()` function is passed a NULL value for the callback function address, the transfer occurs synchronously and the `adi_dma_MemoryCopy()` function does not return to the client until the transfer is complete. No callbacks are made in this case.

Two-Dimensional Transfers

Two-dimensional transfers are handled by calling the `adi_dma_MemoryCopy2D()` function. When calling the `adi_dma_MemoryCopy2D()` function, the client provides the following parameters:

- The stream handle. This is the value that was provided to the client during the `adi_dma_MemoryOpen()` function.
- A pointer to a data structure (of type `ADI_DMA_2D_TRANSFER`) that defines how data will be stored into the destination memory.
- A pointer to a data structure (of type `ADI_DMA_2D_TRANSFER`) that defines how data will be read from the source memory.
- The width of each element, in bytes, that is to be copied. The DMA Manager uses this value to schedule 8-, 16- or 32-bit transfers.
- The address of the callback function that is called when the transfer is complete. When the callback function is actually invoked depends on the callback service handle value that was supplied to the stream when it was opened (either deferred or “live”. If the `adi_dma_MemoryCopy()` function is passed a NULL value for the callback function address, the transfer will occur synchronously and the `adi_dma_MemoryCopy()` function does not return to the client until the transfer is complete. No callbacks are made in this case.

The `ADI_DMA_2D_TRANSFER` data type is a structure that defines the necessary values to describe a two-dimensional transfer. This data type contains the starting address in memory, an `XCount` value defining the number of columns, a `YCount` value defining the number of rows, and `XModify` and `YModify` values describing the stride for each.

Closing Memory Streams

When a memory stream is no longer needed, the `adi_dma_MemoryClose` function is called to close the stream. Once closed, a stream must be reopened before it can perform additional transfers. The client passes the following parameters into the function:

- The stream handle. This is the value that was provided to the client during the `adi_dma_MemoryOpen` function.
- A flag indicating whether or not the DMA Manager should wait for any ongoing transfers on the stream to complete before closing the channel.

Controlling DMA Channels

Controlling DMA on a channel by channel basis allows for the tightest control of DMA scheduling. Before a channel can be used, it must first be opened.

Opening DMA Channels

To open a DMA channel, the client calls the `adi_dma_Open()` function. The client passes into the function the following parameters:

- A handle to the DMA Manager that is to control the channel.
- The channel ID (of type `ADI_DMA_CHANNEL_ID`) that identifies the DMA channel to be opened.
- A client handle that is passed back to the client's callback function. This is a client supplied value, supposedly of some meaning to the client, which is passed back to the client's callback function so that the client can associate this value with the stream that is causing the callback.

- A pointer to a location into which the DMA Manager stores the channel handle. The channel handle is a DMA Manager defined value that uniquely identifies the channel to the DMA Manager.
- The operating mode that defines how the channel will move data. See the section "Operating Modes" on page 6-11.
- A handle to a deferred callback service (typically from the Deferred Callback Service) or a NULL value. If a NULL value is supplied, the DMA Manager will make "live" callbacks to the application. "Live" callbacks are made during hardware interrupt time. If a deferred callback service handle is provided, all callbacks for the stream will use the deferred callback service to make callbacks occur at non-hardware interrupt time.
- The address of the callback function that is called to notify the client of events. Events may be expected events, such as requests for notification when a transfer is complete, to unexpected events such as a DMA error. When the callback function is actually invoked, deferred or "live", depends on the callback service handle value that is supplied.

After the channel has been successfully opened, the channel can be additionally configured, buffers supplied to the channel, and so on. Note that the actual transferring of data does not begin with the `adi_dma_MemoryOpen` function. Dataflow must be specifically enabled via the `adi_dma_Control` function.

Operating Modes

The DMA Manager supports the following operational modes of the Blackfin DMA controller.

Single Transfers

The single transfer operating mode (`ADI_DMA_MODE_SINGLE`) is used to transfer individual, single buffers of data. When using the single transfer mode, the client calls the `adi_dma_Buffer()` function to schedule a transfer. The client passes to the function the following parameters:

- The channel handle. This is the value that was provided to the client during the `adi_dma_Open()` function.
- The starting address of the buffer. This value is the address in memory where data will initially be read from, if the transfer is for outbound data, or the address in memory where data will initially be stored, if the transfer is for inbound data.
- The configuration word for the transfer. This is a 16-bit value that represents the DMA Configuration register for the channel. The DMA Manager include file provides macros that allow the client to quickly and easily create a configuration word. The following fields within the configuration word are the only ones for which values must be provided:

`WNR` (Transfer Direction)	`ADI_DMA_WNR_READ`	Transfer is for outbound data
	`ADI_DMA_WNR_WRITE`	Transfer is for inbound data
`WDSIZE` (Transfer Element Size)	`ADI_DMA_WD_SIZE_8BIT`	Elements are 1 byte wide (8 bits)
	`ADI_DMA_WD_SIZE_16BIT`	Elements are 2 bytes wide (16 bits)
	`ADI_DMA_WD_SIZE_32BIT`	Elements are 4 bytes wide (32 bits)
`DMA2D` (Dimension Select)	`ADI_DMA_DMA2D_LINEAR`	One-dimensional (linear) transfer
	`ADI_DMA_DMA2D_2D`	Two-dimensional transfer
`DI_SEL` (Data Interrupt Timing Select) applies only when DMA2D = 1	`ADI_DMA_DI_SEL_OUTER_LOOP`	A callback is generated when the entire transfer has completed (outer loop).
	`ADI_DMA_DI_SEL_INNER_LOOP`	A callback is generated on each inner loop completion.

`DI_EN` (Data Interrupt Enable)	`ADI_DMA_DI_EN_DISABLE`	No callback is generated.
	`ADI_DMA_DI_EN_ENABLE`	The DMA Manager generates a callback to the client when the transfer completes.

- The `XCount` value. For one-dimensional transfers, this value defines the number of elements to be transferred. For two-dimensional transfers, this value defines the inner loop count (number of columns).
- The `XModify` value. For one-dimensional transfers, this value defines the address increment/decrement (stride) for each successive element. For two-dimensional transfers, this value defines the inner loop address increment/decrement (stride) for each successive element up to but not including the last element in each inner loop. After the last element in each inner loop, the `YModify` value is applied instead, except on the very last element of the transfer.
- The `YCount` value. This parameter is ignored for one-dimensional transfers. For two-dimensional transfers, the value represents the outer loop count (number of rows).
- The `YModify` value. This parameter is ignored for one-dimensional transfers. For two-dimensional transfers, this value defines the outer loop address increment/decrement (stride) that is applied after each inner loop completion. This value is the offset between the last element of one row and the first element of the next row.

Regardless of whether or not dataflow on the channel is enabled, the `adi_dma_Buffer()` function returns immediately to the caller. If dataflow is already enabled on the channel, the DMA Manager begins executing the transfer, otherwise the transfer does not begin until the dataflow is enabled via the `adi_dma_Control()` function. When using the single transfer mode, the `adi_dma_Buffer()` function can be called at any time, as long as a transfer on the channel is not already in progress.

Circular Transfers

The circular transfer mode (`ADI_DMA_MODE_CIRCULAR`) leverages the autobuffer capability of the DMA controller. Using the circular transfer mode, the client provides the DMA Manager with a single contiguous buffer comprising *n* sub buffers, as shown in Figure 6-1 on page 6-16. When dataflow is enabled, the DMA Manager begins transferring data at the start of the buffer, continuing on throughout the entire buffer, and then automatically looping back to the top of the buffer again, and repeating indefinitely. The client can optionally direct the DMA Manager to generate callbacks at the completion of each sub buffer, to generate callbacks at the completion of the entire buffer, or not to generate callbacks.

When using the circular transfer mode, the client calls the `adi_dma_Buffer()` function with the following parameters:

- The channel handle. This is the value that was provided to the client during the `adi_dma_Open()` function.
- The starting address of the buffer. This value is the address in memory where data will initially be read from, if the transfer is for outbound data, or the address in memory where data will initially be stored, if the transfer is for inbound data.
- The configuration word for the transfer. This is a 16-bit value that represents the DMA Configuration register for the channel. The DMA Manager include file provides macros that allow the client to

quickly and easily create a configuration word. The only fields within the configuration word the client need provide values for are:

WNR (Transfer Direction)	ADI_DMA_WNR_READ	A transfer is for outbound data.
	ADI_DMA_WNR_WRITE	A transfer is for inbound data.
DI_SEL (Data Interrupt Timing Select)	ADI_DMA_DI_SEL_OUTER_LOOP	A callback is generated on completion of whole buffer only.
	ADI_DMA_DI_SEL_INNER_LOOP	A callback is generated on each inner loop completion.
DI_EN (Data Interrupt Enable)	ADI_DMA_DI_EN_DISABLE	No callback will be generated.
	ADI_DMA_DI_EN_ENABLE	Callbacks are generated according the setting of DI_SEL.

- The XCount value. This parameter should be set to the number of elements in a single sub buffer.
- The XModify value. The width, in bytes, of an element. Allowed values are 1, 2 and 4 only.
- The YCount value. This parameter should be set to the number of sub buffers contained within the whole buffer.
- The YModify value. This parameter is ignored.

When using the circular mode, the adi_dma_Buffer() function must be called prior to enabling dataflow on the channel. If after enabling dataflow, the client wants to change to a different circular buffer, the client must first disable dataflow on the channel, then call the adi_dma_Buffer() function with the new buffer data, then re-enable dataflow on the appropriate channel.

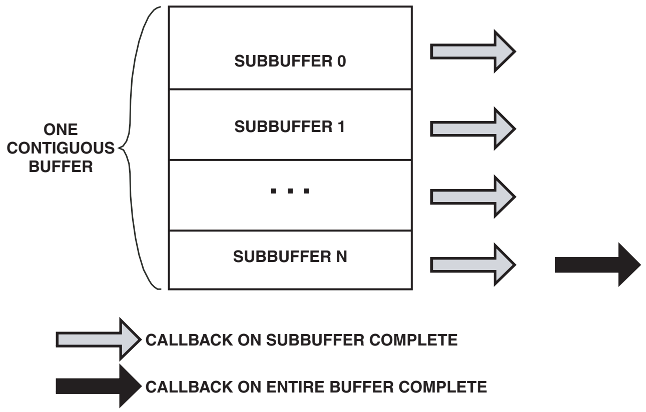
ONE CONTIGUOUS BUFFER
SUBBUFFER 0
SUBBUFFER 1
. . .
SUBBUFFER N
CALLBACK ON SUBBUFFER COMPLETE
CALLBACK ON ENTIRE BUFFER COMPLETE

Figure 6-1. Circular Buffer

Large Descriptor Chaining Model

The large descriptor chaining model (`ADI_DMA_MODE_DESCRIPTOR_LARGE`) allows the client to create chains of descriptors, residing anywhere in memory, where each descriptor describes a specific work unit.

Using the large descriptor chaining mode, the client provides the DMA Manager with one or more descriptor chains, as shown in Figure 6-2. Descriptors can be submitted at any time, regardless of the dataflow state. The DMA Manager maintains independent queues of descriptors for each channel, keeping the DMA controller busy with transfers until all queued descriptors have been processed. Both one-dimensional transfers and two-dimensional transfers can be intermixed on the same channel. Each transfer can define a different transfer type, length, and so on. Additionally, callbacks to the client's callback function, can be made upon completion of every descriptor, any individual descriptor or configured to never callback.

When the large descriptor chaining mode is used, descriptor chains are submitted to the channel using the `adi_dma_Queue()` function with the following parameters:

- The channel handle. This is the value that was provided to the client during the `adi_dma_Open()` function.
- A handle, of the type `ADI_DMA_DESCRIPTOR_HANDLE`, to a descriptor. As the same `adi_dma_Queue()` function is used for all descriptor based operating modes, including large descriptors, small descriptors and arrays of descriptors, the `ADI_DMA_DESCRIPTOR_HANDLE` data type acts as a container that conveniently represents each of the descriptor types.

For the large descriptor chaining mode, descriptors are of the type `ADI_DMA_DESCRIPTOR_LARGE`, a data type that defines a large model descriptor. When calling the `adi_dma_Queue()` function, the client can either pass in the address of the descriptor union (`ADI_DMA_DESCRIPTOR_UNION`) or alternatively, the address of the descriptor itself (`ADI_DMA_DESCRIPTOR_LARGE`) to the `ADI_DMA_DESCRIPTOR_HANDLE` data type. This descriptor can be either a single descriptor or the first descriptor in a chain of descriptors.

Large model descriptors contain all the information necessary for the DMA Manager to control the operation of the DMA controller. This information includes:

- A pointer to the next large descriptor in the chain. If this field is NULL, then the given descriptor is the only descriptor the client is submitting to the channel.
- The starting address of the buffer. This value is the address in memory where data will initially be read from, if the transfer is for outbound data, or the address in memory where data will initially be stored, if the transfer is for inbound data.

- The configuration word for the transfer. This is a 16-bit value that represents the DMA Configuration register for the channel. The DMA Manager include file provides macros that allow the client to quickly and easily create a configuration word. The only fields within the configuration word the client need provide values for are:

`WNR` (Transfer Direction)	`ADI_DMA_WNR_READ`	Transfer is for outbound data.
	`ADI_DMA_WNR_WRITE`	Transfer is for inbound data.
`WDSIZE` (Transfer Element Size)	`ADI_DMA_WD_SIZE_8BIT`	Elements are 1 byte wide (8 bits).
	`ADI_DMA_WD_SIZE_16BIT`	Elements are 2 bytes wide (16 bits).
	`ADI_DMA_WD_SIZE_32BIT`	Elements are 4 bytes wide (32 bits).
`DMA2D` (Dimension Select)	`ADI_DMA_DMA2D_LINEAR`	One-dimensional (linear) transfer
	`ADI_DMA_DMA2D_2D`	Two-dimensional transfer
`DI_EN` (Data Interrupt Enable)	`ADI_DMA_DI_EN_DISABLE`	No callback will be generated.
	`ADI_DMA_DI_EN_ENABLE`	The DMA Manager generates a callback to the client when the transfer completes.

- The `XCount` value. For one-dimensional transfers, this value defines the number of elements to be transferred. For two-dimensional transfers, this value defines the inner loop count (number of columns).

- The `XModify` value. For one-dimensional transfers, this value defines the address increment/decrement (stride) for each successive element. For two-dimensional transfers, this value defines the inner loop address increment/decrement (stride) for each successive element up to but not including the last element in each inner loop. After the last element in each inner loop, the `YModify` value is applied instead, except on the very last element of the transfer.

- The `YCount` value. This parameter is ignored for one-dimensional transfers. For two-dimensional transfers, the value represents the outer loop count (number of rows).
- The `YModify` value. This parameter is ignored for one-dimensional transfers. For two-dimensional transfers, this value defines the outer loop address increment/decrement (stride) that is applied after each inner loop completion. This value is the offset between the last element of one row and the first element of the next row.

Although the DMA Manager does not constrain when descriptors can be provided to a channel, for DMA channels that will be processing inbound data it is best practice to provide descriptors to the channel via the `adi_dma_Queue()` function before enabling dataflow. By doing this, the DMA controller uses a space where data can be stored. If dataflow is enabled on an inbound channel prior to providing descriptors, it is possible for data to be received by the DMA channel but not have anywhere to store it.

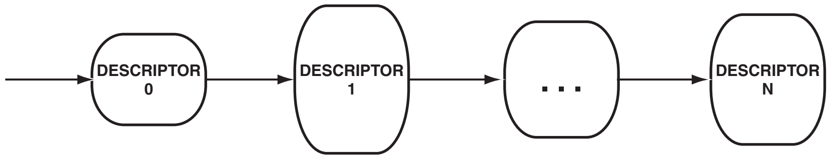

Figure 6-2. Descriptor Chain

Small Descriptor Chaining Model

The small descriptor chaining model (`ADI_DMA_MODE_DESCRIPTOR_SMALL`) is similar to the large descriptor chaining model. The only material difference between the two models is that in the small descriptor model, the pointer to the next descriptor in a chain of descriptors consists of only the lower 16 bits of address, rather than a full 32-bit address. This means that all descriptors on a channel that is using the small descriptor model must have the same upper 16 bits of address. In other words, all small model descriptors for a channel must be located within the same 64KB segment.

This difference is encapsulated in the `ADI_DMA_DESCRIPTOR_SMALL` data type. In order to avoid data alignment issues, a consequence of having the next descriptor pointer exist as a 16-bit entry rather than a 32-bit entry, the starting address of the data within the descriptor is declared as two 16-bit entries, rather than a single 32-bit entry. Performing two 16-bit accesses, rather than a single 32-bit access avoids any alignment exceptions.

Other than these differences, the small descriptor chaining model is functionally identical to the large descriptor chaining model.

Arrays of Descriptors

The descriptor array mode (`ADI_DMA_MODE_DESCRIPTOR_ARRAY`) is not yet supported in the Device Manager.

Configuring a DMA Channel

Once a DMA channel has been opened, the client can detect and modify the configuration of the channel via the `adi_dma_Control` function. The complete list of configuration control commands are provided in Table 6-2 on page 6-62. In most cases, the client passes the following parameters to the `adi_dma_Control()` function:

- The channel handle. This is the value that was provided to the client during the `adi_dma_Open` function.
- The command ID. This `ADI_DMA_CMD` data type identifies the controllable item that is being configured.
- A command specific value. This semantics of this parameter are defined by the command ID. For example, given a command ID of `ADI_DMA_CMD_SET_DATAFLOW`, the command specific value is either TRUE or FALSE, to enable or disable dataflow on the channel. The command specific value is always cast to (`void*`).

Closing a DMA Channel

To close a DMA channel, the client calls the `adi_dma_Close()` function. The client passes the following parameters into the function:

- The channel handle. This is the value that was provided to the client during the `adi_dma_Open()` function.
- A flag indicating whether or not the DMA Manager should wait for any DMA activity on the channel to complete before closing the channel.

Once a channel has been closed, the channel must be reopened with the `adi_dma_Open()` function before it can be used again.

Transfer Completions

Client applications can use two different mechanisms to determine when transfers complete. One method is by polling the channel, the other method is through callbacks.

In addition to polling and callbacks, the memory stream functions offer a synchronous capability. When used synchronously, the `adi_dma_MemoryCopy()` and `adi_dma_MemoryCopy2D()` functions return to the client only when the transfer is complete.

Polling

Clients can use the `adi_dma_Control()` function to interrogate a specific channel to determine if a transfer is in progress by using the `ADI_DMA_CMD_GET_TRANSFER_STATUS` command ID. When given this command, the DMA Manager examines the status of the individual DMA channel. The function provides a response of TRUE, if a transfer is in progress, and a response of FALSE, if the no transfer is currently in progress.

Note that memory streams can also be interrogated for transfer status. Instead of passing the channel handle (`ADI_DMA_CHANNEL_HANDLE`) parameter to the `adi_dma_Control()` function, the client should pass the stream handle (`ADI_DMA_STREAM_HANDLE`) parameter (casted to the `ADI_DMA_CHANNEL_HANDLE` data type) to the `adi_dma_Control()` function.

Callbacks

Callbacks are the more commonly used mechanism for clients to determine when transfers have completed. Callbacks are either “live”, meaning they are made at interrupt time, or deferred, meaning they are made after the hardware interrupt has completed processing using a callback service.

Memory Stream Callbacks

When using memory streams, if the client provided a callback function as a parameter to the `adi_dma_MemoryCopy()` or `adi_dma_MemoryCopy2D()` functions, the callback function is invoked by the DMA Manager when the transfer is complete.

When using memory streams, the following arguments are passed to client callback functions:

- The client handle. This is the client supplied value that was provided in the `adi_dma_MemoryOpen()` function.
- Event ID. This value is `ADI_DMA_EVENT_DESCRIPTOR_PROCESSED`.
- Starting destination address of the transfer.

Circular Transfer Callbacks

When using the circular transfer method (`ADI_DMA_MODE_CIRCULAR`), the client uses the configuration word to specify the frequency of callbacks. When directed to callback the client on each sub buffer completion, the DMA Manager invokes the client's callback function after each sub buffer completes. For example, this is useful in double-buffering schemes, where two sub buffers (ping/pong) are used.

When using circular transfers, the following arguments are passed to client callback functions:

- The client handle. This is the client supplied value that was provided in the `adi_dma_Open()` function.
- Event ID. This value is either the `ADI_DMA_EVENT_INNER_LOOP_PROCESSED` when a sub buffer has completed processing or the `ADI_DMA_EVENT_OUTER_LOOP_PROCESSED` when the entire buffer has completed processing.
- Starting address of the data buffer.

Descriptor Callbacks

When using any of the descriptor based transfer methods (ADI_DMA_MODE_DESCRIPTOR_LARGE, ADI_DMA_MODE_DESCRIPTOR_SMALL or ADI_DMA_DESCRIPTOR_ARRAY), the client uses the configuration word of the descriptor to define whether or not a callback is to be generated following processing of a descriptor. When directed to callback the client upon completion of the descriptor, the client callback function is passed the following arguments:

- The client handle. This is the client supplied value that was provided in the adi_dma_Open() function.
- Event ID. This value is ADI_DMA_EVENT_DESCRIPTOR_PROCESSED.
- Starting address of the data.

Descriptor Based Submodes

When using the small or large model descriptor based transfers, two submodes allow the client application greater flexibility in processing descriptors. Each of these submodes can be used independently or in combination. Each submode is enabled or disabled via the adi_dma_Control() function. Clients who want to use these submodes must enable them prior to enabling dataflow on the channel. Both submodes are disabled by default.

Loopback Submode

The loopback submode is controlled by the ADI_DMA_CMD_SET_LOOPBACK command.

When the loopback submode is enabled (after the DMA Manager has processed the last descriptor in the chain of descriptors provided to a channel), it automatically loops back to the first descriptor that was provided to the channel. This effectively creates an infinite loop of descriptors, as illustrated in Figure 6-3. For example, with the loopback submode, the client can provide the descriptors at initialization time, let the DMA Manager process the descriptors, and never have to resupply the DMA Manager with additional descriptors.

As in the non-loopback case, each descriptor, any one, none or all descriptors can be tagged to generate a callback to the client after processing.

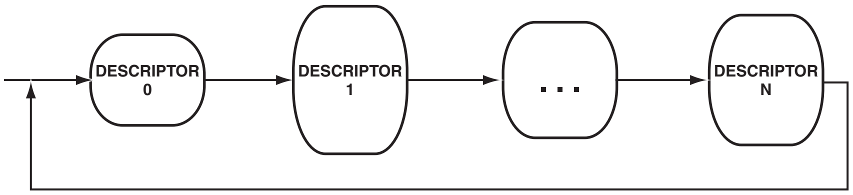

Figure 6-3. Descriptor Chain with Loopback

Streaming Submode

The streaming submode is controlled by the `ADI_DMA_CMD_SET_STREAMING` command.

When not using the streaming submode, the DMA Manager pauses the DMA controller after a descriptor that has been tagged to generate a callback has been processed. The DMA Manager does this because the Blackfin DMA controller does not provide any status information indicating that a specific descriptor has been processed. If the DMA Manager did not pause the controller, it is possible that before the DMA Manager can recognize and process the callback interrupt for a given descriptor, the DMA controller may have completed processing of yet another descriptor. Unless the DMA controller pauses until the DMA Manager processes the interrupt, the DMA Manager cannot definitively determine which callback interrupt is associated with which descriptor.

When not streaming, the DMA Manager also pauses the DMA controller when a channel has exhausted its supply of descriptors.

The streaming submode allows the client to alter this behavior. When the streaming submode is enabled, the DMA Manager never pauses the DMA controller, allowing the DMA transfers to occur at the maximum throughput rate. When streaming, the client is required to ensure the following conditions:

- The channel always has descriptors to process and never runs out of descriptors.
- The system timing is such that the DMA Manager can service the callback interrupt for any descriptor tagged for a callback, before another descriptor on the same channel that is tagged for callback is processed.

These conditions can be fairly easily met in most systems.

DMA Channel to Peripheral Mapping

The Blackfin processor allows the user to change the default mapping of the various DMA supported peripherals to the various DMA channels. Note however, that the mappings for the Memory DMA channels are typically fixed and cannot be changed.

The DMA Manager provides two functions that allow the client to easily detect and change the mapping of DMA channels to peripherals. These functions can be called at any time after the DMA Manager has been initialized, but they must be processed before the channel is opened.

Sensing a Mapping

The client calls the `adi_dma_GetMapping()` function to detect the DMA channel ID to which a peripheral is mapped. The `adi_dma_GetMapping()` function takes the following parameters:

- The peripheral ID. This value, an `ADI_DMA_PMAP` type, enumerates the peripheral whose mapping is being detected.
- Pointer to an `ADI_DMA_CHANNEL_ID` value. This value is the address of a location in memory into which the function will store the channel ID to which the given peripheral is mapped.

Setting a Mapping

The client calls the `adi_dma_SetMapping()` function to set the mapping of a given channel ID to a given peripheral. The client should take care to ensure that a one to one mapping exists between peripherals and channel IDs. The `adi_dma_SetMapping()` function takes the following parameters:

- The peripheral ID. This value, an `ADI_DMA_PMAP` type, enumerates the peripheral whose mapping is being set.
- The channel ID. This value, an `ADI_DMA_CHANNEL_ID` value, enumerates the DMA channel to which the given peripheral is to be mapped.

Interrupts

The DMA Manager uses the services of the Interrupt Manager to configure all DMA related interrupts. All hooking of interrupts is isolated into the `adi_dma_Open()` and `adi_dma_MemoryOpen()` functions while all unhooking of interrupts occurs in the `adi_dma_Close()` and `adi_dma_MemoryClose()` functions.

By default, the DMA Manager uses the *Interrupt Vector Group* (IVG) settings as set up by the Interrupt Manager. The mapping of DMA channels to IVG levels can be altered by the client via calls into the Interrupt Manager. See "Interrupt Manager" on page 2-1 for more information on altering mapping of DMA channels to IVGs.

Hooking Interrupts

When the client opens the first DMA channel is opened, the `adi_dma_Open()` function hooks into the appropriate IVG chain for the DMA error interrupt. The handler for DMA errors does nothing other than clear the appropriate DMA error and notify the client's callback function that a DMA error occurred.

In addition to the DMA error interrupt, the `adi_dma_Open()` function hooks the DMA data interrupt handler into the appropriate IVG level for the given channel. The data interrupt handler is used to post callbacks resulting from the completion of DMA transfers. In addition to posting the notification callbacks, the data handler ensures the channel is refreshed and restarted (if necessary) with any new pending transfers.

Unhooking Interrupts

When the last remaining open DMA channel is closed, the `adi_dma_Close()` function unhooks the DMA error handler from the appropriate IVG handler chain. In addition, if there are no other open channels that are mapped to the same IVG as the channel being closed, the `adi_dma_Close()` function unhooks the DMA data handler from the chain of handlers for that IVG.

Two-Dimensional DMA

When using linear DMA, data is moved in a one-dimensional, linear fashion. This is the most common type of transfer, where *n* elements of 'w' width are moved from one location, or taken in through a device, to another memory location, or out through a device.

Two-dimensional DMA is a convenient feature that allows data to be transferred in a non-linear fashion, which is especially useful in video type applications. Two-dimensional DMA supports arbitrary row (YCount) and column (XCount) sizes up to 64K x 64K elements, as well as row modify values (YModify) and column modify values up to +/- 32K bytes.

When using channel DMA, descriptors are used to define the parameters for the transfer. When using memory streams, the `ADI_DMA_2D_TRANSFER` data type is used to define the parameters for the transfer.

For example, suppose we want to retrieve a 16 x 8 block of bytes (data) from a video frame buffer (frame) of size N x M pixels at location frame[6][6] and store it in a separate memory area (data) to process. After the data has been processed the values are then copied back to the original location. Figure 6-4 illustrates the area of the frame to be processed.

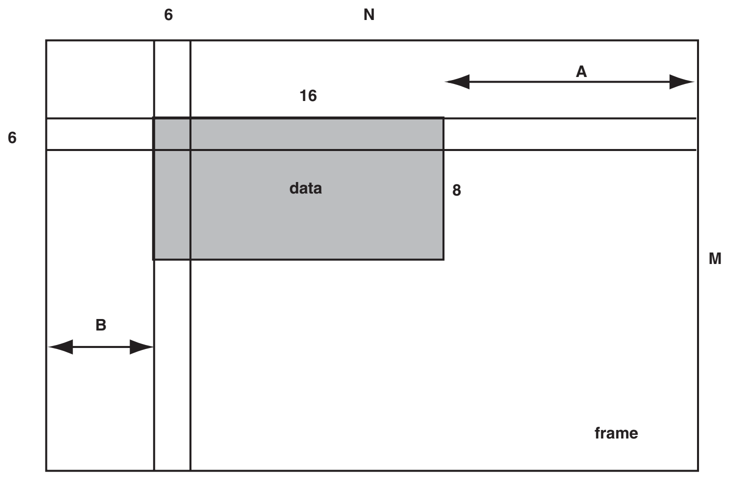

Figure 6-4. Selecting a 16 x 8 Block of Data from a Video Frame of Size N x M

To select each row of the 16 x 8 block, the inner loop of the required 2D DMA configuration has 16 values (`XCOUNT=16`) and a stride (`XMODIFY`) of 1. The outer loop comprises 8 values (`YCOUNT=8`) and a stride (`YMODIFY`) of N-15 (A + B in Figure 6-4) chosen to instruct the DMA controller to jump from the end of one row to the start of the next.

It would also be possible to extract interleaved data (for example, RGB values for a video frame) by modifying both the x and y `modify` values. For example to receive a stream of R,G,B,R,G,B,... values from an N x M frame, consider Figure 6-5:

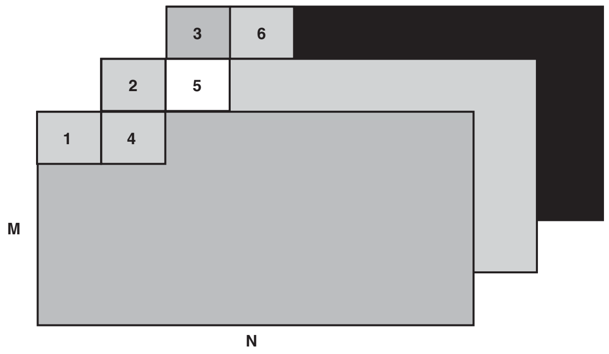

Figure 6-5. Capturing a Video Data Stream of (R,G,B Pixels) x (N x M image Size)

In this case the inner loop of the required 2D DMA configuration has 3 values (`XCOUNT=3`) and a stride (`XMODIFY`) of N*M chosen so that successive elements in each row (or RGB tuple) are 1 - 2 - 3, 4 - 5 - 6, and so on. in Figure 6-5. The outer loop of the 2D DMA configuration has N*M values (`YCOUNT=N*M`) and a negative stride (`YMODIFY`) of 1-2*N*M chosen to instruct the DMA controller to jump from element 3 to 4, 6 to 7, and so on at the end of each inner loop.

DMA Manager API Reference

This section provides descriptions of the DMA Manager API functions.

Notation Conventions

The reference pages for the API functions use the following format:

Name and purpose of the function

Description – Function specification

Prototype – Required header file and functional prototype

Arguments – Description of function arguments

Return Value – Description of function return values

The DMA Manager API supports the following functions.

Table 6-1. DMA Manager API Functions

Function	Description
Primary Functions	
`adi_dma_Init`	Initializes a DMA Manager. See "adi_dma_Init" on page 6-41.
`adi_dma_Terminate`	Shuts down and terminates a DMA Manager. See "adi_dma_Terminate" on page 6-53.
`adi_dma_Open`	Opens a DMA channel for use. See "adi_dma_Open" on page 6-49.
`adi_dma_Close`	Closes a DMA channel. See "adi_dma_Close" on page 6-39.
`adi_dma_Control`	Controls/queries the operation of a DMA Channel. See "adi_dma_Control" on page 6-36.
`adi_dma_Queue`	Queues a descriptor chain. See "adi_dma_Queue" on page 6-51.

Table 6-1. DMA Manager API Functions (Cont'd)

Function	Description
`adi_dma_Buffer`	Provides a single or circular buffer. See "adi_dma_Buffer" on page 6-34.
	Helper Functions
`adi_dma_GetMapping`	Gets the DMA Channel ID to which a peripheral is mapped. See "adi_dma_GetMapping" on page 6-40.
`adi_dma_SetMapping`	Sets the DMA Channel ID to which a peripheral is mapped. See "adi_dma_SetMapping" on page 6-52.
	Memory DMA Functions
`adi_dma_MemoryOpen`	Opens a memory DMA stream for use. See "adi_dma_MemoryOpen" on page 6-47.
`adi_dma_MemoryClose`	Closes a memory DMA stream. See "adi_dma_MemoryClose" on page 6-42.
`adi_dma_MemoryCopy`	Copies memory in a linear, one-dimensional fashion. See "adi_dma_MemoryCopy" on page 6-43.
`adi_dma_MemoryCopy2D`	Copies memory in a two-dimensional fashion. See "adi_dma_MemoryCopy2D" on page 6-45.

adi_dma_Buffer

Description

This function assigns a one-shot or a circular buffer to a DMA channel and configures the DMA Channel according to the parameters supplied.

Prototype

```
ADI_DMA_RESULT adi_dma_Buffer(
        ADI_DMA_CHANNEL_HANDLE    ChannelHandle,
        void                      *StartAddress,
        ADI_DMA_CONFIG_REG        Config,
        u16                       XCount,
        S16                       XModify,
        u16                       YCount,
        S16                       YModify
);
```

Arguments

`ChannelHandle`	This argument uniquely identifies the DMA channel that the buffer is to be assigned to and is the value returned when the DMA channel was opened.
`StartAddress`	This is the location of the start of the buffer to be either filled or transmitted.
`Config`	This is the DMA configuration control register for the transfer.
`XCount`	This is the total number of words to be transferred in a one-dimensional buffer or the number of data elements per row in a two-dimensional buffer.
`XModify`	The offset in bytes between each word to be transferred (1-D) or the offset in bytes between each row element (2-D).

`YCount`	This is the number of rows to be transferred.
`YModify`	The offset in bytes between the last data element of one row and the first element of the next.

Return Value

`ADI_DMA_RESULT_SUCCESS`	The buffer was assigned successfully.
`ADI_DMA_RESULT_BAD_HANDLE`	The `ChannelHandle` does not contain a valid Channel handle.
`ADI_DMA_RESULT_BAD_MODE`	The DMA channel has not been opened for either single or circular buffer operation.
`ADI_DMA_RESULT_ALREADY_RUNNING`	A DMA operation is in progress.

adi_dma_Control

Description

The `adi_dma_Control` function controls/queries the operation of the specified DMA Channel.

The function can be used in several ways:

- A single command is passed.

  ```
  adi_dma_Control(ChannelHandle, ADI_DMA_CMD_SET_LOOPBACK,
  (void*)  TRUE);
  ```

- A single command, value pair is passed; for example,

  ```
  ADI_DMA_CMD_VALUE_PAIR cmd = {
           ADI_DMA_CMD_SET_WORD_SIZE, (void*) ADI_DMA_
  WDSIZE_32BIT};
  adi_dma_Control(ChannelHandle, cmd.CommandID ,cmd.Value);
  ```

- A single `ADI_DMA_CMD_VALUE_PAIR` structure is passed (by reference):

  ```
  adi_dma_Control(ChannelHandle,ADI_DMA_CMD_VALUE_
  PAIR,&cmd);
  ```

- A table of `ADI_COMMAND_PAIR` structures is passed. The table must have following terminator entry to signify the end of the table of commands: `{ ADI_DMA_CMD_END, 0 }`. For example,

  ```
  ADI_DMA_CMD_VALUE_PAIR table = {
      {ADI_DMA_CMD_SET_LOOPBACK,  (void*)LoopbackFlag},
      {ADI_DMA_CMD_SET_DATAFLOW,  (void*)TRUE},
      { ADI_DMA_CMD_END, NULL };

  adi_dma_Control(ChannelHandle,ADI_DMA_CMD_TABLE,&table);
  ```

The set of commands that can be issued using the adi_dma_Control function is defined in "DMA Commands" on page 6-62.

Prototype

```
ADI_DMA_RESULT adi_dma_Control(
        ADI_DMA_CHANNEL_HANDLE     ChannelHandle,
        ADI_DMA_CM                 Command,
        void                       *Value
);
```

Arguments

ChannelHandle	This argument uniquely identifies the DMA channel that the buffer is to be assigned to and is the value returned when the DMA channel was opened.
Command	This is an ADI_DMA_CMD enumeration value (see "DMA Commands" on page 6-62 for a full list of commands).
Value	Depending on the value for Command, this parameter is one of the following: • If Command has the value ADI_DMA_CM_VALUE_PAIR, the system issues the address of a single ADI_DMA_CMD_VALUE_PAIR element specifying the command. • If Command has the value ADI_DMA_CMD_TABLE, the system issues the address of an array of ADI_DMA_CMD_VALUE_PAIR elements specifying one or more commands. The last entry in the table must be {ADI_DMA_CMD_END,NULL}. • For any other value, Command specifies the command to be processed and Value is the associated value for the command. In the case of a command that queries a value, the value of the setting is stored at the location pointed to by the pointer Value.

Return Value

ADI_DMA_RESULT_SUCCESS	This function completed successfully.
ADI_DMA_RESULT_BAD_COM-MAND	The command is invalid. Either a bad command or a specific command is not allowed in this context.
ADI_DMA_RESULT_ALREADY_RUNNING	The commands could not be performed as the channel is currently transferring data.

adi_dma_Close

Description

This function closes a channel and releases the configuration memory for further use. Depending on the value of the `WaitFlag` argument, either the channel is closed immediately or after ongoing transfers have completed.

Prototype

```
ADI_DMA_RESULT adi_dma_Close(
        ADI_DMA_CHANNEL_HANDLE       ChannelHandle,
        u32                          WaitFlag);
```

Arguments

`ChannelHandle`	This argument uniquely identifies the DMA channel to be closed and is the value returned when the DMA channel was opened.
`WaitFlag`	If set to `TRUE (1)`, the argument instructs the DMA Manager to wait for ongoing transfers to complete before closing the channel; otherwise, if set to `FALSE (0)`, the channel will be closed immediately terminating any ongoing transfers.

Return Value

`ADI_DMA_RESULT_SUCCESS`	DMA Channel successfully closed.
`ADI_DMA_RESULT_BAD_HANDLE`	`ChannelHandle` does not point to a valid channel.
`ADI_DMA_RESULT_CANT_UNHOOK_INTERRUPT`	The data handler and/or error handler cannot be unhooked.

adi_dma_GetMapping

Description

This function is used to identify the DMA Channel ID to which a DMA compatible peripheral is mapped.

Prototype

```
ADI_DMA_RESULT adi_dma_GetMapping(
        ADI_DMA_PMAP            Peripheral,
        ADI_DMA_CHANNEL_ID      *pChannelID
);
```

Arguments

`Peripheral`	The peripheral ID is being queried.
`*pChannelID`	This is the location where the DMA Manager stores the channel ID to which the peripheral is a mapped.

Return Value

`ADI_DMA_RESULT_SUCCESS`	The device is identified and DMA information is returned.
`ADI_DMA_RESULT_BAD_PERIPHERAL`	A bad peripheral value was encountered.
`ADI_DMA_RESULT_NOT_MAPPED`	No mapping was found for the device.

adi_dma_Init

Description

This function initializes a DMA Manager.

Prototype

```
ADI_DMA_RESULT adi_dma_Init(
        void                        *pMemory,
        sizet                       MemorySize,
        u32                         *pMaxChannels
        ADI_DMA_MANAGER_HANDLE      *pManagerHandle,
        void                        *pCriticalRegionArg
);
```

Arguments

*pMemory	This is the pointer to memory that the DMA can use.
MemorySize	This is the size, in bytes, of the memory provided.
*pMaxChannels	This is the location in memory where the DMA Manager stores the number of simultaneously open channels that can be supported given the memory provided.
*pManagerHandle	This is the location in memory where the DMA Manager stores the handle to the DMA Manager.
*pCriticalRegionArg	This is the parameter that the DMA Manager passes to the adi_int_EnterCriticalRegion() function.

Return Value

This function returns ADI_DMA_RESULT_SUCCESS if successful. Any other value indicates an error. Possible errors include:

ADI_DMA_RESULT_NOMEMORY	Insufficient memory is available to initialize the DMA Manager.

adi_dma_MemoryClose

Description

This function closes down a memory DMA stream, freeing up all resources used by the memory stream.

Prototype

```
ADI_DMA_RESULT adi_dma_MemoryOpen(
    ADI_DMA_STREAM_HANDLE        StreamHandle,
    u32                          WaitFlag
) ;
```

Arguments

`StreamHandle`	This is the handle to the DMA memory stream
`WaitFlag`	If set to TRUE (1), this argument instructs the DMA Manager to wait for ongoing transfers to complete before closing down the memory stream; otherwise, if set to FALSE (0), the channel is closed immediately, terminating any transfers in progress.

Return Value

The function returns `ADI_DMA_RESULT_SUCCESS` if successful. Any other value indicates an error. Possible errors include:

`ADI_DMA_RESULT_BAD_HANDLE`	The `StreamHandle` parameter does not point to a valid memory stream.

adi_dma_MemoryCopy

Description

This function performs a one-dimensional, linear memory copy.

Prototype

```
ADI_DMA_RESULT adi_dma_MemoryCopy(
    ADI_DMA_STREAM_HANDLE   StreamHandle,
    void                    *pDest,
    void                    *pSrc,
    u16                     ElementCount,
    u16                     ElementWidth,
    ADI_DCB_CALLBACK_FN     ClientCallback
) ;
```

Arguments

`StreamHandle`	This is the handle to the DMA memory stream.
`*pDest`	This is the starting address into which the memory will be copied.
`*pDest`	This is the starting address from which the memory will be copied.
`ElementCount`	This is the number of elements to transfer.
`ElementWidth`	This is the width, in bytes, of an element. Allowed values are 1, 2 and 4.
`ClientCallback`	Callback function that is called when the transfer completes. If NULL, the call to the `adi_dma_MemoryCopy()` function is considered synchronous and does not return to the client until the transfer has completed.

Return Value

The function returns `ADI_DMA_RESULT_SUCCESS` if successful. Any other value indicates an error. Possible errors include:

`ADI_DMA_RESULT_BAD_HANDLE`	The `StreamHandle` parameter does not point to a valid memory stream.
`ADI_DMA_RESULT_IN_USE`	The memory stream already has a transfer in progress.

adi_dma_MemoryCopy2D

Description

This function performs a two-dimensional memory copy.

Prototype

```
ADI_DMA_RESULT adi_dma_MemoryCopy2D(
    ADI_DMA_STREAM_HANDLE      StreamHandle,
    ADI_DMA_2D_TRANSFER        *pDest,
    ADI_DMA_2D_TRANSFER        *pSrc,
    u32                        ElementWidth,
    ADI_DCB_CALLBACK_FN        ClientCallback

);
```

Arguments

`StreamHandle`	This is the handle to the DMA memory stream.
`*pDest`	This is the pointer to the structure that describes how and where the data will be copied into memory.
`*pDest`	This is the pointer to the structure that describes how and where the data will be copied from memory.
`ElementWidth`	This is the width, in bytes, of an element. Allowed values are 1, 2 and 4.
`ClientCallback`	This is the callback function that is called when the transfer completes. If NULL, the call to the `adi_dma_MemoryCopy()` function is considered synchronous and does not return to the client until the transfer has completed.

Return Value

The function returns `ADI_DMA_RESULT_SUCCESS` if successful. Any other value indicates an error. Possible errors include:

`ADI_DMA_RESULT_BAD_HANDLE`	The `StreamHandle` parameter does not point to a valid memory stream.
`ADI_DMA_RESULT_IN_USE`	The memory stream already has a transfer in progress.

adi_dma_MemoryOpen

Description

This function opens a memory DMA stream for use. Once it is opened, memory DMA transfers can be scheduled on the stream.

Prototype

```
ADI_DMA_RESULT adi_dma_MemoryOpen(
    ADI_DMA_MANAGER_HANDLE   ManagerHandle,
    ADI_DMA_STREAM_ID        StreamID,
    void                     *ClientHandle,
    ADI_DMA_STREAM_HANDLE    *pStreamHandle,
    ADI_DCB_HANDLE           DCBServiceHandle
) ;
```

Arguments

`ManagerHandle`	This is the handle to the DMA Manager.
`StreamID`	This is the memory stream ID that is being opened.
`*ClientHandle`	This is an identifier defined by the client. The DMA Manager includes this identifier in all DMA Manager initiated communication with the client, specifically in calls to the callback function.
`*pStreamHandle`	This is the pointer to a client provided location wither the DMA Manager stores an identifier defined by the DMA Manager. All subsequent communication initiated by the client to the DMA Manager for this memory stream includes this handle.
`DCBServiceHandle`	This is the handle to the deferred callback service to be used for any memory stream events. A value of NULL means that deferred callbacks are not used and all callbacks occur at DMA interrupt time.

Return Value

The function returns `ADI_DMA_RESULT_SUCCESS` if successful. Any other value indicates an error. Possible errors include:

`ADI_DMA_RESULT_ALL_IN_USE`	All channel memory is in use.
`ADI_DMA_RESULT_CANT_HOOK_INTERRUPT`	The system cannot hook a DMA data or error interrupt.

adi_dma_Open

Description

The `adi_dma_Open` function opens a DMA channel for use. The DMA Manager ensures the channel is not already opened and then initializes any appropriate data structures.

Prototype

```
ADI_DMA_RESULT adi_dma_Open(
   ADI_DMA_CHANNEL_ID         ChannelID
   ADI_DMA_MANAGER_HANDLE     ManagerHandle
   void                       *ClientHandle,
   ADI_DMA_CHANNEL_HANDLE     *pChannelHandle,
   ADI_DMA_MODE               Mode,
   ADI_DCB_HANDLE             DCBServiceHandle,
   ADI_DCB_CALLBACK_FN        ClientCallback
);
```

Arguments

ManagerHandle	This is the handle to the DMA Manager.
ChannelID	This is the ADI_DMA_CHANNEL_ID enumeration value (see "ADI_DMA_CHANNEL_ID" on page 6-58).
*ClientHandle	This is an identifier defined by the client. The DMA Manager includes this identifier in all DMA Manager initiated communication with the client, specifically in calls to the callback function.
*pChannelHandle	This is the pointer to a client provided location where the DMA Manager stores an identifier defined by the DMA Manager. All subsequent communication initiated by the client to the DMA Manager for this channel includes the handle to specify the channel to which it is referring.
Mode	This is the an ADI_DMA_MODE enumeration value (on page 6-58) specifying the data transfer mode to be used by the opened DMA channel.
DCBServiceHandle	This is the handle to the deferred callback service to be used for the given channel. A value of NULL means that deferred callbacks are not used and all callbacks occur at DMA interrupt time.
ClientCallback	This is the address of a call-back function defined by the application. The value passed for the ClientHandle parameter is the value supplied by the application when the channel was opened.

Return Value

The function returns ADI_DMA_RESULT_SUCCESS if the channel was successfully opened. Any other value indicates an error. Possible errors include:

ADI_DMA_RESULT_ALL_IN_USE	All channel memory is in use.
ADI_DMA_RESULT_CANT_HOOK_INTERRUPT	The system cannot hook a DMA data or error interrupt.

adi_dma_Queue

Description

This function queues a descriptor, or chain of descriptors, to the specified DMA channel.

When using descriptor chains, the descriptor is added to the end of the list of descriptors already queued to the channel, if any. The last descriptor in the chain must have its `pNext` pointer set to NULL.

Prototype

```
ADI_DMA_RESULT adi_dma_Queue(
        ADI_DMA_CHANNEL_HANDLE     ChannelHandle,
        ADI_DMA_DESCRIPTOR_HANDLE  DescriptorHandle
);
```

Arguments

`ChannelHandle`	This argument uniquely identifies the DMA channel that the descriptor is to be queued on and is the value returned when the DMA channel was opened.
`DescriptorHandle`	This is a pointer to the first descriptor in the chain.

Return Value

`ADI_DMA_RESULT_SUCCESS`	The descriptor was queued successfully.
`ADI_DMA_RESULT_BAD_HANDLE`	The `ChannelHandle` does not contain a valid channel handle.
`ADI_DMA_RESULT_BAD_DESCRIPTOR`	The descriptor handle is NULL.
`ADI_DMA_RESULT_ALREADY_RUNNING`	This argument cannot submit additional descriptors to a channel configured for a loopback with dataflow enabled.

adi_dma_SetMapping

Description

This function maps the DMA Channel ID to the given peripheral.

Prototype

```
ADI_DMA_RESULT adi_dma_SetMapping(
        ADI_DMA_PMAP            Peripheral,
        ADI_DMA_CHANNEL_ID      ChannelID
);
```

Arguments

`Peripheral`	This is the peripheral ID to which the DMA channel is to be mapped.
`ChannelID`	This is the channel ID that is to be mapped to the peripheral.

Return Value

`ADI_DMA_RESULT_SUCCESS`	The channel was successfully mapped.
`ADI_DMA_RESULT_BAD_PERIPHERAL`	A bad peripheral value was encountered.
`ADI_DMA_RESULT_ALREADY_RUNNING`	The mapping could not be performed as the channel is currently transferring data.

adi_dma_Terminate

Description

This function closes down all DMA activity and terminates the DMA Manager.

Prototype

```
ADI_DMA_RESULT adi_dma_Terminate(
    ADI_DMA_MANAGER_HANDLE      ManagerHandle,
);
```

Arguments

`ManagerHandle`	This is the handle to the DMA Manager.

Return Value

The function returns `ADI_DMA_RESULT_SUCCESS` if successful. Any other value indicates an error.

Public Data Structures, Enumerations and Macros

This section defines both the public data structures and enumerations used by the DMA Manager. These data structures are made available to client applications or device driver libraries via the header file, `adi_dma.h`. All types have the `ADI_DMA_` prefix to avoid ambiguity with client developers' own data types.

This section contains:

Data Types

Several data types that shield the client developer from the internal implementation of the library and the details of DMA programming are used. These data types also provide an interface that is partially decoupled from the functionality offered by individual processors.

ADI_DMA_CHANNEL_HANDLE

The `ADI_DMA_CHANNEL_HANDLE` type identifies each separate DMA channel to the DMA Manager. When passed to the DMA Manager function, it uniquely identifies the channel function to which it needs to refer or upon which it must operate. The DMA Manager returns this handle to the application when a DMA channel is opened. All other DMA Manager functions that need to identify a channel require this parameter to be passed.

ADI_DMA_DESCRIPTOR_UNION / ADI_DMA_DESCRIPTOR_HANDLE

The `ADI_DMA_DESCRIPTOR_UNION` data structure represents a union of the small descriptor, large descriptor, and descriptor array data types. The `ADI_DMA_DESCRIPTOR_HANDLE` is then a typedef that describes a pointer to the union. The `ADI_DMA_DESCRIPTOR_HANDLE` is passed into the `adi_dma_`

Queue() function as a means to provide the function with either a) a small descriptor chain, b) a large descriptor chain, or c) an array of descriptors. By using the handle/union, only a single adi_dma_Queue() function is needed, rather than separate functions for each of the descriptor data types.

```
typedef union ADI_DMA_DESCRIPTOR_UNION {
   ADI_DMA_DESCRIPTOR_SMALL         Small;
   ADI_DMA_DESCRIPTOR_LARGE         Large;
   ADI_DMA_DESCRIPTOR_ARRAY         Array;
} ADI_DMA_DESCRIPTOR_UNION;
typedef ADI_DMA_DESCRIPTOR_UNION  *ADI_DMA_DESCRIPTOR_HANDLE;
```

ADI_DMA_STREAM_HANDLE

The ADI_DMA_STREAM_HANDLE type identifies a memory stream to the DMA Manager. When passed to the adi_dma_MemoryXXX functions, the handle uniquely identifies the memory stream onto which the DMA Manager is to operate. The DMA Manager returns this handle to the application when a DMA memory stream is opened. All other memory stream functions require this parameter to be passed.

Data Structures

The structures that define each type of descriptor and the DMA Configuration register are available in the public header file, adi_dma.h. The field names follow the convention used in the hardware reference manual for the appropriate processor.

ADI_DMA_2D_TRANSFER

The ADI_DMA_2D_TRANSFER data structure defines the characteristics of either the source or destination component of a two-dimensional memory copy.

```
typedef struct ADI_DMA_2D_TRANSFER {
    void                        *StartAddress;
    u16                         XCount;
    s16                         XModify;
    u16                         YCount;
    s16                         YModify;
} ADI_DMA_2D_TRANSFER;
```

ADI_DMA_CONFIG_REG

The ADI_DMA_CONFIG_REG type defines the structure for the DMA Configuration Control Word. In addition, macros are provided to allow the client to set individual fields within the word.

ADI_DMA_DESCRIPTOR_ARRAY

The ADI_DMA_DESCRIPTOR_ARRAY structure defines the contents of a descriptor array element:

```
typedef struct ADI_DMA_DESCRIPTOR_ARRAY {
    void                        *StartAddress;
    ADI_DMA_CONFIG_REG          Config;
    u16                         XCount;
    s16                         XModify;
    u16                         YCount;
    s16                         YModify;
} ADI_DMA_DESCRIPTOR_ARRAY;
```

ADI_DMA_DESCRIPTOR_LARGE

The `ADI_DMA_DESCRIPTOR_LARGE` structure defines the contents of a large descriptor.

```
typedef struct ADI_DMA_DESCRIPTOR_LARGE {
    struct ADI_DMA_DESCRIPTOR_LARGE    *pNext;
    void                               *StartAddress;
    ADI_DMA_CONFIG_REG                 Config;
    u16                                XCount;
    s16                                XModify;
    u16                                YCount;
    s16                                YModify;
} ADI_DMA_DESCRIPTOR_LARGE;
```

ADI_DMA_DESCRIPTOR_SMALL

The `ADI_DMA_DESCRIPTOR_SMALL` structure defines the contents of a small descriptor:

```
typedef struct ADI_DMA_DESCRIPTOR_SMALL {
    u16                         *pNext;
    u16                         StartAddressLow;
    u16                         StartAddressHigh;
    ADI_DMA_CONFIG_REG          Config;
    u16                         XCount;
    s16                         XModify;
    u16                         YCount;
    s16                         YModify;
} ADI_DMA_DESCRIPTOR_SMALL;
```

General Enumerations

The enumerations control and provide feedback for the operation of the DMA Manager.

ADI_DMA_CHANNEL_ID

The `ADI_DMA_CHANNEL_ID` enumeration contains values for each and every DMA channel of the processor. This value is used in the `adi_dma_Open()` function to identify which channel is to be opened. The specific enumeration values are dependent on the specific processor being targeted.

ADI_DMA_EVENT

The `ADI_DMA_EVENT` enumeration describes the types of events that can be reported to the client's callback function. Associated with the `ADI_DMA_EVENT` parameter is another parameter that points to the companion argument, `pArg`, for the event.

Value	Event	Companion Argument
`ADI_DMA_EVENT_DESCRIPTOR_PROCESSED`	A descriptor has completed processing or a memory stream has completed a memory copy operation.	The address of the descriptor just processed, or NULL if the event is a memory stream completion event.
`ADI_DMA_EVENT_INNER_LOOP_PROCESSED`	A sub buffer has completed processing.	The start address of the circular buffer.
`ADI_DMA_EVENT_OUTER_LOOP_PROCESSED`	The entire circular buffer has completed processing.	
`ADI_DMA_EVENT_ERROR_INTERRUPT`	DMA error interrupt has been generated.	NULL

ADI_DMA_MODE

The `ADI_DMA_MODE` enumeration defines how a channel is to process the data to be transferred. It takes the following values:

`ADI_DMA_DATA_MODE_UNDEFINED`	Undefined
`ADI_DMA_DATA_MODE_SINGLE`	This is a single one-shot buffer.
`ADI_DMA_DATA_MODE_CIRCULAR`	This is a single circular buffer.
`ADI_DMA_DATA_MODE_DESCRIPTOR_ARRAY`	This is an array of descriptors.

`ADI_DMA_DATA_MODE_DESCRIPTOR_SMALL`	This is a chain of small descriptors.
`ADI_DMA_DATA_MODE_DESCRIPTOR_LARGE`	This is a chain of large descriptors.

ADI_DMA_PMAP

The `ADI_DMA_PMAP` enumeration defines each of the DMA supported on-chip peripherals of the processor. This value is used to detect and set the mappings of on-chip peripherals to DMA channels using the `adi_dma_GetMapping()` and `adi_dma_SetMapping()` functions. The specific enumeration values are dependent on the specific processor being targeted.

ADI_DMA_RESULT

All public DMA Manager functions return a result code of the enumeration type, `ADI_DMA_RESULT`. Possible values are:

`ADI_DMA_RESULT_SUCCESS`	Generic success is reported.
`ADI_DMA_RESULT_FAIL`	Generic failure is reported.
`ADI_DMA_RESULT_BAD_DEVICE`	A bad device information was received.
`ADI_DMA_RESULT_BAD_HANDLE`	A bad device handle was encountered.
`ADI_DMA_RESULT_BAD_DESCRIPTOR`	A bad descriptor was encountered.
`ADI_DMA_RESULT_BAD_MODE`	A bad channel mode was encountered.
`ADI_DMA_RESULT_IN_USE`	Channel is already in use.
`ADI_DMA_RESULT_ALREADY_RUNNING`	DMA is already running.
`ADI_DMA_RESULT_NO_BUFFER`	Channel has no buffer.
`ADI_DMA_RESULT_BAD_COMMAND`	Invalid `Config` item was received.
`ADI_DMA_RESULT_NO_MEMORY`	A memory to channel object cannot be assign.

ADI_DMA_STREAM_ID

The `ADI_DMA_STREAM_ID` enumeration contains values for every DMA channel of the processor. This value is used in the `adi_dma_Open()` function to identify which channel is to be opened. The specific enumeration values are dependent on the specific processor being targeted.

ADI_DMA_CONFIG_REG Field Values

These values are to be used to set the relevant bits in the DMA Configuration word.

ADI_DMA_DMA2D

`ADI_DMA_LINEAR`	Linear buffer
`ADI_DMA_2D`	2D DMA operation

ADI_DMA_DI_EN

`ADI_DMA_DI_EN_DISABLE`	Disable callbacks on completion.
`ADI_DMA_DI_EN_ENABLE`	Enable callbacks on completion.

ADI_DMA_DI_SEL

`ADI_DMA_DI_SEL_OUTER_LOOP`	Callback after completing whole buffer (default).
`ADI_DMA_DI_SEL_INNER_LOOP`	Callback after completing each inner loop.

ADI_DMA_EN

`ADI_DMA_DISABLE`	Disable DMA transfer on the channel.
`ADI_DMA_ENABLE`	Enable DMA transfer on the channel.

ADI_DMA_WDSIZE

`ADI_DMA_8BIT`	8-bit words
`ADI_DMA_16BIT`	16-bit words
`ADI_DMA_32BIT`	32-bit words

ADI_DMA_WNR

`ADI_DMA_READ`	Transfer from memory to peripheral.
`ADI_DMA_WRITE`	Transfer from peripheral to memory.

DMA Commands

DMA channels and memory streams can be controlled via calls to the `adi_dma_Command()` function. Table 6-2 describes the commands and values that can be issued via the function:

Table 6-2. DMA Commands

Command ID	Value	Description
`ADI_DMA_CMD_TABLE`	`ADI_DMA_CMD_VALUE_PAIR *`	Pointer to a table of commands
`ADI_DMA_CMD_PAIR`	`ADI_DMA_CMD_VALUE_PAIR *`	Pointer to a single command pair
`ADI_DMA_CMD_END`	NULL	Signifies end of table
`ADI_DMA_CMD_SET_LOOPBACK`	TRUE/FALSE	Enables/disables loopback
`ADI_DMA_CMD_SET_STREAMING`	TRUE/FALSE	Enables/disables streaming
`ADI_DMA_CMD_SET_DATAFLOW`	TRUE/FALSE	Enables/disables dataflow
`ADI_DMA_CMD_FLUSH`	n/a	Flushes all buffers and descriptors on a channel
`ADI_DMA_CMD_GET_TRANSFER_STATUS`	`u32 *`	Provides the transfer status, TRUE - in progress, FALSE - not in progress
`ADI_DMA_CMD_TC_SET_DCB`	`u16`	Sets the traffic control DCB value
`ADI_DMA_CMD_TC_SET_DEB`	`u16`	Sets the traffic control DEB value

Table 6-2. DMA Commands (Cont'd)

Command ID	Value	Description
ADI_DMA_CMD_TC_SET_DAB	u16	Sets the traffic control DAB value
ADI_DMA_CMD_TC_SET_MDMA	u16	Sets the traffic control MDMA value
ADI_DMA_CMD_TC_GET_DCB_COUNTER	u16*	Gets the traffic control DCB counter value
ADI_DMA_CMD_TC_GET_DEB_COUNTER	u16*	Gets the traffic control DEB counter value
ADI_DMA_CMD_TC_GET_DAB_COUNTER	u16*	Gets the traffic control DAB counter value
ADI_DMA_CMD_TC_GET_MDMA_COUNTER	u16*	Gets the traffic control MDMA counter value

7 DEVICE DRIVER MANAGER

This chapter describes the Analog Devices, Inc. Device Driver Model.

The device driver model is used to control devices, both internal and external, to ADI processors. This includes on-board peripherals, such as SPORTs and Parallel Peripheral Interface (PPI), and off-chip connected devices such as codecs and converters.

This chapter contains:

- "Device Driver Model Overview" on page 7-3
  provides a general overview of the functionality provided by the device driver model and a brief description of the overall device driver architecture.
- "Using the Device Manager" on page 7-5
  describes how applications should invoke and interact with the device driver model including an explanation of the different data-flow methods that are supported in the model.
- "Device Manager Design" on page 7-18
  describes the Device Driver Manager API and inner workings of the Device Driver Manager. Specifically, this section describes how the Device Driver Manager operates and what it does in response to API calls and interaction with physical drivers.
- "Physical Driver Design" on page 7-35
  explains how physical drivers can be written to comply with the model and describes how physical device drivers interact with the Device Driver Manager.

- "Device Manager API Reference" on page 7-47
  describes the API functions of the Device Driver Manager.
- "Physical Driver API Reference" on page 7-57
  describes the API used between the Device Driver Manager and each physical driver.
- "Examples" on page 7-64
  provides PPI Driver and UART Driver code examples.

The interface from the application to the device driver provides a consistent, simple and familiar API to most programmers. While there is always some level of overhead involved in any standardization type effort, the benefits of a unified model far outweigh any minor inefficiencies. The model makes it relatively simple to create a new device driver, allows applications to largely insulate themselves from any device driver specifics and allows the device drivers to maximize use of any hardware features.

It is not expected that this model will be universally acceptable. There will always be devices that do not fit into the model, or applications that want to work with a device in some unique manner, and so on. The objective of this model is to provide a simple, efficient framework that will work for the majority of applications.

All sources to the device driver model are included in the various distributions of the model. While it is not expected that the sources will need to be modified or tailored to any specific application, they are provided in order for the user to fully understand how the code works.

While the terms "Device Manager" and "physical driver" refer to the respective software components, the term "device driver" is also used in this document. The term "device driver" refers to the combination of the Device Driver Manager (called "Device Manager" in this book) and physical driver.

Device Driver Model Overview

The device driver model is built using a hierarchical approach. Figure 7-1 illustrates the various components of the system design.

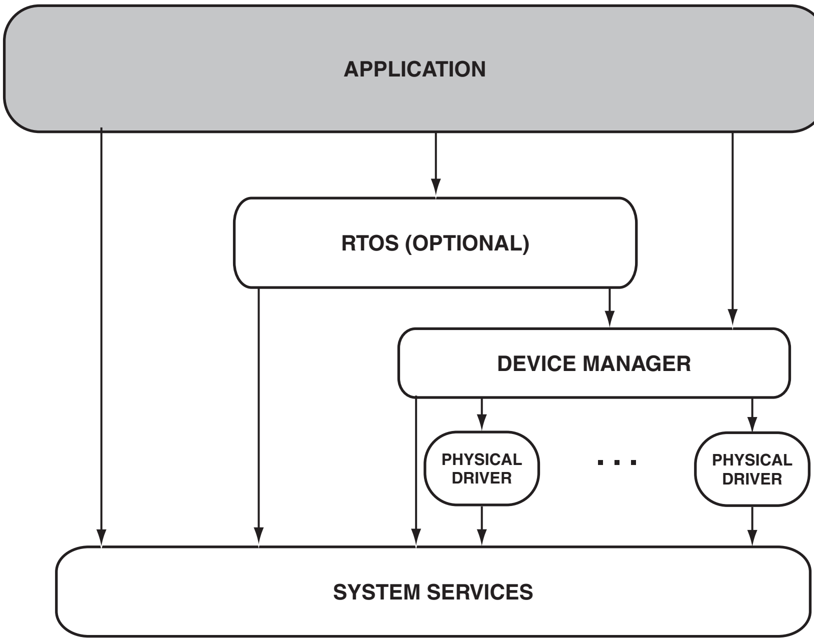

Figure 7-1. System Design and Hierarchy

The components shown above are:

- **Application** – Though typically the user's application, this block can be any software component that can be thought of as a client of the Device Manager. Note that the client does not have to be a single functional block. The Device Manager can support any number of clients. For example, a client may be a single user application or the client may be any number of tasks in RTOS-controlled systems.

- **RTOS** – Some systems use the services of a Real-Time Operating System (RTOS). The device driver model is not tailored to a particular RTOS nor does it require the presence of an RTOS. The device driver model does not require any functionality or services from an RTOS. Some RTOSs require that applications go through the RTOS in order to access device drivers. In these systems, the RTOS is simply viewed as a client to the Device Manager.
- **Device Manager** – The Device Manager provides the single point of access into the device driver model. The Device Manager provides the API into the model. All interaction between the client and device drivers occurs through the Device Manager. In addition to providing the API, the Device Manager ensures that the client makes call into the API in the proper sequence, performs synchronization services as needed, and controls all peripheral DMA, via the System Services DMA Manager, for devices that are supported by peripheral DMA.
- **Physical Drivers** – Physical device drivers provide the functionality necessary to control a physical device, for example any configurations register setting, device parameter setting, and so on. Physical drivers are responsible for hooking into the error interrupts for their device and processing them accordingly. If a device is not supported by peripheral DMA, the physical driver must provide the mechanism, a programmed I/O or the like, to move data through the device.
- **System Services** – The device driver components rely heavily on the functionality provided by the System Services. For example, the Device Manager relies on the Interrupt Manager and if required, the DMA Manager and Deferred Callback services. The functionality provided by the System Services is also available to physical

drivers to use. For example, a UART driver may need to know the SCLK frequency in order to configure the UART to operate at a specific baud rate. Through the Power Management service, the UART physical driver can ascertain the current SCLK frequency.

Both the device driver model and System Services are designed as portable software components. They are mainly written in "C", with some assembly code in critical sections. As such, software that interacts with the device driver model and system services must adhere to the C run-time model, calling conventions, passing parameters, and so on. Applications and physical drivers can be written in either C or assembly. Wherever possible, there are no dependencies on the code generation toolchain. System include files are not required nor are the services of the toolchain's run-time libraries. The device driver model and System Services can be built and run under any of the known code generation toolchains.

No dynamic memory allocation is used in the device driver model or System Services. Static memory allocation has been kept to a minimum and the vast majority of all data memory required is passed into the device driver model and System Services by the client or application. This allows the user to determine the amount of memory allocated and from which memory space, and the device driver model and System Services to use.

Using the Device Manager

The Device Manager provides the access point into the device driver model. The Device Manager presents the Device Manager API to the application or client.

This section contains:

- "Device Manager Overview" on page 7-6
- "Initialization Sequence" on page 7-16

Device Manager Overview

The Device Manager API consists of six functions:

- `adi_dev_Init` – Provides data and initializes the Device Manager.
- `adi_dev_Terminate` - Frees data and closes the Device Manager.
- `adi_dev_Open` – Opens the device for use.
- `adi_dev_Control` – Sets and detects device specific parameters.
- `adi_dev_Read` – Reads data from a device or queues reception buffers to a device.
- `adi_dev_Write` – Writes data to a device or queues transmission buffers to a device.
- `adi_dev_Close` – Closes the device.

In addition to the API functions into the Device Manager, the application provides the Device Manager with a callback function. Often, the Device Manager or physical driver encounters an event that needs to be passed to the user application. The event may be an expected event, such as an indication that the device driver has completed processing a buffer, or it may be an unexpected event, such as an error condition that has been generated by the device. All events are reported back to the application via a callback function. A callback function is simply a function within the user application that the Device Manager calls to pass along event information.

Theory of Operation

The device driver model is built around the concept that a device is used to move data into and/or out of the system. In most systems, a device is used to move data into the system, where the data will be processed in some fashion, and then another device will take the processed data and move it out of the system. Often, there are multiple devices running simultaneously in the system. The Device Manager provides a simple and straightforward interface regardless of how many devices are active at any one point in time and what the underlying implementation details are for each device.

Data

Data that is moved into or out of the device is encapsulated in a buffer. The Device Manager API defines three different types of buffers; a one-dimensional buffer called `ADI_DEV_1D_BUFFER`, a two-dimensional buffer called `ADI_DEV_2D_BUFFER`, and a circular (autobuffer type) buffer called `ADI_DEV_CIRCULAR_BUFFER`. Because physically moving data around uses valuable computing resources and has very little benefit, typically only pointers to buffers are passed between components. The Device Manager API defines the `ADI_DEV_BUFFER` data type as a pointer to a union of a one-dimensional buffer, a two-dimensional buffer and circular buffer. Though each of these types of buffers is processed differently, where there is no significant difference in processing, they are collectively referred to as simply a buffer within this text.

In general, applications provide buffers through the Device Manager API, where the buffers are processed, then made available again to the application. The `adi_dev_Read` function provides buffers to the device which are to be filled with data that is inbound from the device. The `adi_dev_Write` function provides buffers to the device that contain data to be sent out through the device.

Buffers are always processed in the order they are received. Buffers provided to a given device do not need to be a uniform size; each individual buffer can be any arbitrary size. Further, both one-dimensional and two-dimensional buffers can be provided to a single device. Circular buffers are a little more complex (see “Providing Buffers to a Device” on page 7-14).

Initializing the Device Manager

Before using a device, the application or client must first initialize the Device Manager. The client initializes the Device Manager by calling the `adi_dev_Init` function, passing a portion of memory to it that the Device Manager can use for processing. The client decides how much memory, and from which memory space, to provide to the Device Manager; the more memory that is provided, the more physical devices that can be simultaneously opened.

The Device Manager requires a contiguous block of memory that can be thought of in two parts—one part is base memory required for the Device Manager to instantiate itself, and the other is memory that is required to support n number of simultaneously-opened device drivers. Macros are provided that define the amount of memory (in bytes) that are required for the base memory and incremental device driver memory. These macros are `ADI_DEV_BASE_MEMORY` and `ADI_DEV_DEVICE_MEMORY`. For instance, if the client wanted to initialize the Device Manager and would have at most four device drivers open simultaneously at any point in time, the amount of memory required is:

```
ADI_DEV_BASE_MEMORY+(ADI_DEV_DEVICE_MEMORY*4)).
```

When called, the initialization function, `adi_dev_Init()`, initializes the memory that was passed in. Like all functions within the Device Manager, the initialization function returns a return code that indicates success or the specific error that occurred during the function call. All Device Manager API functions return the `ADI_DEV_RESULT_SUCCESS` value to indicate success. All error codes are in the following form: `ADI_DEV_RESULT_XXXX`.

In addition to the return code, the `adi_dev_Init()` function returns a count of the number of device drivers it can manage simultaneously, and a handle into the Device Manager. The device count can be tested to ensure the Device Manager can control the requested number of device drivers.

Another parameter passed to the `adi_dev_Init()` function is a critical region parameter. When it is necessary to protect a critical region of code, the Device Manager and all physical drivers leverage the Interrupt Manager system service to protect the critical code. The critical region parameter passed into the `adi_dev_Init()` function is, in turn, passed to the `adi_int_ProtectCriticalRegion()` function. See "adi_int_Init" on page 2-18 in the Interrupt Manager chapter.

Termination

When the Device Manager is no longer needed, the client can terminate the Device Manager using the `adi_dev_Terminate()` function. This function is passed the Device Manager handle, given to the client in the `adi_dev_Init()` function. The Device Manager closes any open physical devices and then returns to the caller. After the return from the `adi_dev_Terminate()` function, the memory that was supplied to the Device Manager via the `adi_dev_Init()` function can be reused by the client. Once terminated, the Device Manager must be re-initialized in order to be used again.

ⓘ Note that in many embedded systems, the Device Manager is never terminated.

Opening a Device

After the Device Manager has been initialized, in order to use a device, the client must first open the device with the `adi_dev_Open` API function.

The client passes in parameters indicating which device driver it wants to open (the `pEntryPoint` parameter), which instance of the device it wants to open (the `DevNumber` parameter), and the direction it wants data to flow (inbound, outbound or both), and so on. The client also passed in the handle to the DMA service the Device Manager and physical drivers should use. This parameter can be NULL if the client knows DMA is not used.

The `pDeviceHandle` parameter points to a location where the Device Manager stores the handle to the device driver that is being opened. All subsequent API calls for this device that is being opened must include this handle. The `ClientHandle` is a parameter that the Device Manager passes back to the client with each call to the client's callback function.

Two other parameters are passed into the `adi_dev_Open` function are also callback related. The `DCBHandle` parameter is a handle to the deferred callback service that the device driver uses to call the client's callback function. If `DCBHandle` is non-NULL, the device driver uses the specified Deferred Callback Service for all callbacks. If the `DCBHandle` is NULL, all callbacks are live; meaning that they are not deferred and are executed immediately, typically at interrupt time. The `ClientCallback` parameter points to the client's callback function.

The callback function is called in response to asynchronous events experienced by the device driver. Some events may be expected, such as the completion of processing of a buffer, and some events may be unexpected, such as the device generating an error condition. Regardless of the type of event, the Device Manager calls the callback function to notify the client of the event.

Note that dataflow through a device does not start with the `adi_dev_Open` function. This function simply opens the device for use as the device may need to be configured in some way before dataflow is enabled.

Configuring a Device

The `adi_dev_Control` function is used to configure and enable/disable dataflow through a device. When opened, most device drivers initialize with some default settings. If these default settings are sufficient for the application, then little or no application configuration is required. Other times, the default settings may not be appropriate for an application and so the device needs some amount of configuring. The `adi_dev_Control` function is used to set and detect device-specific configurations.

When configuration settings need to be set or detected, the client calls the `adi_dev_Control` function to set or detect the parameter. The `adi_dev_Control` function takes as parameters the `DeviceHandle` described in "Opening a Device", a command ID that identifies the parameter to be set or detected, and a pointer to the memory location that contains the value of the parameter to be set or where the value of the parameter being detected is stored. The Device Manager defines some standard parameters; however, physical drivers are free to add their own command IDs beyond those defined by the Device Manager. For example. the physical device driver for a DAC may create a command ID to set the volume level of the output. The application developer should check with the physical driver documentation to determine what parameters are configurable and what the configuration choices are.

Regardless of whether or not the client needs to make configuration changes, the client is required to make two calls into the `adi_dev_Control` function. These calls set the dataflow method of the device and enable dataflow for the device. These are described in the following sections.

Dataflow Method

The Device Manager supports three dataflow methods: circular, chained, and chained with loopback. Prior to providing the Device Manager with any buffers or enabling dataflow, the application must tell the Device Manager which dataflow method to use by calling the `adi_dev_Control` function with the `ADI_DEV_CMD_SET_DATAFLOW_METHOD` command. Only after the dataflow method has been defined can the client provide inbound buffers (via the `adi_dev_Read` function) or outbound buffers (via the `adi_devWrite` function) to the device.

As shown in Figure 7-2, the circular dataflow method defines the method whereby a single circular buffer is provided to the Device Manager, assuming the device was opened for unidirectional traffic.

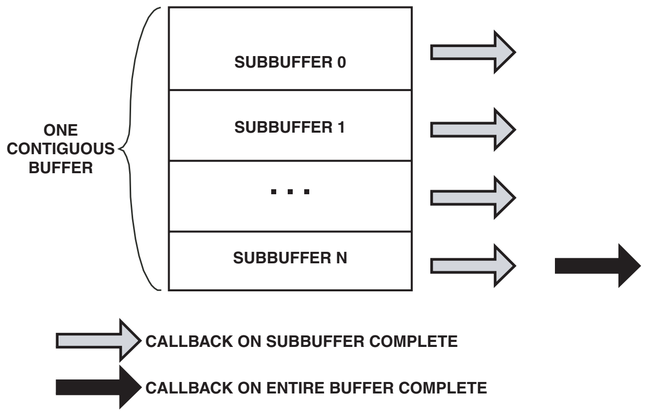
ONE CONTIGUOUS BUFFER
SUBBUFFER 0
SUBBUFFER 1
. . .
SUBBUFFER N
CALLBACK ON SUBBUFFER COMPLETE
CALLBACK ON ENTIRE BUFFER COMPLETE

Figure 7-2. Circular Buffer Operation

When providing the Device Manager with the circular buffer, the application tells the Device Manager how many sub buffers are within the circular buffer; two sub buffers are used for a traditional "ping/pong" scheme, though Blackfin processors support any number of sub buffers. The application also tells the Device Manager when it wants to be called back during processing of the circular buffer.

Three options are provided: no callback ever, a callback after each sub buffer is processed, or a callback after the entire buffer has been processed. The Device Manager begins processing at the start of the buffer. If so directed, the Device Manager notifies the application via the callback function when each sub buffer completes or when the entire buffer has completed processing. After reaching the end of the buffer, the Device Manager automatically restarts processing at the top of the buffer and so on.

As shown in Figure 7-3, with the chained dataflow method, one or more one-dimensional and/or two-dimensional buffers are provided to the Device Manager. Any number of buffers can be provided; buffers can be of different sizes and both one-dimensional and two-dimensional buffers can be provided to the same device. Each buffer, any one, none or all buffers can be tagged to generate a callback to the application when they are processed. Additional buffers can be provided at any time before or after dataflow has been enabled. The Device Manager guarantees to process the buffers in the order they are provided to the Device Manager.

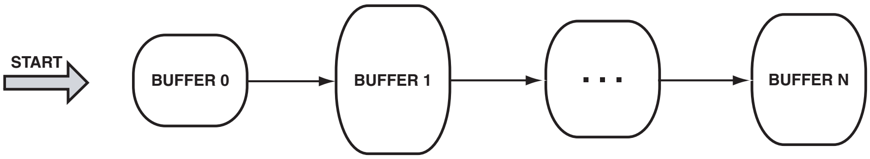

Figure 7-3. Chained Buffers

When using the chained dataflow method, the application can command the Device Manager to operate in synchronous mode. Normally, the Device Manager operates in asynchronous mode. In asynchronous mode, the `adi_dev_Read` and `adi_dev_Write` function calls return immediately to the application before all the buffers passed to the `adi_dev_Read` or `adi_dev_Write` function have been processed. In synchronous mode, the `adi_dev_Read` and `adi_dev_Write` functions do not return back to the application until all buffers provided to the `adi_dev_Read` or `adi_dev_Write` function have been processed. Though seldom used in real-time systems, the Device Manager supports the synchronous operating mode.

As shown in Figure 7-4, the chained with loopback method is similar to the chained dataflow method except that after the Device Manager has processed the last buffer, it automatically loops back to first buffer that was provided to the device. This operation effectively creates an infinite loop of buffers. With the chained with loopback method, the application can provide the buffers at initialization time, let the Device Manager process the buffers, and never have to re-supply the Device Manager with additional buffers. As with the chained dataflow method, each buffer, any one, none or all buffers can be tagged to generate a callback to the application when they are processed.

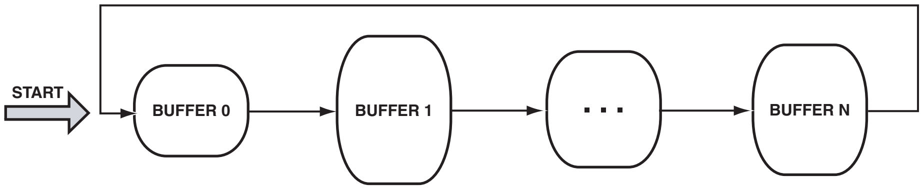

Figure 7-4. Chained Buffers with Loopback

Enabling Dataflow

Once the dataflow method has been defined, and buffers provided to the device as appropriate (see "Providing Buffers to a Device"), the application should enable dataflow by calling the `adi_dev_Control` function with the `ADI_DEV_CMD_SET_DATAFLOW` command. Dataflow starts immediately so the application should ensure that, if not using synchronous mode, devices that have been opened for inbound or bidirectional data have been provided with buffers, or else data may be lost.

Providing Buffers to a Device

Buffers are provided to a device via the `adi_dev_Read` and `adi_dev_Write` API function calls. The `adi_dev_Read` function provides buffers for inbound data, `adi_dev_Write` for outbound data. How the client provides buffers to the device via these API calls is slightly different depending on the dataflow method that has been chosen.

When a device has been configured to use the circular dataflow method, the application provides the device driver with one and only one buffer for inbound data and/or one and only one buffer for outbound data. The data buffer that is provided points to a contiguous piece of memory corresponding to however many sub buffers the application wants to use.

For example, assume that the application wants to process data in 512-byte increments and wants to work in a traditional "ping/pong" type (two sub buffer) fashion. The application provides the device driver with a single data buffer 1024 bytes in length, consisting of two 512-byte sub buffers. By doing this, the device driver can be using 512 bytes of the buffer while the application can be using the other 512 bytes simultaneously. Another example would be an application that wants to process a standard NTSC video frame (525 lines with 1716 bytes per line). The data buffer that is provided to the Device Manager could be a contiguous piece of memory `900900` bytes in size (`525 * 1716`). The sub buffer count in this case is 525. Regardless of how many sub buffers are provided, with the circular dataflow method, once the buffer has been provided to the device driver, the application never needs to give the device another buffer as the same one is used indefinitely.

When a device has been configured to use the chained dataflow method, any number of one-dimensional and two-dimensional buffers can be provided to the device. Buffers can be given to the device one at a time or multiple buffers can be provided with a single call to `adi_dev_Read` and/or `adi_dev_Write`. The application can provide the device driver with additional buffers at any time, before or even after dataflow has been enabled. Assuming the device driver is running in asynchronous mode, any individual buffer, no buffers or all buffers can be flagged to generate a callback when the device driver has completed processing it. Each buffer can be of a different size and both one-dimensional and two-dimensional buffers can be provided to the same device.

Providing buffers to devices that have been configured with the chained with loopback dataflow method is identical to providing buffers to devices using the chained dataflow method, except that buffers can only by provided while dataflow is disabled.

Closing a Device

When the device is no longer needed by the client, the device should be closed via the `adi_dev_Close` API function call. The `adi_dev_Close` function terminates dataflow if it is enabled and frees up all resources, including DMA and others that were used by the device driver. Should the application need to re-use the device after it is closed, it can be re-opened via the `adi_dev_Open` function.

Callbacks

The Device Manager calls the application's callback function to notify the client of events that occur. Events may be expected events (such as completion of the processing of a buffer) or unexpected events (such as an error occurring on a device). Typically, the client's callback function is organized as the equivalent of a "C" switch statement, invoking the appropriate processing as required by the given event type. The Device Manager defines several events and physical drivers can add additional events as required by the device they are controlling.

Initialization Sequence

Because the Device Manager and physical drivers rely on the System Services, the System Services should be initialized prior to opening a device driver. For example, when opening a device driver, the Device Manager requires handles to the Deferred Callback and DMA services (assuming both are being used). As such, it is good practice to initialize and open the System Services before opening any device drivers. See "Initialization" on page 1-7 and "Termination" on page 1-9 for more information on initializing and terminating the device drivers and system services.

Stackable Drivers

It is possible to create drivers that call other drivers. For example, the Blackfin EZ-Kit board contains an AD1836 audio codec. The AD1836 codec has a control and status interface that is suitable for connection to an SPI port, while the AD1836 audio data is provided to/from the device by using a high speed serial line, in this case the SPORT peripheral. If a system was being developed where the AD1836 codec was the only device that would ever be connected to the processor, then a single physical device driver could be written that controls and manages both the SPI and SPORT.

Alternatively, and especially so if other peripherals are to share the SPI port (for example, separate SPI and SPORT drivers could be controlled by an AD1836 driver) that hierarchically sits above the SPI and SPORT driver, making calls into those physical drivers as necessary. In this stackable fashion, it is possible to create mode complex drivers such as the AD1836 driver or a TCP/IP stack driver that sits atop an Ethernet controller.

Device Manager Design

The Device Manager provides the single point of access into the device driver model. The Device Manager provides the application with the API into the device drivers. All interaction between the client and device drivers occurs through the Device Manager—**applications never communicate directly with a physical driver**. The Device Manager also provides all DMA control, sequencing, queuing, and so on, for devices that are supported by peripheral DMA.

This section contains:

- "Device Manager API Description" on page 7-18
- "Device Manager Code" on page 7-23

Users typically do not need to understand the design and implementation details of the Device Manager. This section is included for those users who want to have a deeper understanding of the design. This section is particularly useful however, for writers of physical drivers who can use this information to aid in the development of physical drivers.

Device Manager API Description

The macros, definitions and data structures defined by the Device Manager API are key to understanding the design of the Device Manager. The Device Manager API is described in the file `adi_dev.h`. This file is located in the `Blackfin/Include/Drivers` directory

This section contains:

- "Memory Usage Macros" on page 7-19
- "Handles" on page 7-20
- "Dataflow Enumerations" on page 7-20

Memory Usage Macros

The first section in the `adi_dev.h` file contains macros that define the amount of memory usage by the Device Manager. These macros can be used by the client to determine how much memory should be allocated to the Device Manager via the `adi_dev_Init` function.

The `ADI_DEV_BASE_MEMORY` defines the number of bytes that the Device Manager needs. The `ADI_DEV_DEVICE_MEMORY` defines the number of bytes the Device Manager needs to control each physical driver. When providing memory to the Device Manager, the client should provide the following amount of memory.

```
ADI_DEV_BASE_MEMORY + (n * ADI_DEV_DEVICE_MEMORY)
```

Where "`n`" is the maximum number of physical drivers that are to be simultaneously opened in the system.

Handles

Next in the `adi_dev.h` file are `typedefs` for the various handle types that are used by the Device Manager. Handles are typically pointers to data structures that are used within the Device Manager. They are used as a means to identify the data pertaining to the device being managed quickly.

Dataflow Enumerations

Next in the `adi_dev.h` file are enumerations for the various dataflow methods supported by the Device Manager and enumerations indicating the dataflow direction. These enumerations are not extensible by physical drivers quickly.

Command IDs

The next section enumerates the command IDs that are defined by the Device Manager. These command IDs are passed to the Device Manager via the `adi_dev_Control` function.

Physical drivers can add any number of additional command IDs that are relevant to their particular device. Physical drivers begin adding their own command IDs beginning with the `ADI_DEV_CMD_PDD_START` enumeration value.

Also included in this section is a data structure defining a configuration command pair. This is provided as a convenience that allows clients to pass a table of commands into the `adi_dev_Control` function, rather than being forced to call the `adi_dev_Control` function for each command (see on page 7-27).

Callback Events

The next section in the `adi_dev.h` file contains enumerations for callback events. When an event occurs, the client's callback function is invoked and passed the enumeration of the event that occurred.

The Device Manager defines some common events. As with command IDs, physical drivers can add their own callback events beginning with the `ADI_DEV_EVENT_PDD_START` enumeration value.

Return Codes

The next section in the `adi_dev.h` file contains enumerations for return codes. All API functions within the Device Manager return a code indicating the results of the function call.

The Device Manager defines some typical return codes. As with command IDs and callback events, physical drivers can add their own return codes beginning with the `ADI_DEV_RESULT_PDD_START` enumeration value.

Circular Buffer Callback Options

The next section in the `adi_dev.h` file contains enumerations for the type of callback the client requests when the Device Manager is using the circular buffer dataflow method.

Enumerations are provided indicating the Device Manager should make no callbacks, make a callback on sub buffer completion, or make a callback on whole buffer completion.

Buffer Data Types

The `ADI_DEV_1D_BUFFER`, `ADI_DEV_2D_BUFFER` and `ADI_DEV_CIRCULAR_BUFFER` data structures are used to provide data buffers to the driver. At the top of these data structures is a reserved area. This reserved area allows the device drivers access to a small amount of memory that is attached to each buffer. How, or even if, the device driver uses this reserved area is a matter that depends on the implementation.

Note that if the physical device driver is supported by peripheral DMA, the Device Manager uses this reserved area to create a DMA descriptor describing the buffer. This descriptor is in turn passed to the DMA Manager System Service in order to use DMA to move the data, as described in the buffer structure.

If the physical driver is not supported by peripheral DMA, the physical driver can use this reserved area for any purpose; for example, queue management, or whatever mechanism the physical driver uses to move the data.

Also included in this section is a data structure, `ADI_DEV_BUFFER`, which represents a union of one-dimensional, two-dimensional and circular buffers. This datatype is used as a convenient method to refer to a buffer in a generic fashion, without knowing the specific type of buffer. The API functions `adi_dev_Read` and `adi_dev_Write` use the `ADI_DEV_BUFFER` type when passing buffers to these functions.

Physical Driver Entry Point

The next section in the `adi_dev.h` file contains a data structure that describes the entry point into a physical driver. The `ADI_DEV_PDD_ENTRY_POINT` structure is simply a data type that points to the functions within the physical driver that are called by the Device Manager.

API Function Definitions

The last section in the `adi_dev.h` file describes the API calls into the Device Manager. Each function is declared here with the appropriate parameters for each call. Each function is described in detail in "API Functional Description" on page 7-24.

Device Manager Code

All code for the Device Manager is kept in the `adi_dev.c` file. This sections describes the code of the Device Manager. This file is located in the `Blackfin/Lib/Src/Drivers` directory.

This section contains:

- "Data Structures" on page 7-23
- "Static Data" on page 7-23
- "Static Function Declarations" on page 7-23
- "API Functional Description" on page 7-24

Data Structures

The only additional data structures that are defined are the `ADI_DEV_MANAGER` and `ADI_DEV_DEVICE` structures. These structures contain all the data necessary for operation of the Device Manager itself and for management and control of the physical driver.

Static Data

The Device Manager uses a single piece of static data. The `InitialDeviceSettings` item is copied into an `ADI_DEV_DEVICE` structure when a device is opened. This provides a quick and efficient means to initialize an `ADI_DEV_DEVICE` structure without having to populate each item individually.

Static Function Declarations

This section declares static functions that are used within the Device Manager. Each of these functions is described in detail below. Only the API functions are declared to be global, all other functions are static to the Device Manager.

API Functional Description

This section describes the functionality that is performed for each of the API functions in the Device Manager. The API functions include:

- `adi_dev_Init` – on page 7-24
- `adi_dev_Open` – on page 7-24
- `adi_dev_Close` – on page 7-25
- `adi_dev_Read` – on page 7-26
- `adi_dev_Write` – on page 7-27
- `adi_dev_Control` – on page 7-27

adi_dev_Init

The `adi_dev_Init` function is used to initialize the Device Manager.

ⓘ For detailed reference information, see "adi_dev_Init" on page 7-50.

Processing begins by checking to ensure enough memory was provided to operate the Device Manager. The function then determines how many physical devices can be controlled with the remaining memory provided.

The critical region pointers are then stored and the data structure for each device that can be supported is marked as available for use. The function then returns to the caller.

adi_dev_Open

The `adi_dev_Open` function is used to open a device for use.

ⓘ For detailed reference information, see "adi_dev_Open" on page 7-52.

Processing begins by finding a free `ADI_DEV_DEVICE` data structure to be used to control the device. The address of that data structure is stored in the client-provided location as the handle to the device.

The `ADI_DEV_DEVICE` structure is initialized and populated with the information describing the device.

Once the `ADI_DEV_DEVICE` structure has been initialized, the Device Manager calls the `adi_pdd_Open` function of the physical driver. The physical driver then executes, doing whatever it needs to do to open the device it controls. If for some reason the physical driver fails to open the device, the Device Manager frees up the `ADI_DEV_DEVICE` structure and returns the return code from the physical device back to the application. Note that because the return code values can be extended by the physical device, the return code can be as specific as possible as to why the device failed to open.

If the physical device opens correctly, the Device Manager interrogates the physical device to see if it is supported by peripheral DMA. The Device Manager saves this information in the `ADI_DEV_DEVICE` structure.

adi_dev_Close

The `adi_dev_Close` function is called by the application when the device is no longer needed.

For detailed reference information, see "adi_dev_Close" on page 7-48.

After the device handle has been validated, assuming error checking is enabled, the function calls the `adi_pdd_Control` function of the physical driver to terminate dataflow. Once dataflow has been terminated, any DMA channels that were opened for the device are closed. The `adi_pdd_Close` function of the physical driver is then called to shut down the device and free up any resources used by the physical device. Lastly, the `ADI_DEV_DEVICE` structure is flagged as closed so that it may be reused at some later point in time.

adi_dev_Read

The `adi_dev_Read` function is called by the application to provide the device with buffers into which inbound data is to be stored. Assuming error checking is enabled, processing begins in this function by validating the device handle, and insuring that the device has been opened for inbound, or bidirectional, traffic and that the dataflow method has already been defined. If the dataflow method has not yet been defined, the Device Manager does not have enough information to know what to do with the buffer.

For detailed reference information, see "adi_dev_Read" on page 7-54.

The `pBuffer` parameter passed into the function can point to a single buffer or a chain of buffers. Further, if the device is supported by peripheral DMA, the reserved area within the buffer data structure needs to be configured appropriately. All these details are taken care of in the `PrepareBufferList` static function (see on page 7-32 for more information on this function).

Once the buffer list has been prepared, a check is made to see if the device is supported by peripheral DMA. If so, the DMA Manager is called to queue the buffers on the proper DMA channel using the appropriate dataflow method; chained descriptors are passed to the DMA Manager via the `adi_dma_Queue` function, and circular buffers passed via the `adi_dma_Circular` function. If peripheral DMA is not supported, the buffers are passed directly to the physical driver using the `adi_pdd_Read` function. Note that when a device is supported by peripheral DMA, the physical driver is extremely simple as the Device Manager handles all data buffers for the physical device.

Lastly, a check is made to see if the device is operating in synchronous or asynchronous mode. If it is operating in asynchronous mode, the `adi_dev_Read` function returns to the application immediately. If it is operating in synchronous mode, the `adi_dev_Read` function waits in a loop until the buffer or the last buffer within the list of buffers (if multiple buffers were provided as a parameter) has been processed before returning to the application. Again, the physical driver has no knowledge of, nor the need for the synchronous/asynchronous mode information.

adi_dev_Write

The `adi_dev_Write` function operates virtually identically to the `adi_dev_Read` function, except the data is destined for the outbound rather than inbound direction.

For detailed reference information, see "adi_dev_Write" on page 7-56.

adi_dev_Control

The `adi_dev_Control` function is used to process configuration-type commands from the application. Like all the API functions, if error checking is enabled, the device handle is validated upon entry into the function.

For detailed reference information, see "adi_dev_Control" on page 7-49.

Processing within the `adi_dev_Control` function is based upon the command ID that is passed in as a parameter. Some commands can be processed entirely by the Device Manager, some commands are processed by the physical driver only, while others need to be processed by both the Device Manager and the physical driver. In order to accomplish this, the bulk of this function is designed as a "C" switch statement. Each command that the Device Manager cares about has an entry in the statement.

When a command is passed that the Device Manager needs to process, the Device Manager processes the command and then sets a flag stating whether or not the command needs to passed down to the physical driver. When processing gets to the bottom of the function, if the command needs to be passed to the physical driver, the `adi_pdd_Control` function of the physical driver is called and the return code from the physical driver is passed back to the application. This arrangement allows each physical driver to extend the command IDs and allow them to create their own unique command IDs that the application can control.

The Device Manager processes the following commands:

- `ADI_DEV_CMD_GET_2D_SUPPORT` – This command is used to determine whether or not the device supports two-dimensional data movement. On Blackfin processors, if a device is supported by peripheral DMA, then two-dimensional data movement is provided. If the device is not supported by peripheral DMA, the command is passed to the physical driver for determining if the physical driver can support 2D data.
- `ADI_DEV_CMD_SET_SYNCHRONOUS` – This command is used to put the Device Manager in synchronous mode for the given device. The only processing here is to set the flag in the `ADI_DEV_DEVICE` structure. This command is never passed to the physical driver as all synchronous activity is controlled by the Device Manager. Hiding this from the physical driver has the added benefit of physical drivers not caring, nor having to take special processing, to accommodate synchronous or asynchronous modes. The physical driver can operate in whatever manner is best suited to the device.
- `ADI_DEV_CMD_SET_DATAFLOW_METHOD` – This command is used to set the dataflow method for the given device. If the device is not supported by peripheral DMA, then the Device Manager takes no action other than making note of the dataflow method and passing the command along to the physical driver via the `adi_pdd_Control` function. If the device is supported by peripheral DMA, then the default value used for the DMA configuration control register is updated with settings appropriate for the dataflow method. Further, once the dataflow method has been defined by the application, the Device Manager then has enough information to open whatever DMA channels are necessary in support of the device. The physical driver is interrogated via the `adi_pdd_Control`

function as to which DMA controller and channel number the device has been assigned for inbound and/or outbound data. The DMA Manager is then accessed to open the appropriate channels with the appropriate modes, such as circular or chained descriptors. If the device is opened with the `ADI_DEV_MODE_CHAINED_LOOPBACK` dataflow method, the DMA Manager is so configured. Note that the `ADI_DEV_DEVICE` structure is kept updated with the appropriate information as to which controllers and channels are opened or closed, what the operating modes are, and so on.

- `ADI_DEV_CMD_SET_DATAFLOW` – This command is issued to enable or disable dataflow on a device. The logic involved to enable or disable dataflow is fairly complex and isolated in a static function called `SetDataflow` (see on page 7-34 for more information on this function).

- `ADI_DEV_CMD_SET_STREAMING` - This command is issued to enable or disable the streaming mode of the device driver. (To fully understand what the streaming mode operation entails, users should be familiar with the streaming capability of the DMA Manager System Service ("DMA Manager" on page 6-1). Though peripheral DMA support is not required of a device that supports streaming, devices that are supported by peripheral DMA automatically leverage the streaming capabilities of the DMA Manager.)

  When streaming mode is enabled, the device is configured to treat data coming into and/or out of the device as a continuous stream of data. This typically allows the device driver to transmit and receive data through the device at maximum speed.

  In order to use the streaming mode of the Device Manager, the application must ensure that the following conditions are met:

  - The device always has buffers to process and never runs out of buffers. This means that the application guarantees that devices that are opened for inbound or bidirectional dataflow always have a buffer in which to store data that is received and that devices that are opened for outbound or bidirectional dataflow always have a buffer to transmit out through the device.

- The system timing is such that the Device Manager can acknowledge and service callbacks for a buffer before a callback for another buffer on that same device and going in that same direction (inbound or outbound) is generated.

These conditions can be fairly easily met in most systems.

Static Functions

This section describes the static functions within the Device Manager that are used in support of the API functions.

PDDCallback

The `PDDCallback` function is called in response to events from the physical driver. After error checking the device handle, if error checking is enabled, the Device Manager simply passes these events back to the application.

Note that in this routine (and the `DMACallback` function) the Device Manager calls the client callback function directly, without concern for whether or not live callbacks are in effect. It can do this as the physical driver is passed the handle to the Deferred Callback Service as part of the `adi_pdd_Open` function. As such, if the Deferred Callback Service is being used, the invocation of the `PDDCallback` function in the Device Manager has already been deferred by the physical driver. In this way, the `PDDCallback` function can directly call the client's callback function.

DMACallback

The `DMACallback` function is called in response to DMA events from the DMA Manager for devices that are supported by peripheral DMA. Assuming error checking is enabled, the device handle is first validated. The function then determines what event has occurred and performs its processing based on the event type.

If the event indicates that a descriptor has been processed, the processed flag and processed count fields of the buffer are updated. The application's callback function is then invoked in order to notify the application of the event.

If the event indicates that DMA processing has generated the `ADI_DEV_EVENT_SUBBUFFER_PROCESSED` event, the function makes the appropriate callback into the application stating that a sub buffer has completed processing. If the event indicates that DMA processing has generated the `ADI_`

DEV_EVENT_BUFFER_PROCESSED event, the function makes the appropriate callback into the application stating that the whole buffer has completed processing.

The DMA Manager reports asynchronous DMA errors via the callback mechanism. There errors are in turn passed back to the client via its callback function.

Note that in this routine (and the PDDCallback function), the Device Manager calls the client callback function directly, without concern for whether or not live callbacks are in effect. It can do this as the DMA Manager is passed the handle to the Deferred Callback Service as part of the adi_dma_Open function. As such, if the Deferred Callback Service is being used, the invocation of the DMACallback function in the Device Manager has already been deferred by the DMA Manager. In this way, the dmaCallback function can directly call the client's callback function.

PrepareBufferList

The PrepareBufferList function prepares a single buffer or list of buffers for submission to the DMA Manager, if the device is supported by peripheral DMA, or the physical driver, if the device is not supported by peripheral DMA.

The function begins by determining the value of the direction field in the DMA Configuration Control register. Because the data structures for circular buffers, one-dimensional buffers and two-dimensional buffers are different, each must be treated separately.

If passed as a circular buffer, the function assumes there is only one buffer in the buffer list. For devices opened with the ADI_DEV_MODE_CIRCULAR dataflow method, only a single buffer should be provided so this is a valid assumption to make. The function configures the DMA Configuration Control register according to the parameters within the circular buffer data structure. The Configuration Control register is set to generate inner loop interrupts if the application wants to be called back when each sub buffer has completed processing, or is set to generate outer loop interrupts if the application wants to be called back when the entire buffer has completed processing, or neither if the application does not want any callbacks. The word size is set to the width of a data element in the buffer and the direction field is set appropriately. The function then returns to the caller.

If the buffer type passed into the function specifies one-dimensional or two-dimensional buffers, the processing is largely the same except where noted.

For each buffer passed in, the processed flag and processed count fields within the buffer structure are cleared. If the physical device is supported by peripheral DMA, the reserved area at the beginning of each buffer structure is converted into a large model descriptor. The descriptor is then configured according to the parameters within the buffer structure, including such things as buffer size, width of an element, data direction, whether or not it is one-dimensional or two-dimensional, and so on. The descriptor for each buffer in the chain is updated to point to the next descriptor, for the corresponding buffer, within the chain. The last descriptor in the chain, corresponding to the last buffer within the chain, is updated to point to NULL for the next descriptor. After processing is completed, a chain of buffers is established. All the buffers are appropriately initialized and the reserved area in each buffer contains a DMA descriptor for that buffer that in turn points to the DMA descriptor for the next buffer in the chain.

Lastly, if the device is opened for synchronous mode and peripheral DMA is supported, the last descriptor in the chain is forced to generate a callback from the DMA Manager to the Device Manager. This allows the Device Manager to acknowledge when the last buffer has been processed so that it can update the processed fields appropriately. The last descriptor also acts as the trigger that responds each time the `adi_dev_Read/adi_dev_Write` function returns back to the application.

SetDataflow

The `SetDataflow` function is called in response to the `ADI_DEV_CMD_SET_DATAFLOW` command being received by the `adi_dev_Control` API function. This function enables or disables dataflow according to the flag.

The `SetDataflow` function begins processing by ensuring the system is not trying to enable dataflow when it is already enabled or disable dataflow when it is already disabled. If this check is not performed, DMA and or the physical drivers would likely generate errors.

When dataflow is being disabled, the function first calls the `adi_pdd_Control` function of the physical driver to disable dataflow. If the device is using peripheral DMA, it is important to disable dataflow at the device first, before shutting down DMA. Once the physical driver has disabled dataflow, any and all DMA channels that were opened for the device are closed. This is affected by calls to the DMA Manager.

When dataflow is being enabled, if the device is supported by peripheral DMA, the function first enables dataflow on the DMA channels by making calls into the DMA Manager to enable dataflow on the channel or channels that have been opened for the device. After the dataflow on the DMA channels has been enabled, the function calls the `adi_pdd_Control` function of the physical driver to enable dataflow.

Physical Driver Design

The physical driver is that part of the driver that controls the hardware for the device. Only the physical driver has knowledge of the device's control and status registers, and the fields within those registers. Unlike the Device Manager, where there is only a single Device Manager in the system, there can be any number of physical drivers present in a system.

This section contains:

Physical Driver Design Overview

Under application control, only the Device Manager communicates with each of the physical device drivers. Applications never interact directly with a physical driver or vice versa. However, similar to the execution sequence that applications have with the Device Manager, the Device Manager controls the physical device drivers in much the same manner. The Device Manager opens, controls, and closes physical device drivers analogous to how the application opens, controls, and closes the Device Manager.

Each physical driver in the system is controlled independently from the other physical drivers in the system. While multiple physical drivers can exist simultaneously in a system, multiple physical drivers should never be controlling the same device.

In general, a physical driver should control all instances of a device within a system. For example, if there are four serial ports (SPORTS) in the system, a single physical driver for the SPORT peripheral should be capable of controlling all four serial ports individually and simultaneously.

The physical driver is responsible for hooking any and all interrupts as needed for the physical device. Many physical devices generate interrupts on error conditions. These interrupts should be caught by the physical driver and passed back up as an event via the callback mechanism. The Interrupt Manager provides a very simple, straightforward mechanism that should be used for all interrupt processing. This makes the task of porting device drivers to different operating environments, toolchains and operating systems very straightforward.

If a device is supported by peripheral DMA, the physical driver is greatly simplified as the Device Manager typically controls all DMA interaction, without any involvement from the physical driver. When a device is opened, the Device Manager interrogates the physical driver as to whether or not the device is supported by peripheral DMA. If the physical driver responds in the affirmative, the Device Manager controls all DMA activity via the DMA Manager API, including initialization, providing data buffers, callback mechanisms and so on. As such, the Device Manager never calls the `adi_pdd_Read` and `adi_pdd_Write` routines of a physical driver that is supported by peripheral DMA. Physical drivers for devices that are supported by peripheral DMA are quite simple to implement.

For devices that are not supported by peripheral DMA, physical drivers can still take advantage of the DMA Manager as memory DMA can be an effective strategy for reading/writing to devices that use programmed I/O. If directed to use deferred callbacks, physical drivers should use the services of the Deferred Callback Manager exclusively in order to post callbacks into the Device Manager. See "Deferred Callback Manager" for more information.

Physical drivers have their own API, which is accessed by the Device Manager. The sections below describe the API and functionality that should be provided by the physical driver.

Physical Device Driver API Description

The API into a physical device driver is similar to the API between the Device Manager and the application in that there is a function in the physical driver API that maps to each function in the Device Manager API, except for `adi_dev_Init`. These functions are all prefixed with `adi_pdd` and are defined in the `adi_dev.h` Device Manager's include file.

The physical device driver functions are encapsulated in a structure called `ADI_DEV_PDD_ENTRY_POINT`. Each physical driver exports an entry point structure. The application passes the address of this structure to the Device Manager as part of the `adi_dev_Open` function call. The Device Manager, in turn, uses this data structure to call the individual routines in the physical driver. This mechanism allows multiple physical drivers to exist in the same system without causing name space conflicts.

There are five functions in the physical driver API. These functions are described in the sections below. The API functions include:

- `adi_pdd_Open` – Opens a device for use.
- `adi_pdd_Close` – Closes a device.
- `adi_pdd_Read` – Provides buffers for reception of data from a device.
- `adi_pdd_Write` – Provides buffers containing data for transmission out the device.
- `adi_pdd_Control` – Configures the device.

Physical Driver Include File ("xxx.h")

The API for physical drivers is defined in the `adi_dev.h` include file of the Device Manager. However, physical drivers can extend some of the definitions and enumerations defined by the Device Manager. Additional command IDs, event IDs and return codes can be created by each physical driver. These extensible definitions are described below. These definitions are normally defined in an include file provided with the physical driver. For example, the PPI driver, whose code is contained in the file `adi_ppi.c` has a companion `adi_ppi.h` include file. The only contents of the include file are the extensible definitions that the physical driver is making available to the application.

Client applications should include the Device Manager `adi_dev.h` file, and the include file for each of the physical drivers they will be using. For example, a client application using the PPI physical driver should include the `adi_dev.h` and `adi_ppi.h` include files. The `adi_dev.h` include file and physical driver include files for all Analog Devices provided drivers are found in the `Blackfin/Include/Drivers` directory.

Extensible Definitions

The physical driver can define its own extensions to the command IDs, event IDs and return codes, beyond those already defined by the Device Manager in the `adi_dev.h` file.

The last defined command ID defined by the Device Manager is labeled `ADI_DEV_CMD_PDD_START`. Physical drivers can create any number of additional command IDs as long as they begin the enumeration higher than the `ADI_DEV_CMD_PDD_START` value. Applications can issue these command ID via the `adi_dev_Control` API function. When the `adi_dev_Control` function of the Device Manager sees a command ID greater than the `ADI_DEV_CMD_PDD_START` value, the Device Manager passes the call onto the physical driver's `adi_pdd_Control` function, passing along the parameters provided by the application. This gives the physical driver the option of creating additional command IDs that are relevant to the device being controlled.

For example, a physical driver for a DAC may define a command ID that allows the application to set or detect the output volume level for the DAC.

In a similar fashion, physical drivers can create additional event IDs that they can pass back to the application. The last event ID defined by the Device Manager is labeled `ADI_DEV_EVENT_PDD_START`. Physical drivers can create any number of additional event IDs as long as they begin the enumeration higher than the `ADI_DEV_EVENT_PDD_START` value. Physical drivers can send these events to the application via a callback to the Device Manager. When the Device Manager's `PDDCallback` function is passed an event ID greater than the `ADI_DEV_EVENT_PDD_START` value, it passes the event and parameters passed to the Device Manager's callback function along to the application. This gives the physical driver the option of creating additional event IDs that are relevant to the device being controlled. For example, a physical driver that is controlling a device that is detecting the level of a signal can create an event that notifies the application when the signal has reached some predetermined value.

Physical drivers can also return custom-defined error codes. The last return code defined by the Device Manager is entitled `ADI_DEV_RESULT_PDD_START`. Physical drivers can create any number of additional return codes as long as they begin the enumeration higher than the `ADI_DEV_RESULT_PDD_START` value. Physical drivers can return these error codes in response to any physical driver API function call from the Device Manager. The Device Manager routinely looks for the `ADI_DEV_RESULT_SUCCESS` error code. Anything other than `ADI_DEV_RESULT_SUCCESS` is interpreted to be an error of some kind. When a physical driver API function returns an error code not equal to `ADI_DEV_RESULT_SUCCESS`, the Device Manager passes the error code back to the application as the return value for the Device Manager API function that triggered the error. This gives the physical driver the option of creating additional return codes that are relevant to the device being controlled.

For example, a physical driver may return a unique error code in response to a command to affect a parameter on the device. The physical driver could return an error code that provides some high level of detail as to what caused the error.

ADI_DEV_PDD_ENTRY_POINT

The physical driver's include function needs to include a declaration of the entry point into the driver. This declaration should declare, as a global variable, the address of the entry point for the physical driver. The application passes the address of the entry point to the Device Manager when the device is opened. For example, the line

```
extern ADI_DEV_PDD_ENTRY_POINT PPIEntryPoint;
                // entry point to the PPI driver
```

in the PPI driver's include file tells the application to pass the variable `PPIEntryPoint` as the entry point parameter in the `adi_dev_Open` function call to open the PPI device driver.

Physical Driver Source ("xxx.c")

All functions within the physical driver source code, including the actual physical driver API functions, should be declared static so that they are not exposed to any other software component. The only global piece of code or data should be the entry point address. The entry point is a simple structure that contains the addresses of the physical driver API functions in the order shown below.

```
ADI_DEV_PDD_ENTRY_POINT PPIEntryPoint = {
      adi_pdd_Open,
      adi_pdd_Close,
        adi_pdd_Read,
        adi_pdd_Write,
        adi_pdd_Control
};
```

Source code for all Analog Devices supplied physical drivers is located in the `Blackfin/Lib/Src/Drivers` directory.

All code within the driver source should be in support of the five physical driver API functions. These functions and the logic that they need to provide are described below. All physical driver API functions must return an error code. The Device Manager checks the return code for every physical driver API call. If the physical driver returns anything other than `ADI_DEV_RESULT_SUCCESS`, it assumed to be some type of failure.

ⓘ Similar to what is implemented in the Device Manager, it is highly recommended that physical drivers implement some type of switchable error checking, ideally using the `ADI_DEV_DEBUG` macro. As a minimum, physical driver handles (`ADI_DEV_PDD_HANDLE`) should be validated in each API function.

adi_pdd_Open

The `adi_pdd_Open` function is called by the Device Manager in response to the application calling the `adi_dev_Open` function. Its purpose is to open the device for use.

ⓘ For detailed reference information, see "adi_pdd_Open" on page 7-60.

The `adi_pdd_Open` function should first verify that the device being requested is available for use and supports the data direction requested. Appropriate error codes should be returned should the device be unavailable or not support the requested direction.

The device being controlled should be initialized and flushed of any stray data or pending interrupts. Any interrupts that are required to be handled in support of the device should be hooked. For devices that are supported with peripheral DMA, typically only the error interrupt needs to be hooked. The Interrupt Manager of the System Services should be used for all hooking of interrupts. Enabling/disabling of interrupts through the System Interrupt Controller (SIC) should also be accomplished using the Interrupt Manager service calls.

The physical driver should save the handle to the callback service. If non-NULL, meaning that deferred callbacks are in use, the physical driver should invoke all callbacks through the service identified by the callback service handle. If NULL, meaning all callbacks should be live and not deferred, the physical driver should call the Device Manager's callback function directly when sending events.

The physical driver should also save the `ADI_DEV_PDD_HANDLE` value in the location provided by the Device Manager. The Device Manager passes this handle back to the physical driver in all other API function calls.

The `adi_pdd_Open` function should return `ADI_DEV_RESULT_SUCCESS` if successful.

adi_pdd_Control

The `adi_pdd_Control` function is called by the Device Manager in response to the application calling the `adi_dev_Control` function. Its purpose is to process configuration-type commands from the Device Manager and client application. Like all the API functions, if error checking is enabled, the routine should validate the physical driver handle upon entry into the function.

ⓘ For detailed reference information, see "adi_pdd_Control" on page 7-59.

Processing within the `adi_pdd_Control` function should be based upon the command ID that is passed in as a parameter. Of the command IDs enumerated by the Device Manager in the `adi_dev.h` file, as a minimum, physical drivers must process the following commands:

- `ADI_DEV_CMD_SET_DATAFLOW` – Turns on and off the flow of data through the device.
- `ADI_DEV_CMD_GET_PERIPHERAL_DMA_SUPPORT` – Responds with `TRUE` or `FALSE` if the device is supported by peripheral DMA. If the device is supported by peripheral DMA, the `adi_pdd_Control` function should also be prepared to respond to the following command IDs:
    - `ADI_DEV_CMD_GET_INBOUND_DMA_PMAP_ID` – Responds with the DMA peripheral map (PMAP) ID for the given device.
    - `ADI_DEV_CMD_GET_OUTBOUND_DMA_PMAP_ID` – Responds with the DMA peripheral map (PMAP) ID for the given device.

In most cases, the `adi_pdd_Control` function of the physical driver should be constructed similarly to a "C" style switch statement. Each command that the physical driver cares about, including the required command IDs listed above and any additional command IDs created by the physical driver itself, should have an entry in the statement. If the physical driver receives a command ID it does not understand, it should typically return the `ADI_DEV_RESULT_NOT_SUPPORTED` return code.

adi_pdd_Read

The `adi_pdd_Read` function is called by the Device Manager in response to the application calling the `adi_dev_Read` function. Its purpose is to fill buffers with inbound data that is received from the device. With all API functions, if error checking is enabled the routine should validate the physical driver handle is upon entry into the function.

For detailed reference information, see "adi_pdd_Read" on page 7-62.

For devices that are supported by peripheral DMA, the Device Manager manages all buffer queueing and reception. As a result, if the device is supported by peripheral DMA, the `adi_pdd_Read` function is never called by the Device Manager and no functionality need be provided by this routine. This greatly simplifies device drivers for devices that are supported by processor DMA. Physical drivers that are supported by peripheral DMA still need to provide this function but should simply return `ADI_DEV_RESULT_NOT_SUPPORTED` as this routine should never get called.

For devices that are not supported by peripheral DMA, the `adi_pdd_Read` function is passed one or more buffers that the application has provided for inbound data reception. The physical driver can choose to process the buffers immediately, or provide the logic and functionality to queue or somehow stage these buffers for use at some later point in time. However, the physical driver is required to process the buffers in the order in which they were received.

For some devices, it may not be possible or practical to completely fill a buffer with data. For example, consider an Ethernet driver. The Ethernet driver typically receives packets that vary in length. The application may know what the maximum size Ethernet packet is and provide the driver with buffers sized to the maximum packet size. The driver may then receive a packet from the network that is smaller than the maximum packet size. It would be impractical for the physical driver to wait until additional packets were received and completely fill the buffer before processing. So, it is the physical driver's option to decide when to consider a buffer fully processed. Each buffer has a processed flag and processed size flag that the physical driver should set, based on when it considers a buffer processed and how much valid data the buffer contains.

Also, any buffer can be flagged by the application for notification when the buffer has completed processing. If a buffer is not flagged for a call-back, the physical driver should not notify the Device Manager when the buffer has been processed. If, however, the buffer is flagged for a callback (once the buffer has been processed), the physical driver is obligated to set the processed flag and processed size field in the buffer, and notify the Device Manager via the Device Manager's callback function that was passed to the physical driver as a parameter in the `adi_pdd_Open` function call, that the buffer has completed processing.

adi_pdd_Write

The `adi_pdd_Write` function is called by the Device Manager in response to the application calling the `adi_dev_Write` function. Its purpose is to transmit the data within the buffers out through the device. For all API functions, if error checking is enabled, the routine validates the physical driver handle upon entry into the function.

For detailed reference information, see "adi_pdd_Write" on page 7-63.

As in the case for `adi_pdd_Read`, for devices that are supported by peripheral DMA, the Device Manager manages all buffer queueing and transmission. As a result, if the device is supported by peripheral DMA, the `adi_pdd_Write` function is never called by the Device Manager and no functionality need be provided by this routine. This greatly simplifies device drivers for devices that are supported by processor DMA. Physical drivers that are supported by peripheral DMA still need to provide this function but should simply return `ADI_DEV_NOT_SUPPORTED` as this routine should never get called.

For devices that are not supported by peripheral DMA, the `adi_pdd_Write` function is passed one or more buffers that the application has provided for transmission out through the device. The physical driver can choose to immediately process the buffers, or provide the logic and functionality to queue or somehow stage these buffers for transmission at some later point in time. The physical driver is required, however, to process the buffers in the order in which they were received.

Each buffer has a processed flag and processed size flag that the physical driver should set based on when it considers a buffer processed and how much data was transmitted out through the device. Unlike in the `adi_pdd_Read` case, it is expected that the entire contents of the buffer will be transmitted.

Also, any buffer can be flagged by the application for notification when the buffer has completed processing. If a buffer is not flagged for a callback, the physical driver should not notify the Device Manager when the buffer has been processed. However, if the buffer is flagged for a callback, once the buffer has been processed the physical driver is obligated to set the processed flag and processed size field in the buffer and notify the Device Manager via the Device Manager's callback function that was passed to the physical driver as a parameter in the `adi_pdd_Open` function call, that the buffer has completed processing.

adi_pdd_Close

The `adi_pdd_Close` function is called by the Device Manager in response to the application calling the `adi_dev_Close` function. Its purpose is to gracefully shutdown and idle the device. For all API functions, if error checking is enabled, the routine should validate the physical driver handle is upon entry into the function.

For detailed reference information, see "adi_pdd_Close" on page 7-58.

After validating the driver handle, the `adi_pdd_Close` function should terminate all data transmission and reception if is not already stopped, as it is possible for the application to call the `adi_dev_Close` function while dataflow is enabled.

The function should idle the device and leave the device in a state such that it can be opened again should the application re-open the device at some later point in time. All resources that were allocated in support of the device should be released. For example if an error interrupt was hooked during the `adi_pdd_Open` function, it should be released as part of the `adi_pdd_Close` function.

Device Manager API Reference

This section describes the API of the Device Manager. The Device Manager API is defined in the `adi_dev.h` file.

Notation Conventions

The reference pages for the API functions use the following format:

Name and purpose of the function

Description – Function specification

Prototype – Required header file and functional prototype

Arguments – Description of function arguments

Return Value – Description of function return values

adi_dev_Close

Description

This function closes a device. Dataflow is stopped if it has not already been stopped and the device is put back into an idled state. After calling `adi_dev_Close`, the only way to access the device again is to first open it with the `adi_dev_Open` function call.

Prototype

```
ADI_DEV_RESULT adi_dev_Close(
               ADI_DEV_DEVICE_HANDLE    DeviceHandle);
```

Arguments

`DeviceHandle`	This is the handle used to identify the device.

Return Value

`ADI_DEV_RESULT_SUCCESS`	The device closed successfully.
`ADI_DEV_RESULT_BAD_DEVICE_HANDLE`	The device handle does not identify a valid device.
`ADI_DEV_RESULT_DMA_ERROR`	An error occurred while closing down DMA for the device.
`xxx`	This is a device-specific return code.

adi_dev_Control

Description

This function sets or detects a configuration parameter for a device.

Prototype

```
ADI_DMA_RESULT adi_dev_Control(
        ADI_DEV_DEVICE_HANDLE         DeviceHandle,
        u32                           Command,
        void                          *pArg
);
```

Arguments

`DeviceHandle`	This is the handle used to identify the device.
`Command`	This is the command identifier.
`pArg`	This is the address of command specific parameter.

Return Value

`ADI_DEV_RESULT_SUCCESS`	The function completed successfully.
`ADI_DEV_RESULT_BAD_DEVICE_HANDLE`	The device handle does not identify a valid device.
`ADI_DEV_RESULT_DMA_ERROR`	An error was reported while configuring the DMA Manager.
`ADI_DEV_RESULT_NOT_SUPPORTED`	The command is not supported.
`xxx`	This is a device-specific return code.

adi_dev_Init

Description

This function creates a Device Manager and initializes memory for the Device Manager. This function is typically called at initialization time.

Prototype

```
ADI_DEV_RESULT adi_dev_Init(
        void                        *pMemory,
        size_t                      MemorySize,
        u32                         *pMaxDevices,
        ADI_DEV_MANAGER_HANDLE      *pManagerHandle,
        void                        *pEnterCriticalParam
);
```

Arguments

`pMemory`	This is the pointer to an area of static memory to be used by the Device Manager.
`MemorySize`	This is the size in bytes of memory being supplied for the Device Manager.
`*pMaxDevices`	On return, this argument contains the number of simultaneously open devices that the Device Manager can support given the memory supplied.
`*pManagerHandle`	This is the pointer to memory location where the handle to the Device Manager will be stored.
`*pEnterCriticalParam`	This is the parameter that is to be passed to the function that protects critical areas of code.

Return Value

Return values include:

`ADI_DEV_RESULT_SUCCESS`	Device Manager was successfully initialized.
`ADI_DEV_RESULT_NO_MEMORY`	Insufficient memory has been supplied to Device Manager.

adi_dev_Open

Description

This function opens a device for use. Internal data structures are initialized, preliminary device control is established, and the device is reset and prepared for use.

Prototype

```
ADI_DEV_RESULT adi_dev_Open(
        ADI_DEV_MANAGER_HANDLE       ManagerHandle,
        ADI_DEV_PDD_ENTRY_POINT      *pEntryPoint,
        u32                          DeviceNumber,
        void                         *ClientHandle,
        ADI_DEV_DEVICE_HANDLE        *pDeviceHandle,
        ADI_DEV_DIRECTION            Direction,
        ADI_DMA_MANAGER_HANDLE       DMAHandle,
        ADI_DCB_HANDLE               DCBHandle,
        ADI_DCB_CALLBACK_FN          ClientCallback
);
```

Arguments

`ManagerHandle`	This is the handle to the Device Manager that controls the device.
`*pEntryPoint`	This is the address of the physical driver's entry point
`DeviceNumber`	This is the number identifying which device is to be opened. Device numbers begin with zero. For example, if there are four serial ports, they are numbered 0 through 3.
`*ClientHandle`	This is an identifier defined by the application. The Device Manager passes this value back to the client as an argument in the callback function.

`*pDeviceHandle`	This is the pointer to an application provided location where the Device Manager stores an identifier defined by the Device Manager. All subsequent communication initiated by the client to the Device Manager for this device includes this handle.
`Direction`	This is the data direction for the device, inbound, outbound or bidirectional.
`DMAHandle`	This is the handle to the DMA Manager service that is used for this device (can be NULL if DMA is not used.)
`DCBHandle`	This is the handle to the Deferred Callback Service that is used for this device. If NULL, all callbacks will be live and not-deferred.
`ClientCallback`	This is the address of the client's callback function.

Return Value

`ADI_DEV_RESULT_SUCCESS`	Device was opened successfully.
`ADI_DEV_RESULT_BAD_MANAGER_HANDLE`	The Device Manager handle does not point to a Device Manager.
`ADI_DEV_RESULT_NO_MEMORY`	Insufficient memory is available to open the device.
`ADI_DEV_RESULT_DEVICE_IN_USE`	The device is already in use.
`xxx`	This is a device-specific return code.

adi_dev_Read

Description

This function reads data from a device or queues reception buffers to a device.

Prototype

```
ADI_DEV_RESULT adi_dev_Read(
        ADI_DEV_DEVICE_HANDLE       DeviceHandle,
        ADI_DEV_BUFFER_TYPE         BufferType,
        ADI_DEV_BUFFER              *pBuffer
);
```

Arguments

DeviceHandle	This is the handle used to identify the device.
BufferType	This argument indicates the type of buffer: one-dimensional, two-dimensional or circular.
*pBuffer	This is the address of the buffer or first buffer in a chain of buffers.

Return Value

ADI_DEV_RESULT_SUCCESS	The function completed successfully.
ADI_DEV_RESULT_BAD_DEVICE_HANDLE	The device handle does not identify a valid device.
ADI_DEV_RESULT_DMA_ERROR	An error was reported while configuring the DMA Manager.
ADI_DEV_RESULT_DATAFLOW_UNDEFINED	The dataflow method has not yet been set.
xxx	This is a device-specific return code.

adi_dev_Terminate

Description

This function frees up all memory used by the Device Manager, stops data flow, closes all open device drivers, and terminates the Device Manager.

Prototype

```
ADI_DEV_RESULT adi_dev_Terminate(
        ADI_DEV_MANAGER_HANDLE       ManagerHandle
);
```

Arguments

`ManagerHandle`	This is the handle to the Device Manager.

Return Value

This function returns `ADI_DEV_RESULT_SUCCESS` if successful. Any other value indicates an error.

adi_dev_Write

Description

This function writes data to a device or queues transmission buffers to a device.

Prototype

```
ADI_DEV_RESULT adi_dev_Write(
        ADI_DEV_DEVICE_HANDLE       DeviceHandle,
        ADI_DEV_BUFFER_TYPE         BufferType,
        ADI_DEV_BUFFER              *pBuffer
);
```

Arguments

`DeviceHandle`	This is the handle used to identify the device.
`BufferType`	This arguments identifies the type of buffer: one-dimensional, two-dimensional or circular.
`*pBuffer`	This is the address of the buffer or first buffer in a chain of buffers.

Return Value

`ADI_DEV_RESULT_SUCCESS`	The function completed successfully.
`ADI_DEV_RESULT_BAD_DEVICE_HANDLE`	The device handle does not identify a valid device.
`ADI_DEV_RESULT_DMA_ERROR`	An error was reported while configuring the DMA Manager.
`ADI_DEV_RESULT_DATAFLOW_UNDE-FINED`	The dataflow method has not yet been set.
`xxx`	This is a device-specific return code.

Physical Driver API Reference

This section describes the API used between the Device Manager and each physical driver. The Physical Driver API is defined in the `adi_dev.h` file.

Notation Conventions

The reference pages for the API functions use the following format:

Name and purpose of the function

Description – Function specification

Prototype – Required header file and functional prototype

Arguments – Description of function arguments

Return Value – Description of function return values

adi_pdd_Close

Description

This function closes a device. Dataflow is stopped if it has not already been stopped and the device is put back into an idle state.

Prototype

```
ADI_DEV_RESULT adi_pdd_Close(
              ADI_PDD_DEVICE_HANDLE    PDDHandle);
```

Arguments

`PDDHandle`	This is the handle used to identify the device.

Return Value

`ADI_DEV_RESULT_SUCCESS`	The device closed successfully.
`xxx`	This is device-specific code.

adi_pdd_Control

Description

This function sets or detects a configuration parameter for a device.

Prototype

```
ADI_DMA_RESULT adi_pdd_Control(
        ADI_DEV_PDD_HANDLE          PDDHandle,
        u32                         Command,
        void                        *pArg
);
```

Arguments

`PDDHandle`	This is the handle used to identify the device.
`Command`	This is the command identifier.
`pArg`	This is the address of command-specific parameter.

Return Value

`ADI_DEV_RESULT_SUCCESS`	The function completed successfully.
`ADI_DEV_RESULT_NOT_SUPPORTED`	This command is not supported.
`xxx`	This is device-specific return code.

adi_pdd_Open

Description

This function opens a physical device for use. Internal data structures are initialized, preliminary device control is established and the device is reset and prepared for use.

Prototype

```
ADI_DEV_RESULT adi_ppd_Open(
        ADI_DEV_MANAGER_HANDLE       ManagerHandle,
        u32                          DeviceNumber,
        ADI_DEV_DEVICE_HANDLE        DeviceHandle,
        ADI_DEV_PDD_HANDLE           *pPDDHandle,
        ADI_DEV_DIRECTION            Direction,
        void                         *pEnterCriticalParam,
        ADI_DMA_MANAGER_HANDLE       DMAHandle,
        ADI_DCB_HANDLE               DCBHandle,
        ADI_DCB_CALLBACK_FN          DMCallback
);
```

Arguments

`ManagerHandle`	This is the handle to the Device Manager that is controlling the physical driver.
`DeviceNumber`	This is the number identifying which device is to be opened. Device numbers begin with zero. For example, if there are four serial ports, they are numbered 0 through 3.
`DeviceHandle`	This is the Device Manager-supplied parameter that uniquely identifies the device to the Device Manager.
`*pPDDHandle`	This is the pointer to a location where the physical driver stores a handle that uniquely identifies the device to the physical driver.
`Direction`	This is the data direction for the device, inbound, outbound or bidirectional

`*pEnterCriticalParam`	This is the parameter that is to be passed to the function that protects critical areas of code.
`DMAHandle`	This is the handle to the DMA Manager service that is used for this device (can be NULL if DMA is not used).
`DCBHandle`	This is the handle to the Deferred Callback Service that will be used for this device. If NULL, all callbacks will be live and not-deferred.
`DMCallback`	This is the address of the Device Manager's callback function.

Return Value

`ADI_DEV_RESULT_SUCCESS`	The device opened successfully.
`ADI_DEV_RESULT_DEVICE_IN_USE`	The Device Manager handle does not point to a Device Manager.
`xxx`	This is the device-specific return code.

adi_pdd_Read

Description

This function provides buffers to a device for reception of inbound data. This function is never called for devices that are supported by peripheral DMA.s

Prototype

```
ADI_DEV_RESULT adi_pdd_Read(
        ADI_DEV_PDD_HANDLE          PDDHandle,
        ADI_DEV_BUFFER_TYPE         BufferType,
        ADI_DEV_BUFFER              *pBuffer
);
```

Arguments

PDDHandle	This is the handle used to identify the device.
BufferType	This identifies the type of buffer: one-dimensional, two-dimensional or circular.
*pBuffer	This is the address of the buffer or first buffer in a chain of buffers

Return Value

ADI_DEV_RESULT_SUCCESS	The function completed successfully.
ADI_DEV_RESULT_BAD_PDD_HANDLE	The PDD handle does not identify a valid device.
xxx	This is the device-specific return code.

adi_pdd_Write

Description

This function provides buffers to a device for transmission of outbound data. This function is never called for devices that are supported by peripheral DMA.

Prototype

```
ADI_DEV_RESULT adi_pdd_Write(
        ADI_DEV_PDD_HANDLE         PDDHandle,
        ADI_DEV_BUFFER_TYPE        BufferType,
        ADI_DEV_BUFFER             *pBuffer
);
```

Arguments

`PDDHandle`	This is the handle used to identify the device.
`BufferType`	This argument identifies the type of buffer: one-dimensional, two-dimensional or circular.
`*pBuffer`	This is the address of the buffer or first buffer in a chain of buffers.

Return Value

`ADI_DEV_RESULT_SUCCESS`	The function completed successfully.
`ADI_DEV_RESULT_BAD_PDD_HANDLE`	The PDD handle does not identify a valid device.
`xxx`	This is device-specific return code.

Examples

Examples showing how to use the Device Driver Model as well as Analog Devices provided device drivers are provided with the Device Driver and System Services distribution. For examples of applications using the device drivers, see the `Blackfin/EZ-Kits` directory. Source code for all Analog Devices provided device drivers is located in the `Blackfin/Lib/Src/Driver` directory.

I INDEX